Expert Systems With Applications 122 (2019) 27–42

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

Iterated feature selection algorithms with layered recurrent neural network for software fault prediction

Hamza Turabieh [a,*], Majdi Mafarja [b], Xiaodong Li [c]

[a] Department of Information Technology, Taif University, Taif, Saudi Arabia
[b] Department of Computer Science, Birzeit University, Birzeit, Palestine
[c] School of Science, RMIT University, Melbourne, Australia

ARTICLE INFO

Article history:
Received 3 February 2018
Revised 18 December 2018
Accepted 19 December 2018
Available online 25 December 2018

Keywords:
Software fault prediction
Feature selection
Layered recurrent neural network

ABSTRACT

Software fault prediction (SFP) is typically used to predict faults in software components. Machine learning techniques (e.g., classification) are widely used to tackle this problem. With the availability of the huge amount of data that can be obtained from mining software historical repositories, it becomes possible to have some features (metrics) that are not correlated with the faults, which consequently mislead the learning algorithm and thus decrease its performance. One possible solution to eliminate those metrics is Feature Selection (FS). In this paper, a novel FS approach is proposed to enhance the performance of a layered recurrent neural network (L-RNN), which is used as a classification technique for the SFP problem. Three different wrapper FS algorithms (i.e, Binary Genetic Algorithm (BGA), Binary Particle Swarm Optimization (BPSO), and Binary Ant Colony Optimization (BACO)) were employed iteratively. To assess the performance of the proposed approach, 19 real-world software projects from PROMISE repository are investigated and the experimental results are discussed. Receiver operating characteristic - area under the curve (ROC-AUC) is used as a performance measure. The results are compared with other state-of-art approaches including Naïve Bayes (NB), Artificial Neural Network (ANN), logistic regression (LR), the k-nearest neighbors (k-NN) and C4.5 decision trees, in terms of area under the curve (AUC). Our results have demonstrated that the proposed approach can outperform other existing methods.

1. Introduction

Software Fault Prediction (SFP) is the process of predicting the fault-prone modules for the future releases of software versions being developed, depending on predefined software metrics or historical fault datasets (from previous projects) (Catal, 2011; Porter & Selby, 1990). The SFP process becomes easier with the adoption of the Agile Software Development (ASD) (Fowler & Highsmith, 2001) methodologies (e.g., Agile Unified Process, Extreme Programming, Scrum and Kanban) rather than the traditional methodologies (e.g., waterfall model (Royce, 1987), software development (Hoda, Salleh, Grundy, & Tee, 2017; Stavru, 2014)). In ASD the incremental delivery of the software opens the door for rapidly adapting the volatile requirements, and increasing the opportunities for collaboration between business owners and software developers (Hoda et al., 2017). Moreover, adopting ASD methodologies allows conducting software engineering activities (maintenance, review, refactoring or testing) synchronously with the development process.

Predicting faults in software subsystems (modules, components, classes, etc.) in the earlier stages (before delivering them to the end user), plays a vital role in reducing the time and effort costs required to accomplish the project, since it reduces the number of modules to be processed in each activity, and eliminates the unnecessary efforts in finding faults during the development process. The importance of SFP comes from the fact that delivering a software version with some faults will affect the subsequent versions. This is because there is a distinct relation between the different versions of the software products.

Implementing SFP in the early stages of the system development process is used to eliminate the possible faults in the future releases of the software. SFP methods vary from depending on the software metrics to Machine Learning (ML) and soft computing (SC) techniques (Rathore & Kumar, 2017a). The software metrics-based methods use some predefined metrics to predict the faults in a given software accordingly (Sheskin, 2003). This approach has a major drawback that if we built a (learning) model, and reused it to predict the faults in different projects with (possibly) the same

* Corresponding author.
E-mail addresses: h.turabieh@tu.edu.sa (H. Turabieh), mmafarja@birzeit.edu (M. Mafarja), xiaodong.li@rmit.edu.au (X. Li).

https://doi.org/10.1016/j.eswa.2018.12.033
0957-4174/

values of the used metrics, it may detect faults in the same areas of the software (Zimmermann, Nagappan, & Zeller, 2008). As an alternative approach, some researchers proposed the use of software change metrics that take into consideration the historical changes in the project to build the prediction models (Sheskin, 2003; Zimmermann et al., 2008). However, those methods become impractical and a time-consuming process when dealing with complex systems that emerged in a wide range of industries (e.g., nuclear power plant, defense, command and control, and medicine). This is because those industries are becoming increasingly dependent on complex software systems due to the advancement in the software development tools. Predicting the possible faults at the early stages of the software development process will substantially help reduce the computation and development cost. In literature, many studies have shown promising results in using SC methods for SFP (Czibula, Marian, & Czibula, 2014; Erturk & Sezer, 2015; 2016; Rathore & Kumar, 2017b).

SC methods showed superior performance in extracting useful knowledge from imprecise and unconstrained data when solving real-life problems. Machine learning (ML) is one of the data-driven soft computing techniques that has been widely used to predict the faulty modules in software projects. Representative ML methods include Logistic Regression (Li & Henry, 1993), Support Vector Machine (Xing, Guo, & Lyu, 2005), Naive Bayes (Alshayeb & Li, 2003), Neural Network (Oliveira, Pontes, Sartori, & Embiruçu, 2017), and Ensemble classifiers (Rathore & Kumar, 2017b; 2017c). Malhotra in Malhotra (2015) conducted a systematic literature review on several ML techniques that were used in SFP. He concluded that ML-based SFP methods showed high capability in predicting the faultiness of a module in a software product and outperformed the traditional statistical methods.

Artificial Neural Network (ANN) is one of the most widely-used ML models in predicting the faultiness of software components in the early stages of the software development lifecycle (SDLC) (Erturk & Sezer, 2016; Maren, Harston, & Pap, 2014). In general, ANN comes in several flavors such as feedforward neural network, radial basis function neural network, and recurrent neural network. Each type of ANN is able to solve complex problems in a different manner. In particular, recurrent neural network shows a competitive performance in solving several classification problems (Maggiori, Charpiat, Tarabalka, & Alliez, 2017), as it considers previous input values through its training process. For SFP, several researchers applied ANN with predefined metrics (features) as inputs to ANN (Chatterjee, Nigam, & Roy, 2017; Erturk & Sezer, 2016). It is difficult to say that these predefined metrics are suitable for developing a good classifier. As a result, finding the most suitable metrics will have a significant effect on the ANN performance, depending on the datasets.

One of the major issues that affect the performance of the learning algorithms is data dimensionality. High dimensional data may contain irrelevant and/or redundant features, that may mislead the learning algorithm, hence decrease its performance. Eliminating these irrelevant or redundant features helps to increase the learning algorithms' performance, and to reduce the computational time required for the prediction process. Feature selection (FS) is one possible solution to reduce the dimensionality of data without decreasing the performance of the learning algorithm (Liu & Motoda, 2012). FS plays a vital role in computational intelligence (Rauber, de Assis Boldt, & Varejão, 2015). FS aims to reduce the dimensionality of data by removing noisy, irrelevant and redundant data. Moreover, FS promotes better understanding of data (Dash & Liu, 1997). One of the main findings in a literature review for the SFP tools (Hall, Beecham, Bowes, Gray, & Counsell, 2012), is that FS improves the performance of the SFP models that employed ML techniques.

Several approaches either use a fixed number of features or all features to predict the fault in software projects. Projects are not similar to each other, using all features or a fixed number will not give a high-performance classifier over all the datasets. As a result, the motivation of this research is to propose a novel approach that will select the most valuable features by exploring the dataset using different feature selection algorithms randomly to enhance the performance of the software fault prediction classifier.

The rest of this paper is organized as follows: a literature review of SFP techniques is presented in Section 2. In Section 3, a background on feature selection algorithms is presented. A detailed discussion of the proposed methodology and the employed SC techniques are presented in Section 4. In Section 5, the datasets that are used in this study for software fault prediction are described. The performance measurement is presented in Section 6. The obtained results and results analysis are presented in Sections 7 and 8. Finally, a summary of our research findings, conclusions, and future works are presented in Section 9.

2. Related works

In literature, many Machine Learning (ML) techniques used for tackling the SFP problem can be found, including Decision Trees (DT) (Khoshgoftaar & Seliya, 2003), Artificial Neural Networks (ANN) (Thwin & Quah, 2005), Naive Bayes (NB) (Menzies, Greenwald, & Frank, 2007), Bayesian Network (BN) (Carrozza, Cotroneo, Natella, Pietrantuono, & Russo, 2013), Case-based Reasoning (CR) (El Emam, Benlarbi, Goel, & Rai, 2001), Fuzzy Clustering (FC) (Yuan, Khoshgoftaar, Allen, & Ganesan, 2000), Multilayer Perceptron (MLP) (Carrozza et al., 2013), Logistic Regression (LR) (Yuan et al., 2000) and Support Vector Machine (SVM). Different approaches were followed in designing SC-based methods to solve the SFP problem. For instance, (Cahill, Hogan, & Thomas, 2013) used two different classifiers (namely SVM and NB) to classify the software modules. They proposed a new ranking method called Rank Sum to validate the obtained results. Different datasets from NASA repository were used to test the proposed approaches. Their results suggested that NB performs better than SVM.

Moreover, Carrozza et al. (2013) implemented five different prediction and regression models (i.e., DT, BNs, MLP, SVM and NB), and analyzed their performance based on different datasets from NASA repository. In addition, they proposed new software complexity metrics for predicting Mandelbugs in complex systems along with the traditional metrics. K-fold cross-validation and confusion matrix was used. SVM and MLP are reported as the best approaches. In Malhotra (2014), a comparative study between a set of SFP methods was provided, where several ML and LR algorithms were implemented to predict the software faults in AR1 and AR6 datasets from the PROMISE repository. A deep comparison between implemented methods was presented and DT outperformed the LR models and other ML algorithms.

A multi-strategy classifier (RB2CBL) that integrates Rule-Based (RB) model with Case-Based Learning (CBL) model was proposed by Khoshgoftaar, Xiao, and Gao (2014). Moreover, Genetic Algorithm (GA) was used to optimize the parameters of the CBL models. In Rathore and Kumar (2017b), the idea of ensemble learners methods (i.e., LRCR and GRCR) was employed, and the performance of those models was evaluated based on Average Absolute Error (AAE) and Average Relative Error (ARE). This ensemble approach is evaluated on different datasets selected from PROMISE data repository and Eclipse bug data repository, and it outperformed other learning-based methods. Moreover, the GRCR based ensemble method outperformed the LRCR based method.

In Erturk and Sezer (2016), a two phases system for SFP was proposed, where Fuzzy Inference System (FIS) was employed at the first phase (no data available about the project), and then

iterative Artificial Neural Network (ANN) is employed in the second phase when some data about the project was collected from the earlier phases. The iterative model outperformed other models and the authors claimed that it can be adopted as on-line approach to predict the faults in software. Recently, Shatnawi (2017) proposed the use of receiver operating characteristic (ROC) analysis as an SFP tool, where a threshold is defined to classify software modules to fault-prone and not fault-prone.

Some preprocessing techniques were employed before applying the learning algorithm. In Cotroneo, Natella, and Pietrantuono (2013), different datasets from three projects (Linux Kernel, MySQL DBMS, and CARDAMOM) were considered to collect the Aging-Related Bugs. Several classification algorithms were employed to predict the software faults. The experiments were conducted in two phases, the first was on the plain datasets (without preprocessing). In the second phase, the datasets were preprocessed with a logarithmic transformation. The preprocessing enhanced the performance of the learning algorithms, such that NB with logarithmic transformation was reported as the best performing classifier (Cotroneo et al., 2013).

Fifteen variants of NB classifier were tested on NASA and Eclipse datasets and compared with many well-known classification algorithms by Dejaeger, Verbraken, and Baesens (2013). Markov blanket feature selection method was used as a preprocessing step. Using AUC and the H-measure as performance measurements, it was found that the augmented NB classifiers perform similarly or better than the traditional NB classifiers, and both outperformed the other tested methods. Okutan and Yıldız (2014) proposed a new approach to select the most important metrics based on the Bayesian networks that were used to define the relationships among software metrics and defect proneness. Another similar approach was proposed in Chen et al. (2014), where five different FS and sample reduction methods were used as preprocessing steps to simplify the training process, and three classification algorithms (NB, C4.5, KNN) were tested on Eclipse project and some datasets from NASA repository. In Jin and Jin (2015), Quantum based PSO (QPSO) feature selection method was applied as a preprocessing step, followed by ANN classifier, to predict software fault-proneness. Using some datasets from NASA repository, the obtained results showed that ANN with QPSO performed better than other approaches. Recently, Miholca, Czibula, and Czibula (2018) proposed a hybrid model (called HyGRAR) that hybridized gradual association rule mining and ANNs. For more comprehensive studies about SFP tools, readers can refer to Catal and Diri (2009), Radjenović, Heričko, Torkar, and Živkovič (2013), Hall et al. (2012), Hoda et al. (2017).

Studying the presented works above, it is clear that using a preprocessing technique on the dataset significantly affected the performance of learning algorithm. FS is an important preprocessing step that reduces the number of features in the dataset, which helps in solving many problems like the overfitting that may compromise the performance of the learning algorithm. Moreover, it was proved that metaheuristic algorithms proved their ability to tackle FS problems efficiently and better than other search strategies (i.e., complete and random strategies).

Due to the stochastic nature of the meta-heuristic algorithms, they cannot guarantee to find the optimal feature subset, and according to the No Free Lunch (NFL) theorem in the optimization field, there is no algorithm that can solve all optimization problems efficiently. In this paper, a pool of meta-heuristics-based FS methods (i.e, GA, PSO and ACO) is used to tackle this problem. The selection of these three algorithms is due to some characteristics of each of them, i.e., the exploration process in GA is better than in PSO and ACO, while PSO makes a balance between the exploration and exploitation process. ACO focuses more on building the solution (exploitation process) than on exploration.

3. Background on feature selection methods

FS is considered as a combinatorial optimization problem, that aims to search for the optimal subset of features from the original dataset, that still faithfully represents the original data. The general FS process consists of two main steps: (i) searching for the minimal reducts, and (ii) evaluating the selected features. From evaluation perspectives, FS algorithms are classified in two main categories: *filter* and *wrapper*. Filter methods consider only the relations between features to evaluate the feature subsets, which makes them faster than the wrapper methods that evaluate the feature subsets by employing a learning algorithm according to a validation method. In general, wrapper methods usually obtain better results than filter methods (Kudo & Sklansky, 2000).

The main challenge of FS methods is how to search for the best subset of features that perfectly represents the original data. Fortunately, FS can be viewed as a search problem, where each point in the search space represents a feature subset. Formally speaking, if a dataset contains three features, e.g., (A, B, C), one of the possible methods to represent a feature subset is to use a binary array with three elements (i.e., number of features in the dataset). If an element has a value equal to 1, then the corresponding feature is included (selected) in the feature subset, otherwise the value is 0, which means it is not selected. Hence (1, 1, 1) means that the three features are selected, while (0, 1, 0) indicates that only the second feature is selected. Since FS is NP-hard problem, large problems are not easy to be solved by using an exact search method. For example, if the dataset includes n features, then 2^n subsets will be generated and assessed. Therefore, heuristic search strategies are often employed as an alternative solution to reduce the computational costs.

Meta-heuristic algorithms, the higher level of heuristics, are stochastic search methods that have demonstrated superior performances in tackling feature selection problems when compared to exact methods (Ezgi & Selma, 2016; Guyon & Elisseeff, 2003). According to (Talbi, 2009), meta-heuristics can be classified into two main types depending on the number of solutions to be processed in each iteration of the optimization process; single-solution based methods (S-based), (e.g., Simulated Annealing (SA) (Van Laarhoven & Aarts, 1987), Tabu Search (TS) (Glover, 1986)), and population-based methods (P-based), (e.g., Genetic Algorithm (GA) (Holland, 1992), Particle Swarm Optimization (PSO) (Kennedy & Eberhart, 1995), and Ant Colony Optimization (ACO) (Maniezzo, 1992)). As the names imply, in single-solution based methods, one solution is manipulated and transformed into a new solution in each iteration. In contrast, a set of solutions is evolved in the population-based methods. The main difference between S-based and P-based is that P-based methods are more exploration oriented methods; i.e., trying to explore the search space as broadly as possible in the hope to find more promising regions, while S-based methods are more exploitation oriented methods; i.e., trying to fine-tune a specific solution in its neighborhood area in the hope of finding the global optimal value.

Meta-heuristic algorithms are able to find the optimal or near-optimal solutions for a certain problem in a reasonable time. However, each algorithm has some strengths and weaknesses. For example, S-based algorithms are not able to provide a high-quality solution for complex problems with high dimensional search space, where the search space increases in an exponential manner with the problem size. As a result, S-based algorithms are not practical for complex problems. While P-based algorithms are able to search different parts of the fitness landscape, act as a low-pass filter of the landscape, ignoring local distractions (Prugel-Bennett, 2010).

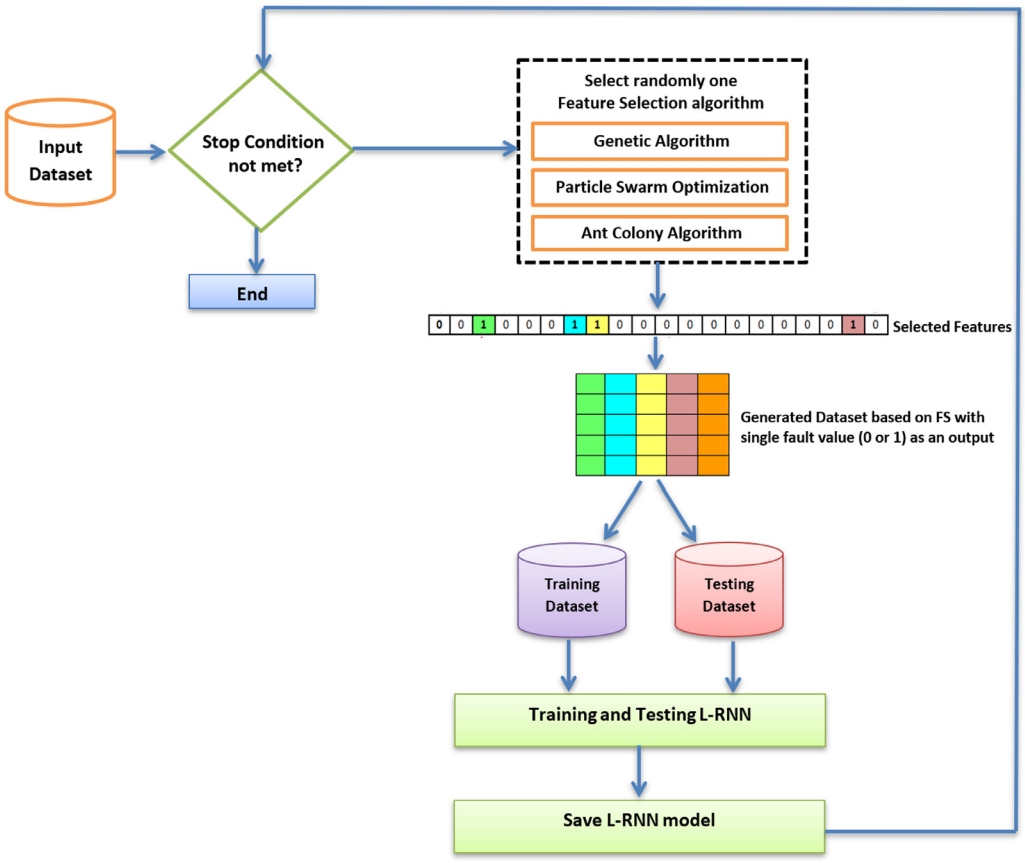

Fig. 1. A pictorial diagram of the proposed methodology.

4. Proposed methodology

As mentioned earlier, SFP is a real-world problem that depends on historical datasets of previous (completed) projects. In general, ML algorithms showed good results when compared with the traditional algorithms (Talbi, 2009). The performance of ML algorithms is highly dependent on the data dimensionality. Therefore, finding the most significant features and removing the unnecessary features will be critical in finding an efficient and robust classification or prediction model. Since the goal of using a FS method is to enhance the performance of the learning algorithm, wrapper FS methods are more suitable than filters methods. In wrapper FS methods, the selection criterion is the performance of the learning algorithm (e.g., classifier).

In this paper, a wrapper feature selection algorithm that depends on a layered recurrent neural network as an evaluator is proposed, as depicted in Fig. 1. The algorithm starts by taking the SFP dataset as an input and then executes the proposed algorithm over a number of iterations. At each iteration, one of the three employed FS algorithms (i.e., BGA, BPSO or BACO) is selected randomly, in order to find the most valuable metrics. Then the proposed algorithm will generate the training and testing dataset based on the selected metrics. A layered recurrent neural network is iterated for a predefined number of iterations and evaluates the obtained model using testing data. If the obtained result reaches an optimal value or the maximum number of iterations, the algorithm will stop. Otherwise, it starts a new iteration.

When designing any optimization algorithm, two key issues should be taken into consideration; the solution representation and the fitness function. In this work, the solution is encoded as a binary vector of length equal to the number of features as shown in Fig. 2. The 0 value indicates that the feature is not selected, while 1 indicates that the feature is selected. As FS process aims to reduce the number of selected features while increasing the accuracy of the classification algorithm, the designed fitness function should consider both issues. In the proposed FS algorithms the ANN is used as evaluator. The fitness function can be seen in Eq. (1):

$$Fitness = E * (1 + \beta * \frac{|R|}{|N|}) \quad (1)$$

where E is the overall error rate (for training and testing) that was obtained from the ANN classifier, β is a user defined variable (here, $\beta = 5$), $|R|$ represents the number of the selected features, $|N|$ is the total number of features is a dataset. The training dataset is used for computing the gradient and updating the network weights and biases. The validation dataset is used through training process to avoid overfitting problem. The error on the validation set is monitored during the training process. The validation error normally decreases during the initial phase of training, as does the training set error. However, when the network begins to overfit the data, the error on the validation set typically begins to rise. The network weights and biases are saved at the minimum of the validation set error. The testing dataset is used to evaluate the obtained model.

The parameter settings of the employed ANN classifier are presented in Table 1. In the following subsections, the main components of the proposed algorithm are described.

4.1. Binary genetic algorithm

Genetic Algorithm (GA) is an evolutionary algorithm that simulates the process of natural selection (Holland, 1992). GA is a

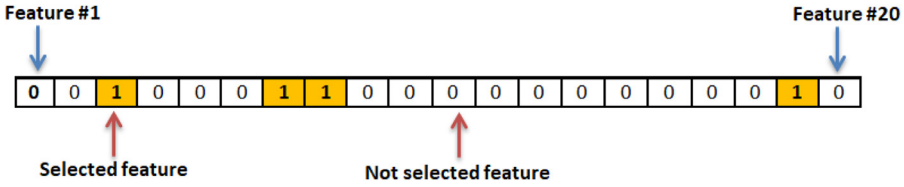

Fig. 2. An example of a feature selection solution.

Table 1
Parameters setting for ANN internal classifier.

Parameters	Values
Number of neurons input layer	Number of selected features
Number of neurons hidden layer	10
Number of neurons output layer	1
Training sample	70% of the data
Testing sample	15% of the data
Validation sample	15% of the data
Fitness function	Mean square error

Table 2
The parameters setting for BGA.

Parameters	Values
Number of iterations	3000
Population size	40
Crossover rate	0.7
Mutation rate	0.1
Selection type	Roulette Wheel Selection
Crossover type	single, double or uniform

Given:
-nP: base population size.
-nI: number of iterations.
-rC: rate of crossover.
-rM: rate of mutation.
Generate initial population of size nP.
Evaluate initial population according to the fitness function.
While ($current\_iteration \leq nI$)
//Breed $rC \times nP$ new solutions.
Select two parent solutions from current population.
Form offspring's solutions via crossover.
IF($rand(0.0, 1.0) <$ rM)
Mutate the offspring's solutions.
end IF
Evaluate each child solution according to the fitness function.
Add offspring's to population.
//population size is now MaxPop=nP× (1+rC).
Remove the rC× nP least-fit solutions from population.
end While
Output the global best solution

Fig. 3. The pseudo-code for Genetic Algorithm.

P-based algorithm, and the best solution is obtained after a sequence of iterative steps. In GA, the optimization process starts by generating a set of solutions that represent the initial population, then genetic operators (i.e., selection, crossover, and mutation) are applied on selected solutions from the population. This step is repeated iteratively until satisfying a stopping condition; i.e., the optimal or near optimal solution is found or a predefined number of iterations is reached. In each iteration, the solutions are evaluated using a fitness function which estimates the significance of the obtained solution (Huang & Wang, 2006).

Crossover and mutation are the main evolutionary operators inside GA. During the optimization process, these operators are applied to the individuals in the population to produce new solutions, reflecting the performance of GA. Generating new solutions starts by selecting two parents from the population pool, based on a specific selection mechanism (e.g., random, roulette wheel, or tournament), then crossover operator (e.g., single, double, uniform) is applied to these parents which then produce two offsprings. In mutation, local changes (e.g., randomly) are applied to the both offsprings. Then the population is updated by considering the produced offsprings that replace some solutions in the population based on an elitism replacement strategy. The updated population is then considered for the next iteration. Fig. 3 shows the pseudo-code for GA.

Table 2 shows the parameters setting for the Binary Genetic Algorithm (BGA) that is used in this paper. A Roulette Wheel Selection (RWS) is used as a selection approach for selecting two parents. Three types of crossover (e.g., single, double, uniform) are implemented and one is randomly selected for each iteration to enhance the exploration process. Fig. 4 simulates the BGA for a single iteration.

4.2. Binary particle swarm optimization

Kennedy and Eberhart (1995) proposed the Particle Swarm Optimization (PSO) algorithm. The main idea behind PSO is to mimic the social behavior of organisms such as bird flocking and fish schooling. In a PSO algorithm, particles (which represent candidate solutions) are flying around, in a multi-dimensional search space. The positions of these particles are adjusted according to particles' own memories and the best-fit particle of the neighboring particles. More specifically, each particle adjusts its position x_{id} and velocity v_{id} according to the best position visited so far (i.e., p_i) and the best position in the neighborhood (i.e., p_g). Note that i is the index of a particle in the swarm ($i = 1, \ldots, Sn$), Sn is the size of swarm, d is the dimension index of a particle (candidate solution), ($d = 1, \ldots, m$), and t represents the iteration number. The velocity and the position of a solution are updated based on Eqs. (2) and (3), respectively. Note that w is a positive inertia weight, r_1 and r_2 are randomly generated numbers between 0 and 1 at each iteration, and c_1 and c_2 present the degree of influence of p_{id} and p_{gd} on the particles velocity, respectively. To control the velocity from flying out of the search space, the velocity v is bounded within range of $[v_{\min}, v_{\max}]$. Every newly visited position represents a new solution, which is also used for updating the global best solution pg. Fig. 5 shows the pseudo-code for the PSO algorithm.

$$v_{id}(t+1) = wv_{id}(t) + c_1 r_1[p_{id}(t) - x_{id}(t)] + c_2 r_2[p_{gd}(t) - x_{id}(t)]. \quad (2)$$

$$x_{id}(t+1) = x_{id}(t) + v_{id}(t+1). \quad (3)$$

In the continuous version of PSO, particles' movements in the search space are defined by the update rule Eqs. (2) and (3). However, FS is a binary optimization problem, where the variable

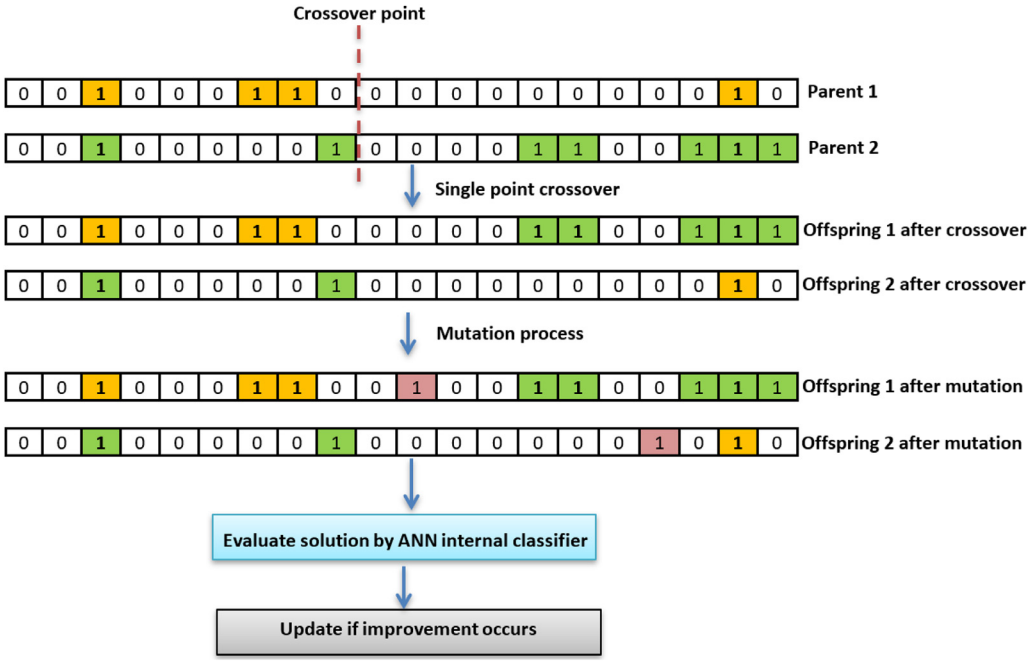

Fig. 4. An example of a Binary Genetic Algorithm for a single iteration.

Given:
-Sn: swarm size.
-t: number of iterations.
-v: initial velocity.
-x: initial position.
-c1: degree of influence of p_{id}.
-c2: degree of influence of p_{gd}.
initialize particle( )
While ($current\_iteration \leq t$)
Evaluate each particle's position according to the fitness function.
Find the best solution of each particle so far.
update the global best solution.
update the velocity of each particle based on Eq. (2).
update the position of each particle based on Eq. (3).
end While
Output the global best solution

Fig. 5. The pseudo-code for Particle Swarm Optimization.

values are restricted to 0 or 1. In (Kennedy & Eberhart, 1997), a binary version of PSO (BPSO) that can be used to solve binary optimization problems was proposed. In BPSO, a sigmoid transfer function (TF) is used to convert the continuous variables to binary ones. The continuous values of the velocity vector are fed into the TF to produce a probability value that converts each element of the position vector to 0 or 1 based on Eq. (4):

$$S(v_{id}(t+1)) = \frac{1}{1+e^{-v_{id}(t)}} \tag{4}$$

where V_i^d represents the velocity value of the dth dimension in the ith vector, and t represents the current iteration.

The position of the current particle is updated according to Eq. (5) based on the probability value $S(v_{id}(t+1))$ obtained from Eq. (4):

$$x_{id}(t+1) = \begin{cases} 1 & \text{if } rand(0.0, 1.0) < S(v_{id}(t+1)) \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

where $x_i^d(t+1)$ represents the element in the dth dimension in the ith position in the next iteration, $rand()$ is a function that generates a random number in the interval 0 and 1. Table 3 shows the parameters setting used for BPSO in this paper. Fig. 6 simulates the BPSO for a single iteration.

Table 3
The parameters setting for BPSO.

Parameters	Values
Number of iterations (t)	3000
Swarm size (Sn)	40
degree of influence (c1)	1.5
degree of influence (c2)	1.5
vmax	1
vmin	0
Inertia weight (w)	0.8

4.3. Binary ant colony optimization

Ant colony optimization (ACO) is a population-based search algorithm that simulates the behavior of real ants in order to find the shortest path. Ant movements are based on a pheromone, which is deposited on some paths by some other ants. ACO is widely-used for solving pathfinding problems, scheduling problems, fuzzy control network routing, and image processing, etc. (Dorigo & Caro, 1999; Mohan & Baskaran, 2012). ACO starts by creating a set of agents (ants) located at different positions of the search space, which can be used to build a candidate solution. Eq. (6) shows the probabilistic transition rule, which is the probability of ant k to determine the next move (include feature i) in its built solution at time step t based on the heuristic information and pheromone values, which is the local trial updates for each ant.

$$P_i^k(t) = \begin{cases} \frac{[\tau_i(t)]^\alpha . [\eta_i(t)]^\beta}{\sum_{u \in J^k} [\tau_u(t)]^\alpha . [\eta_u(t)]^\beta} & \text{if } i \in J^k \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

where τ_i and η_i are two values that present the pheromone value and heuristic desirability associated with feature i; respectively. J^k is the set of acceptable features that can be added to the partial solution. α and β are two values that determine the relative importance of the pheromone value and heuristic information.

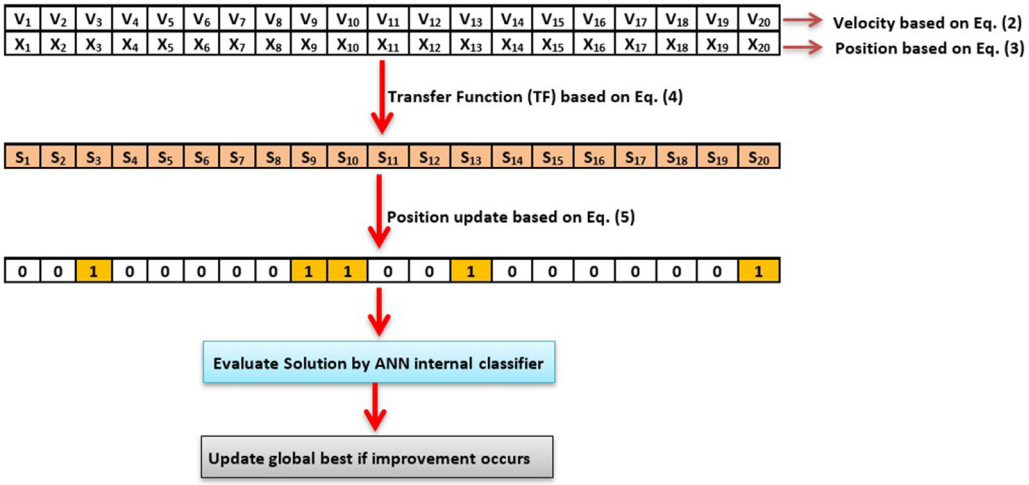

Fig. 6. An example of a BPSO for a single iteration.

Given:
-nA: number of ants.
-nI: number of iterations.
-iP: initial Pheromone.
Generate initial population of size nA (ants).
Initialize the pheromone trail and parameters.
Evaluate initial population according to the fitness function.
Find the best solution of the population.
While (*current_iteration* $\leq nI$)
Do Until *each ant completely builds a solution*
local trial update.
End Do
Update the pheromone.
Determine the best global ant.
end While
Output the global best solution

Fig. 7. The pseudo code for Ant Colony Optimization.

Table 4
The parameters setting for BACO algorithm.

Parameters	Values
Number of iterations	3000
Number of ants (agents)	20
Initial pheromone	1
Pheromone Exponential Weight (α)	0.8
Heuristic Exponential Weight (β)	0.8
Evaporation Rate	0.6

The pheromone values are updated once all agents finish their movements (constructing solutions) based on Eq. (7), where $f(P_i^k(t))$ is the cost of the current solution at time step t. This process is repeated until a stop condition is met. The new solutions found by the agents substitute the old population using an elitism replacement strategy for the next iteration. Fig. 7 shows the pseudo-code for the ACO algorithm.

In binary Ant colony optimization (BACO), each individual ant is represented by a binary bit string $x = (x_1, x_2, \ldots, x_n)$, $x_i \in \{0, 1\}$. Each ant k at a bit j generates a solution based on a probability distribution as shown in Eq. (8), where τ_i represents the pheromone level for position x_i to select $s \in \{0, 1\}$. Fig. 8 simulates a BACO algorithm for a single iteration, where an ant (agent) starts building a solution based on touring concept. Table 4 shows the parameters setting for BACO used in this paper.

$$\tau_i^k(t+1) = \tau_i^k(t) + \frac{1}{f(P_i^k(t))} \tag{7}$$

$$P_i^k(t) = \begin{cases} \frac{\tau_i(t)}{\sum_{u \in J^k} \tau_u(t)} & \text{if } i \in J^k \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

4.4. Layered recurrent neural network

Artificial Neural Network (ANN) approach is a machine learning algorithm based on an abstraction model of the human brain. ANN has been widely used for solving challenging learning or classification problems. ANN is able to learn even if the input data is noisy or incomplete one. Once the ANN is trained, it can perform prediction instantaneously. ANN has been adopted in diverse areas such as robotics, power system, forecasting, optimization and manufacturing (Maren et al., 2014). In this paper, we adopt a Layered Recurrent Neural Network (L-RNN) structure that is able to learn the previous input data adaptively based on layer recurrence structure.

L-RNN has been successfully applied in several complex domains such as image processing (Zhang, Yin, Zhang, Liu, & Bengio, 2017), industrial problems (Qin, Yang, Xue, & Song, 2017) and forecasting (Ruiz, Rueda, Cuéllar, & Pegalajar, 2018; Senjyu, Takara, Uezato, & Funabashi, 2002), since it considers the past values through the training process. Moreover, L-RNN is able to solve complex problems with a set of correct weights. L-RNN has a dynamic memory, i.e., the information can be temporally memorized in the L-RNN model. Basically, the learning process of L-RNN is a time-varying pattern, applying either feed-forward or feedback connections. Moreover, training L-RNN is similar to training a standard neural network, but with a little twist. Each output depends not only on the calculations of the current time step, but also the previous time steps. As a result, outputs of some nodes are inputs to other nodes providing repeated feedback to the network. Feedback nodes remember the values of the previous stage; thus, a new output will depend on previous and current input data (Lipton, 2015).

An example of the basic structure of L-RNN is shown in Fig. 9. Here we illustrate L-RNN at time t. Given an input sequence $\mathbf{L} = (L_1, \ldots, L_t)$, a standard L-RNN computes the hidden vector sequence $\mathbf{P} = (P_1, \ldots, P_t)$ and output vector sequence $\mathbf{y} = (y_1, \ldots, y_t)$ by iterating over Eqs. (9) and (10).

$$P_t = f(W_{hL}L_t + W_{PP}P_{t-1} + b_P) \tag{9}$$

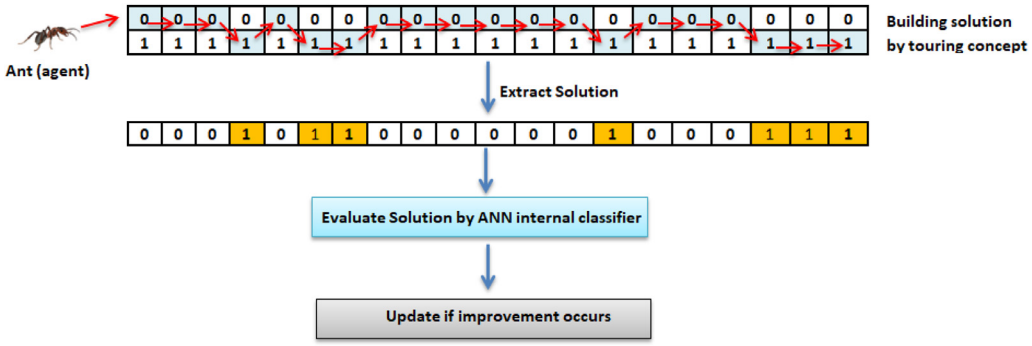

Fig. 8. An example of BACO for a single iteration.

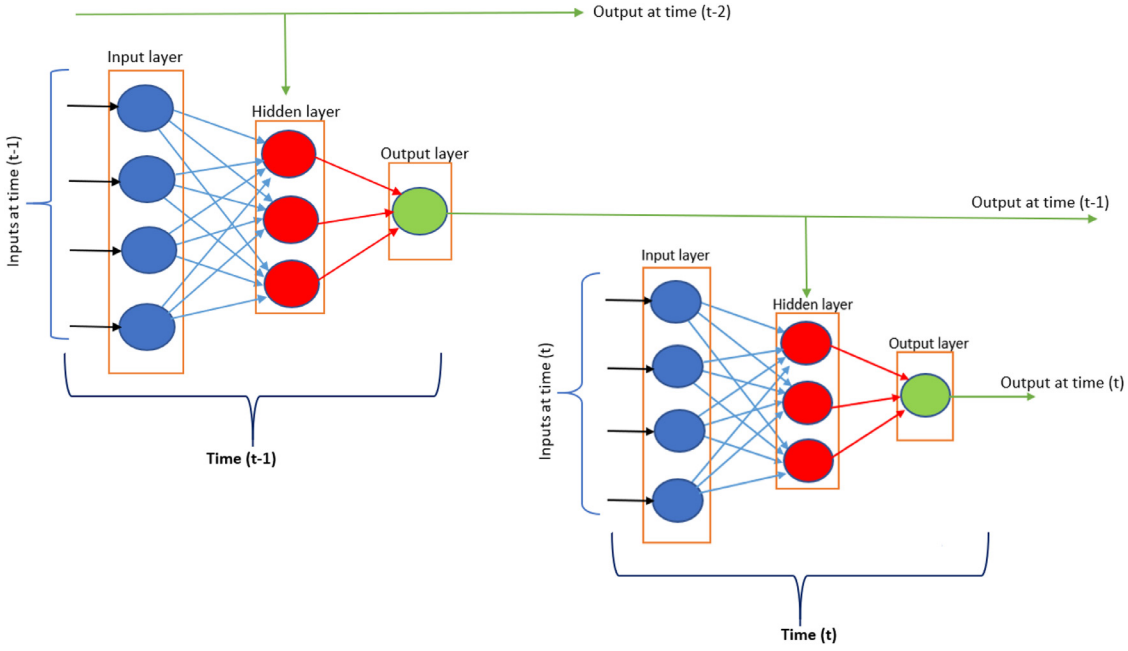

Fig. 9. An example of Layer Recurrent Neural Network (L-RNN).

$$y_t = f(W_{yh}P_t + b_y) \tag{10}$$

where $f()$ is an activation function (sigmoid function), three weight matrices: (i) W_{hL}: which is a matrix that presents the conventional weights between input layer and a hidden layer, (ii) W_{PP}: which is a matrix that presents the weights between a hidden layer and itself at adjacent time steps and (iii) W_{yh}: which is a matrix that presents the weights between a hidden layer and output layer. b_P and b_y are vectors that present bias parameters which help each recurrent neuron to learn an offset.

Generally speaking, there are two types of training algorithms for L-RNN, back-propagation through time and real-time recurrent learning. Back-propagation through time is used to alter the network structure between feedback structures to feed-forward structures. Whilst real-time recurrent learning applies the same set of weights recursively over the network structure. In this paper, we use back-propagation through time. Table 5 shows the parameters settings for L-RNN used in this paper. A *threshold* value 0.5 is used to transform the final output. The output of L-RNN can be used to categorize classes into either faulty ($\geq threshold$) or not faulty ($<threshold$).

Table 5
The parameters setting for L-RNN.

Parameters	Values
Number of iterations	1000
Number of neurons in Input layer	number of features
Number of neurons in Hidden layer	number of features /2
Number of neurons in Output layer	1
Threshold value to transfer output	0.5

5. PROMISE datasets for software fault prediction

Several public datasets are available in the field of software fault prediction, such as the PROMISE (Tera-Promise., 2017), NASA (M.D.P, 2017) and AEEEM (D'Ambros, Lanza, & Robbes, 2010) datasets. Several software metrics have been proposed to investigate the quality of developed software such as object-oriented metrics, which are known as CK metrics suite (Chidamber & Kemerer, 1994). In this paper, we examined 19 real software fault projects from the PROMISE public software engineering repository (Jureczko & Madeyski, 2010; Tera-Promise., 2017), which are highly recommended by several researchers in software engineering. These datasets are noise free and have no missing values. Moreover, the selected projects have various sizes (i.e., having

Table 6
The details of the selected datasets.

Dataset	# of Instances	# of Defective Instances	Rate of defective Instances
ant-1.7	745	166	0.223
camel-1.0	339	13	0.038
camel-1.2	608	216	0.355
camel-1.4	872	145	0.166
camel-1.6	965	188	0.195
jedit-3.4	272	90	0.331
jedit-4.0	306	75	0.245
jedit-4.2	367	48	0.131
jedit-4.3	492	11	0.022
log4j-1.0	135	34	0.252
log4j-1.1	109	37	0.339
log4j-1.2	205	189	0.922
lucene-2.0	195	91	0.467
lucene-2.2	247	144	0.583
lucene-2.4	340	203	0.597
xalan-2.4	723	110	0.152
xalan-2.5	803	387	0.482
xalan-2.6	885	411	0.464
xalan-2.7	909	898	0.988

109–909 instances) and different percentages of defective instances (i.e., ranging from 2.2% to 98.8%). A brief summarization of all selected data is presented in Table 6. Each dataset has 20 different object-oriented metrics as input and a single fault value as an output variable. Table 7 presents the description of the 20 object-oriented metrics.

6. Performance measure

Several criteria are used to evaluate a classifier such as accuracy, precision, recall, F-measure and area under ROC curve (AUC). However, all mentioned criteria except AUC are influenced by a cut-off value on the predicted probability of defect instances. The default value of cut-off is 0.5, which may not be the best cut-off value while evaluating a classifier (Zhang, Mockus, Keivanloo, & Zou, 2016). While ROC is not related to the cut-off value and not affected by the skewness of faulted data (the ROC curves are insensitive to changes in class distributions. If the proportion of negative to positive cases changes in the test dataset, the ROC curves will not change) (Fawcett, 2004). Several researchers suggest using the AUC value for better evaluating any classifier since AUC is not affected by changing data distributions (Ghotra, McIntosh, & Hassan, 2015; Lessmann, Baesens, Mues, & Pietsch, 2008), hence we select the AUC value to evaluate the proposed classifier in this paper.

In short, AUC depends on the trade-off between True Positive (TP) rate against False Positive (FP) rate. The final AUC value can be calculated based on a confusion matrix formed as in Table 8. Where:

1. TP: Correctly predicted positive values where actual and predictive values are both Yes.
2. TN: Correctly predicted negative values where actual and predictive values are both No.
3. FN: When actual class is yes but predicted class in No.
4. FP: When actual class is no and predicted class is Yes.

From confusion matrix, two important values are used to calculate the AUC value: sensitivity and specificity, which are defined as follows.

$$Sensitivity = TP_{rate} = \frac{TP}{P} \tag{11}$$

$$Specificity = TN_{rate} = \frac{TN}{N} \tag{12}$$

where P is the number of actually positive samples and N is the number of actually negative samples. Moreover, AUC measures the probability that a randomly chosen defective entity ranks higher than a randomly chosen clean entity. Fig. 10 shows the proposed rules to evaluate any classifier using AUC (Hosmer & Lemeshow, 2000). AUC measures enable researchers to generalize the results even if the data distribution is changed (Koru, Emam, Zhang, Liu, & Mathew, 2008).

7. Experimental results

In this paper, we examine the performance of different feature selection algorithms in enhancing the software fault prediction system. To achieve this, several experiments were performed using MATLAB-R2014a. We applied the proposed iterated feature selection algorithms with L-RNN classifier over 19 different datasets from PROMISE repository (see Table 6). Two sets of experiments were performed (i) L-RNN without cross-validation, where the data is divided 80% for training and 20% for testing; (ii) L-RNN with cross-validation, where self-training is applied based on k cross-validation ($k = 5$). Each dataset is divided into 5 parts, where 4 parts (80%) were used for training and the remaining one (20%) was used for testing. Each set of experiments were tested with and without feature selection. Incase of without feature selection, all features were used as input to L-RNN. Each dataset has been examined 11 times. The following subsections present the obtained results.

7.1. Results without cross-validation

In this subsection, we present the experimental results of L-RNN classifier without cross-validation, as shown in Table 9. Two types of experiments have been performed, the first one without feature selection, while the second one with feature selection. Table 9 reports 4 values for each experiment: average AUC, best AUC, minimum AUC and median of AUC, respectively.

It is clear that L-RNN classifier without cross-validation and without feature selection is able to find acceptable results (AUC ≥ 0.7) in 14 datasets out of 19 based on best AUC value. While with feature selection only 3 datasets are acceptable. For example, L-RNN classifier without feature selection is able to obtain excellent results on jedit-4.0 dataset, where the average, best, minimum and median are 0.8442, 0.9655, 0.4310, 0.9023, respectively. While the results for the same dataset with feature selection are not good enough since best ROC value is 0.6364 which indicates a fair classifier. Figs. 11 and 12 present the boxplots diagram for testing datasets without and with feature selection, respectively; based on the average values, it is obvious that the overall performance is a poor classifier for both cases. However, the obtained results indicate that FS is not always able to enhance the performance of the classifier. Moreover, splitting the datasets into two parts (training and testing) may also create an over-learning problem (training for a long time, where the neural network extracts too much information from the individual cases forgetting the relevant information of the general case). To overcome this problem, a cross-validation has been applied by dividing dataset into k-fold number.

7.2. Results with cross-validation

The results of the L-RNN with cross-validation are presented in Table 10. The performance of the proposed algorithm with feature selection over camel-1.0, jedit-3.4, jedit-4.3, log4j-1.1 and xalan-2.7 datasets is outstanding based on the average AUC results. While without feature selection only one dataset (jedit-4.3) has an outstanding result. Based on the best AUC values, it is clear that

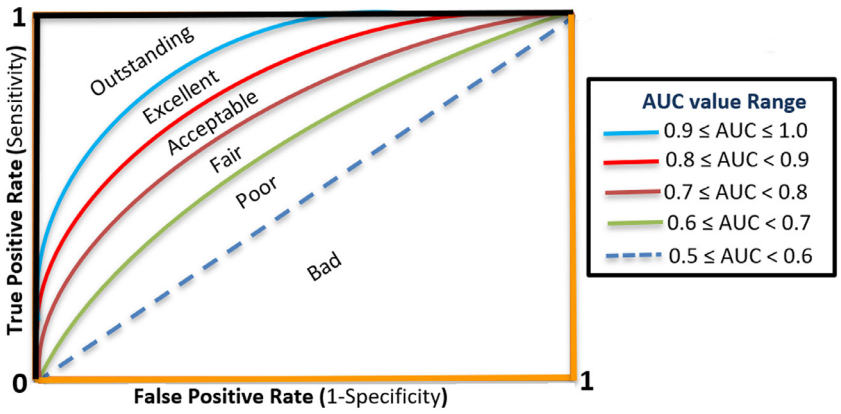

Fig. 10. The ROC curves and AUC values.

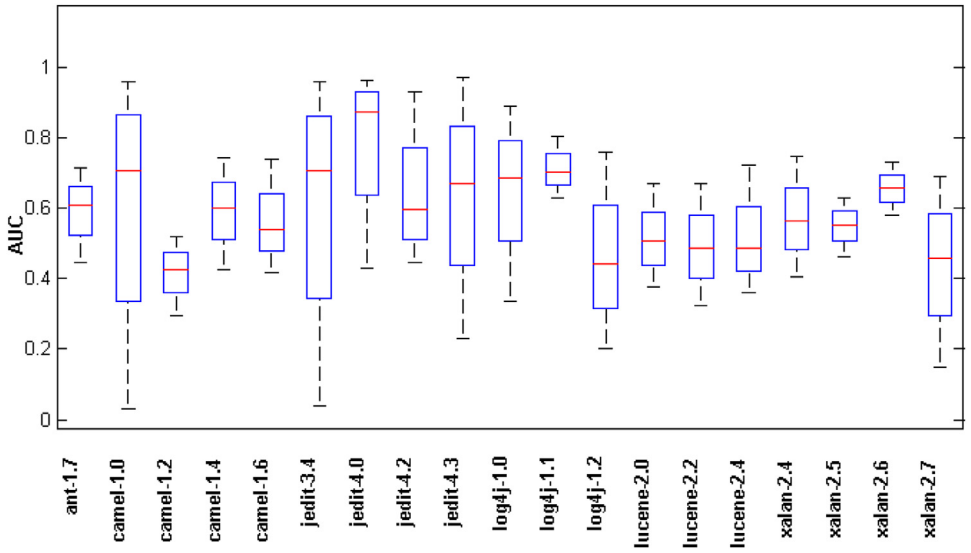

Fig. 11. Boxplots diagram without feature selection and without cross validation.

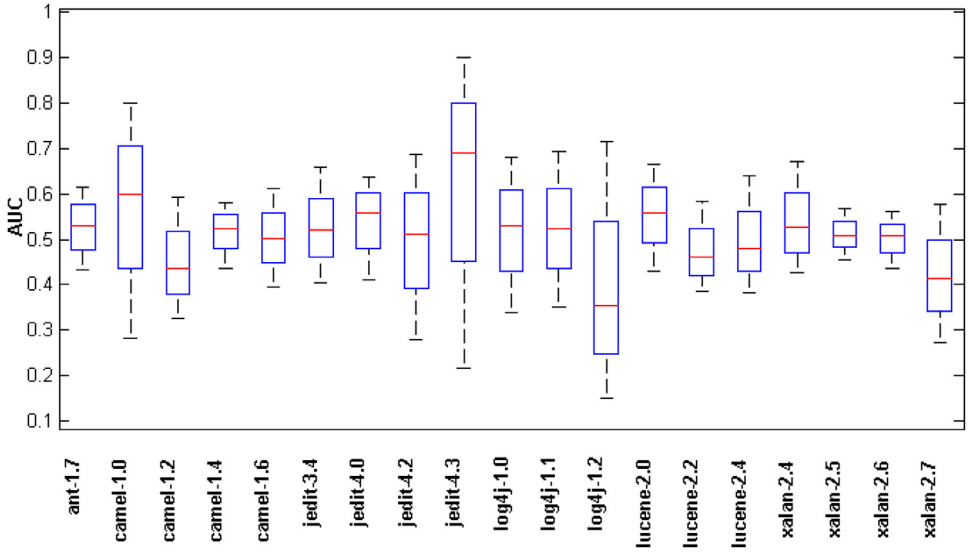

Fig. 12. Boxplots diagram with feature selection and without cross validation.

Table 7
Metrics description.

Metric	Description
wmc	Number of methods defined in a class.
dit	Depth of a class within the class hierarchy from the root of inheritance.
noc	Number of immediate descendants of a class.
cbo	Count the number of classes coupled to class.
rfc	Count the number of distinct methods invoked by a class in response to a received message.
lcom	Count the number of methods that do not share a field to the method pairs that do.
ca	Count the number of dependent classes for a given class.
ce	Count the number of classes on which a class depends.
npm	Number of public methods defined in a class.
lcom3	Count the number of connected components in a method graph.
loc	Count the total number of lines of code of a class.
dam	Computes the ratio of private attributes in a class.
moa	Count the number of data members declared as class type.
mfa	Shows the fraction of the methods inherited by a class to the methods accessible by the functions defined in the class.
cam	Computes the cohesion among methods of a class based on the parameters list.
ic	Count the number of coupled ancestor classes of a class.
cbm	Count the number of new or redefined methods that are coupled with the inherited methods.
amc	Measures the average method size for each class.
max_cc	Maximum counts of the number of logically independent paths in a method.
avg_cc	Average counts of the number of logically independent paths in a method.

Table 8
The confusion matrix.

	Predicted class		
Actual class		Class = Yes	Class = No
	Class = Yes	True Positive (TP)	False Negative (FN)
	Class = No	False Positive (FP)	True Negative (TN)

with feature selection outperforms without feature selection in 17 datasets.

Figs. 13 and 14 show the performance of the datasets based on boxplots diagram. The results demonstrate that the feature selection plays an important role in enhancing the results and robustness.

8. Result analysis

In this section, we present a detailed analysis of the obtained results with a comparison with the results of the well-known approaches in the literature. To evaluate the obtained results and to show how the number of used features affects the classifiers' performance, a statistical comparison using the Wilcoxon statistical test with a significance level of 0.05 was conducted as well. Table 11 presents the p-values of the obtained results. In this table, a *p*-value less than 0.05 indicates that there is a statistical difference between results. We can see that the obtained *p*-value is less than 0.05 for both tests.

By investigating the average of ROC results from Table 9, it is clear that the performance of the LRNN with all features outperforms the same approach with feature selection in all datasets except camel-1.2 and lucene-2.0 datasets. While comparing the average of ROC results from Table 10, we can see that applying feature selection is able to outperform the same approach without feature selection in all datasets. Moreover, the obtained average results with cross validation combined with feature selection algorithms outperforms all other reported results.

The models that do not use the cross validation strategy are simpler in implementation, require less training models and low computational time compared to those use the cross validation strategy. However, the cross validation strategy enables the model to be trained on all samples in the dataset, which yields to more accurate model. Adding feature selection approach reduces the

Table 9
Results of L-RNN classifier without cross-validation.

Dataset	Results on testing without feature selection				Results on testing with feature selection			
	Avg.ROC	Best ROC	Min.ROC	Median ROC	Avg.ROC	Best ROC	Min.ROC	Median ROC
ant-1.7	0.6047	0.7151	0.4470	0.6110	0.5234	0.6154	0.4316	0.5359
camel-1.0	0.6396	0.9621	0.0303	0.7727	0.6074	0.8015	0.2811	0.5902
camel-1.2	0.4262	0.5182	0.2955	0.4295	0.4432	0.5922	0.3246	0.4309
camel-1.4	0.6023	0.7443	0.4275	0.6003	0.5214	0.5793	0.4366	0.5278
camel-1.6	0.5423	0.7422	0.4163	0.5403	0.5045	0.6114	0.3963	0.5010
jedit-3.4	0.6540	0.9623	0.0377	0.7642	0.5180	0.6585	0.4053	0.5232
jedit-4.0	0.8442	0.9655	0.4310	0.9023	0.5494	0.6364	0.4104	0.5675
jedit-4.2	0.6178	0.9310	0.4459	0.5758	0.5185	0.6862	0.2801	0.5046
jedit-4.3	0.6955	0.9742	0.2320	0.6443	0.6838	0.9012	0.2161	0.6994
log4j-1.0	0.6832	0.8918	0.3351	0.6944	0.5354	0.6795	0.3377	0.5234
log4j-1.1	0.7061	0.8070	0.6316	0.7018	0.5222	0.6923	0.3513	0.5281
log4j-1.2	0.4583	0.7611	0.2000	0.4306	0.3614	0.7153	0.1495	0.3450
lucene-2.0	0.5105	0.6721	0.3766	0.5032	0.5531	0.6650	0.4305	0.5664
lucene-2.2	0.4905	0.6718	0.3231	0.4833	0.4630	0.5829	0.3842	0.4571
lucene-2.4	0.4919	0.7231	0.3608	0.4867	0.4851	0.6383	0.3834	0.4739
xalan-2.4	0.5587	0.7481	0.4043	0.5714	0.5364	0.6710	0.4256	0.5171
xalan-2.5	0.5515	0.6294	0.4644	0.5553	0.5098	0.5668	0.4547	0.5082
xalan-2.6	0.6584	0.7322	0.5798	0.6619	0.5064	0.5617	0.4372	0.5076
xalan-2.7	0.4432	0.6920	0.1503	0.4760	0.4201	0.5775	0.2732	0.4087

Table 10
Results of L-RNN classifier with cross-validation.

Dataset	Results on testing without feature selection				Results on testing with feature selection			
	Avg.ROC	Best ROC	Min.ROC	Median ROC	Avg.ROC	Best ROC	Min.ROC	Median ROC
ant-1.7	0.7382	0.7850	0.6888	0.7369	0.8842	0.9048	0.8724	0.8794
camel-1.0	0.8772	0.8921	0.8518	0.8869	0.9052	0.9235	0.8853	0.9042
camel-1.2	0.6136	0.6239	0.5908	0.6174	0.6484	0.6544	0.6414	0.6510
camel-1.4	0.7877	0.7913	0.7807	0.7895	0.7807	0.8575	0.7069	0.7747
camel-1.6	0.6350	0.6726	0.6123	0.6303	0.6914	0.7391	0.6556	0.6934
jedit-3.4	0.8912	0.9087	0.8777	0.8872	0.9177	0.9287	0.9081	0.9189
jedit-4.0	0.8301	0.8328	0.8247	0.8317	0.8700	0.8953	0.8307	0.8783
jedit-4.2	0.8555	0.8978	0.8326	0.8418	0.8913	0.9249	0.8647	0.8886
jedit-4.3	0.9091	0.9333	0.8805	0.9061	0.9131	0.9372	0.8939	0.9131
log4j-1.0	0.8844	0.8976	0.8620	0.8936	0.8759	0.9105	0.8531	0.8712
log4j-1.1	0.8708	0.8754	0.8660	0.8720	0.9145	0.9241	0.9067	0.9127
log4j-1.2	0.8411	0.8500	0.8322	0.8431	0.8522	0.8957	0.8222	0.8449
lucene-2.0	0.8219	0.8407	0.8047	0.8261	0.8677	0.8881	0.8251	0.8741
lucene-2.2	0.7632	0.7819	0.7445	0.7624	0.8236	0.8542	0.8022	0.8270
lucene-2.4	0.7892	0.8187	0.7488	0.7942	0.8259	0.8547	0.8010	0.8185
xalan-2.4	0.8121	0.8356	0.7981	0.8064	0.8270	0.8419	0.8101	0.8277
xalan-2.5	0.6414	0.6837	0.6047	0.6460	0.7493	0.7743	0.7151	0.7537
xalan-2.6	0.6406	0.6779	0.6153	0.6259	0.7125	0.7318	0.7044	0.7055
xalan-2.7	0.8009	0.8335	0.7725	0.7959	0.9295	0.9461	0.9177	0.9241

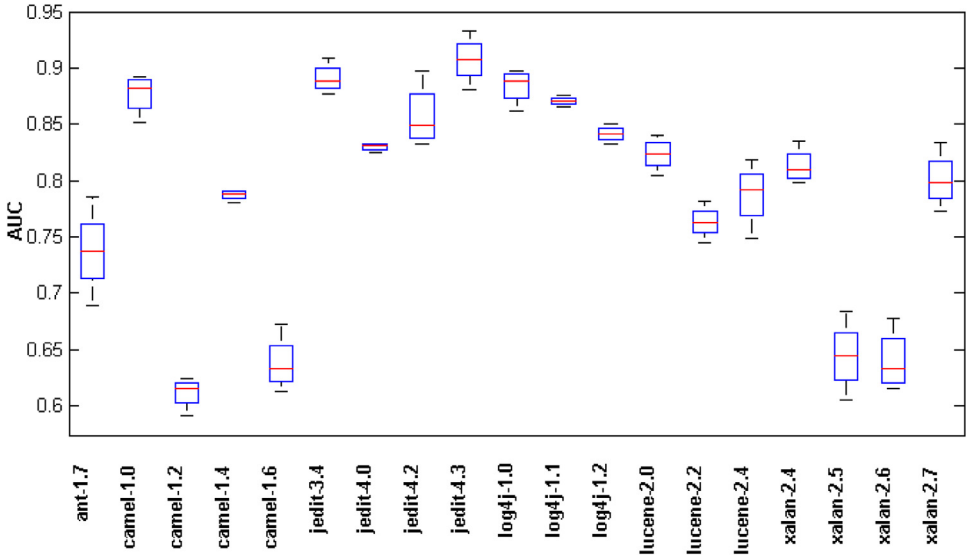

Fig. 13. Boxplots diagram for cross-validation without feature selection.

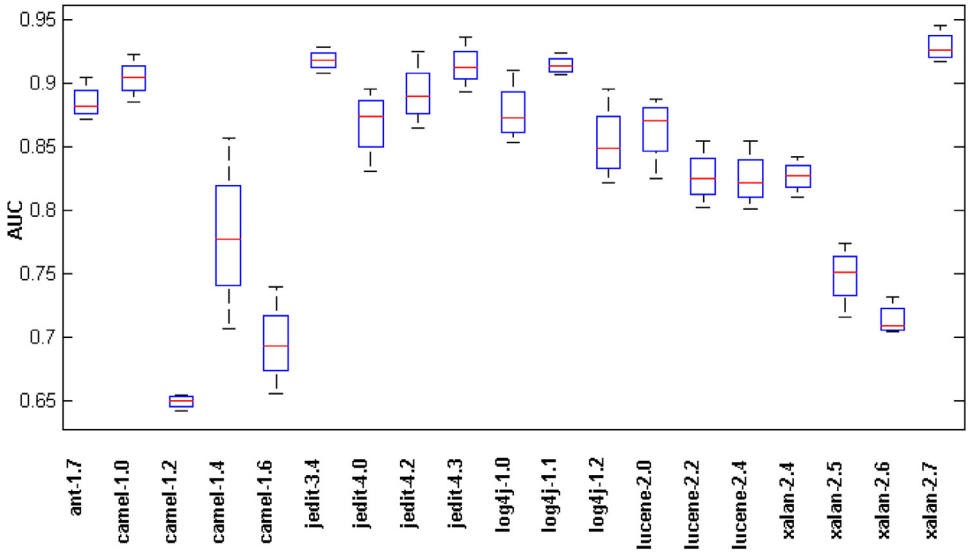

Fig. 14. Boxplots diagram for cross-validation with feature selection.

Table 11
P-value results based on Wilcoxon test.

Compared approaches	*p*_value
Without cross-validation with FS vs. Without cross-validation without FS	9.81E-05
Cross-validation with FS vs. Cross-validation without FS	1.52E-04

Table 12
Comparison between the proposed method and the sate-of-the art methods based on the average AUC values.

Dataset	Our results			(Erturk and Sezer, 2016)		(Shatnawi, 2017)				(Okutan and Yildiz, 2012)
	With cross-validation	Without cross-validation	NB*	ANN	ANFIS	LR	NB	5NN	C4.5	Bayesian Networks
ant-1.7	**0.8842**	0.5234	0.7261	0.8468	0.8184	0.83	0.79	0.76	0.74	0.82
camel-1.0	0.9052	0.6074	0.8881	**0.9242**	0.8939	–	–	–	–	–
camel-1.2	**0.6484**	0.4432	0.5531	0.6008	0.6009	0.57	0.56	0.64	0.52	–
camel-1.4	0.7807	0.5214	0.6603	0.7911	**0.8132**	0.7	0.67	0.67	0.6	–
camel-1.6	0.6914	0.5045	0.6878	0.6807	0.7143	0.65	0.59	0.66	0.54	
jedit-3.4	**0.9177**	0.5180	0.6777	0.8796	0.8997	–	–	–	–	–
jedit-4.0	**0.8700**	0.5494	0.7055	0.8246	0.7826	0.77	0.7	0.81	0.72	–
jedit-4.2	0.8913	0.5185	0.8352	0.8750	**0.9755**	0.84	0.75	0.77	0.64	–
jedit-4.3	**0.9131**	0.6838	0.8273	0.4613	0.9115	–	–	–	–	0.658
log4j-1.0	0.8759	0.5354	0.8455	**0.8929**	0.8857	–	–	–	–	–
log4j-1.1	**0.9145**	0.5222	0.8970	0.9018	0.9018	–	–	–	–	–
log4j-1.2	**0.8522**	0.3614	0.8058	0.7804	0.7719	–	–	–	–	–
lucene-2.0	**0.8677**	0.5531	0.8136	0.8492	0.8651	0.77	0.75	0.7	0.67	–
lucene-2.2	**0.8236**	0.4630	0.7174	0.7628	0.7457	0.62	0.61	0.7	0.58	–
lucene-2.4	**0.8259**	0.4851	0.7723	0.8248	0.8148	0.75	0.69	0.73	0.68	0.633
xalan-2.4	**0.8270**	0.5364	0.7621	0.8186	0.8197	–	–	–	–	–
xalan-2.5	**0.7493**	0.5098	0.6575	0.6747	0.6633	–	–	–	–	0.624
xalan-2.6	**0.7125**	0.5064	0.6400	0.6821	0.6782	–	–	–	–	–
xalan-2.7	**0.9295**	0.4201	0.8054	0.8167	0.8589	–	–	–	–	–
Average	**0.8358**	0.5138	0.7515	0.7836	0.8113	–	–	–	–	–

search space for training process. Moreover, cross validation is better approximation approach either with or without feature selection. As a result, determining the most effective features is an important issue to gain an acceptable classifier and it is not accurate to select a fixed set of features or take all features.

Table 12 presents the results of the proposed algorithm compared with several methods in the literature that used feature selection algorithms or fixed size of features based on AUC values. In this paper, Naive Bayes (NB*) algorithm was applied with cross-validation, where $k = 5$, to explore Naive Bayes performance. Only 3 features were selected (i.e., cbo, wmc and rfc) to make fair compression with (Erturk & Sezer, 2016). It is clear that NB algorithm with three features is not performing well. Moreover, Erturk and Sezer (2016) used only three features based on the suggestion coming from Radjenovic, Hericko, Torkar, and Zivkovic (2013). Shatnawi (2017) used all features in his work, while Okutan and Yıldız (2014) applied Bayesian Network to find the relation between features and select the most effective features.

By investigating the results in Table 12, it can be obviously seen that the results in Shatnawi (2017) and Okutan and Yıldız (2014) were outperformed by the proposed approach. Moreover, the proposed approach outperformed the results in Erturk and Sezer (2016) in 14 datasets out of 19 datasets. Its worth mentioning that (Erturk & Sezer, 2016) used a fixed number of features that were recommended in a previous study. However, in the proposed approach, the features were selected automatically based on a set of well-known metaheuristics algorithms. Thus, the obtained results can be interpreted due to the fact that each project is quite different from another one in the real world. Observing the average results of all approaches, it can be seen that the results of iterated feature selection algorithms with L-RNN classifier with cross-validation outperforms all other approaches based on AUC values. Based on the previous results, it can be concluded that the proposed approach could select the most informative features (i.e., object-oriented metrics) that yield to the highest AUC results.

Table 13
Number of selections by feature selection algorithms for each metric.

Sequence #	Metrics	Number of selections
1	wmc	84
2	dit	85
3	noc	67
4	cbo	96
5	rfc	99
6	lcom	73
7	ca	98
8	ce	87
9	npm	83
10	lcom3	81
11	loc	94
12	dam	88
13	moa	84
14	mfa	79
15	cam	79
16	ic	78
17	cbm	81
18	amc	89
19	max_cc	72
20	avg_cc	72

Table 13 shows the number of times that each object-oriented metric has been selected by feature selection algorithms. The numbers presented in this table are obtained after 11 runs over each dataset. The total number of runs is 209. For example; feature number 4 (cbo) has been selected 96 times over all datasets, which means that cbo metric has a high impact on the prediction process. As a result, to generate a good classification model for software faults prediction cbo metric should not be ignored during collect-

Table 14
Execution time for all datasets in seconds.

Dataset	Without feature selection		With feature selection	
	Without cross validation	With cross validation	Without cross validation	With cross validation
ant-1.7	301.34	1536.52	6325.82	25570.23
camel-1.0	150.82	752.29	5842.41	20174.82
camel-1.2	285.14	1184.13	6021.88	23865.74
camel-1.4	360.58	1925.16	6777.23	27012.37
camel-1.6	125.39	539.87	7529.53	40021.51
jedit-3.4	140.21	675.00	4827.20	17520.30
jedit-4.0	144.21	756.37	5297.61	22281.28
jedit-4.2	158.22	788.21	4961.82	22652.37
jedit-4.3	96.22	491.48	5371.85	20086.44
log4j-1.0	85.61	415.38	3985.25	16650.00
log4j-1.1	115.82	677.57	3527.83	15277.53
log4j-1.2	132.45	705.84	4521.36	26336.42
lucene-2.0	130.57	791.40	4991.75	25362.85
lucene-2.2	169.32	1058.32	4652.88	19216.32
lucene-2.4	172.38	1038.25	5482.11	27157.77
xalan-2.4	378.51	1603.69	7125.33	32514.29
xalan-2.5	420.76	2752.50	7432.96	40275.81
xalan-2.6	432.72	2652.89	7952.60	37790.20
xalan-2.7	475.39	3484.05	7932.22	41849.42
Average	225.03	1254.15	5818.93	26400.82

Table 15
Metrics used for best AUC results for each dataset.

Dataset	Features													Algorithm	AUC
ant-1.7	3	5	6	8	11	13	14	17	19					PSO	0.904833
camel-1.0	1	5	7	12	14	15	16	19						ACO	1
camel-1.2	1	3	4	5	8	9	10	11	12	14				GA	0.6919
camel-1.4	1	2	3	5	6	8	10	11	12	17				GA	0.857479
camel-1.6	1	2	4	7	12	13	17	20						PSO	0.739072
jedit-3.4	1	5	6	8	12	13	17	18						PSO	0.928652
jedit-4.0	1	2	5	6	7	8	12	19	20					GA	0.895276
jedit-4.2	1	2	3	4	5	6	9	12	14	18	19	20		GA	0.9953
jedit-4.3	1	5	6	10	12	13	16	17						ACO	0.9841
log4j-1.0	1	2	7	10	11	13	15	16	17	18	20			GA	1
log4j-1.1	1	2	3	7	10	11	15							ACO	0.924139
log4j-1.2	2	5	7	12	13	14	19	20						GA	0.895727
lucene-2.0	2	3	6	12	14	15	16	17	19					PSO	0.854152
lucene-2.2	1	4	5	7	18									PSO	0.854152
lucene-2.4	3	4	5	8	10	11	20							ACO	0.854696
xalan-2.4	1	2	3	4	5	6	7	8	14	15	16	18	20	GA	0.7095
xalan-2.5	1	3	4	5	7	8	13							GA	0.774263
xalan-2.6	3	6	11	12	13	14								ACO	0.786
xalan-2.7	2	5	6	9	11	13	15	18						PSO	0.946099

ing data for any new project. While feature number 3 (noc) has been selected 67 times, which is less important compared with cbo feature. However, it is hard to ignore any feature during software fault prediction since the difference between numbers reported in Table 13 is not too big. As a result, each project has specific features that have a high impact value on the performance of the classifier.

Table 15 shows the metrics (features) that are selected for each dataset and the algorithm which selected these features. It is clear that each dataset has different metrics based on the dataset complexity. For example, the most effective metrics for jedit-4.3 dataset are 1=wmc, 5=rfc, 6=lcom, 10=lcom3, 12=dam, 13=moa, 16=ic and 17=cbm. As a result, it is important to have an algorithm that is able to determine which features that have a high impact value on the classifier performance, which is the main motivation of our paper.

Moreover, our experimental results show that the proposed algorithm is able to improve the performance of the classification rate as compared with other approaches. However, software faults prediction process is performed off-line, where the execution time is increased exponentially with the size of datasets. In our proposed algorithm, two main processes are performed, one for feature selection and L-RNN classifier. As a result, the execution time is high for large datasets such as xalan-2.7, while it is acceptable for small datasets such as log4j-1.1. Table 14 shows the execution time for all datasets in seconds.

Finally, we can summarize that the performance of L-RNN depends on the characteristics of input data. Feature selection works as data compression. Some data is useless and have little or no effects on classification process. As a result, finding the important and relevant metrics will enhance the training process. Three different FS algorithms are implemented to achieve a balance between exploration and exploitation through searching process to avoid getting stuck in local optima. Feature selection plays a vital role in the training process to enhance the classification rate. This enhancement is achieved by reducing the size of search space (input data) and complexity of L-RNN.

9. Conclusion and future work

In this paper, we have proposed an iterated feature selection algorithm with a layered recurrent neural network for solving the software faults prediction problem. The proposed algorithm is able to select the most important software metrics using different fea-

ture selection algorithms. The classification process is carried out by a layered recurrent neural network. The proposed algorithm is able to obtain an excellent classification rate (with an average of 0.8358 over all datasets) based on AUC results, which outperforms existing results found in the literature such as Naïve Bayes (NB), Artificial Neural Network (ANN), logistic regression (LR), the k-nearest neighbors (k-NN) and C4.5. The obtained results support our claim of the importance of feature selection in building a high-quality classifier rather than using a fixed set of features or all features.

For future work, we plan to investigate the performance of different classifiers such as genetic programming to build a computer model that is able to predict faults based on a selected metrics.

Author contributions form

Contribution	Authors
Study conception and design	Turabieh and Mafarja
Analysis and interpretation of data	Turabieh and Mafarja
Programming	Turabieh and Mafarja
Drafting of manuscript	Turabieh, Mafarja and Li
Critical revision	Turabieh, Mafarja and Li

Acknowledgments

We would like to acknowledge the anonymous reviewers for the valuable comments that improved the quality of this paper. Moreover, we would also like to thank the Editors for their generous comments and support during the review process.

References

Alshayeb, M., & Li, W. (2003). An empirical validation of object-oriented metrics in two different iterative software processes. *IEEE Transactions on Software Engineering, 29*(11), 1043–1049.

Cahill, J., Hogan, J. M., & Thomas, R. (2013). Predicting fault-prone software modules with rank sum classification. *Software engineering conference (ASWEC), 2013 22nd australian (211–219)*.

Carrozza, G., Cotroneo, D., Natella, R., Pietrantuono, R., & Russo, S. (2013). Analysis and prediction of mandelbugs in an industrial software system. *Software testing, verification and validation (ICST), 2013 IEEE sixth international conference on (262–271)*.

Catal, C. (2011). Software fault prediction: A literature review and current trends. *Expert Systems with Applications, 38*(4), 4626–4636.

Catal, C., & Diri, B. (2009). A systematic review of software fault prediction studies. *Expert Systems with Applications, 36*(4), 7346–7354.

Chatterjee, S., Nigam, S., & Roy, A. (2017). Software fault prediction using neuro-fuzzy network and evolutionary learning approach. *Neural Computing and Applications, 28*(1), 1221–1231. doi:10.1007/s00521-016-2437-y.

Chen, J., Liu, S., Liu, W., Chen, X., Gu, Q., & Chen, D. (2014). A two-stage data preprocessing approach for software fault prediction. *Software security and reliability, 2014 eighth international conference on (20–29)*.

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

Cotroneo, D., Natella, R., & Pietrantuono, R. (2013). Predicting aging-related bugs using software complexity metrics. *Performance Evaluation, 70*(3), 163–178.

Czibula, G., Marian, Z., & Czibula, I. G. (2014). Software defect prediction using relational association rule mining. *Information Sciences, 264*, 260–278.

D'Ambros, M., Lanza, M., & Robbes, R. (2010). An extensive comparison of bug prediction approaches. *2010 7th IEEE working conference on mining software repositories (MSR 2010) (31–41)*. doi:10.1109/MSR.2010.5463279.

Dash, M., & Liu, H. (1997). Feature selection for classification. *Intelligent Data Analysis, 1*(1–4), 131–156.

Dejaeger, K., Verbraken, T., & Baesens, B. (2013). Toward comprehensible software fault prediction models using Bayesian network classifiers. *IEEE Transactions on Software Engineering, 39*(2), 237–257.

Dorigo, M., & Caro, G. D. (1999). Ant colony optimization: A new meta-heuristic. In *Proceedings of the 1999 congress on evolutionary computation-CEC99 (cat. no. 99TH8406) (2, 1477 vol. 2)*. doi:10.1109/CEC.1999.782657.

El Emam, K., Benlarbi, S., Goel, N., & Rai, S. N. (2001). Comparing case-based reasoning classifiers for predicting high risk software components. *Journal of Systems and Software, 55*(3), 301–320.

Erturk, E., & Sezer, E. A. (2015). A comparison of some soft computing methods for software fault prediction. *Expert Systems with Applications, 42*(4), 1872–1879.

Erturk, E., & Sezer, E. A. (2016). Iterative software fault prediction with a hybrid approach. *Applied Soft Computing, 49*, 1020–1033.

Ezgi, Z., & Selma, A. O. (2016). A hybrid approach of differential evolution and artificial bee colony for feature selection. *Expert Systems with Applications, 62*, 91–103.

Fawcett, T. (2004). ROC graphs: Notes and practical considerations for researchers. Tech. Rep.

Fowler, M., & Highsmith, J. (2001). The agile manifesto. *Software Development, 9*(8), 28–35.

Ghotra, B., McIntosh, S., & Hassan, A. E. (2015). Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the 37th international conference on software engineering – volume 1 (789–800)*. Piscataway, NJ, USA: IEEE Press. http://dl.acm.org/citation.cfm?id=2818754.2818850.

Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research, 13*(5), 533–549.

Guyon, I., & Elisseeff, A. (2003). An introduction to variable and feature selection [journal article]. *Journal of Machine Learning Research, 3*(Mar), 1157–1182.

Hall, T., Beecham, S., Bowes, D., Gray, D., & Counsell, S. (2012). A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering, 38*(6), 1276–1304.

Hoda, R., Salleh, N., Grundy, J., & Tee, H. M. (2017). Systematic literature reviews in agile software development: A tertiary study. *Information and Software Technology, 85*, 60–70.

Holland, J. H. (1992). Adaptation in natural and artificial systems:. *An introductory analysis with applications to biology, control and artificial intelligence*. Cambridge, MA, USA: MIT Press.

Hosmer, D. W., & Lemeshow, S. (2000). Applied logistic regression. *Wiley-interscience publication*. Wiley Series in probability and statistics) (2nd ed).

Huang, C. L., & Wang, C. J. (2006). A GA-based feature selection and parameters optimization for support vector machines. *Expert Systems with Applications, 31*(2), 231–240. doi:10.1016/j.eswa.2005.09.024. http://www.sciencedirect.com/science/article/pii/S0957417405002083.

Jin, C., & Jin, S. W. (2015). Prediction approach of software fault-proneness based on hybrid artificial neural network and quantum particle swarm optimization. *Applied Soft Computing, 35*, 717–725.

Jureczko, M., & Madeyski, L. (2010). Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th international conference on predictive models in software engineering (9:1–9:10)*. New York, NY, USA: ACM. doi:10.1145/1868328.1868342. http://doi.acm.org/10.1145/1868328.1868342.

Kennedy, J., & Eberhart, R. (1995). Particle swarm optimization. *Neural networks, 1995. proceedings., IEEE international conference on.(4, 1942–1948 vol.4)*. doi:10.1109/ICNN.1995.488968.

Kennedy, J., & Eberhart, R. C. (1997). A discrete binary version of the particle swarm algorithm. *Systems, man, and cybernetics, 1997. computational cybernetics and simulation., 1997 IEEE international conference on (5, 4104–4108)*.

Khoshgoftaar, T. M., & Seliya, N. (2003). Software quality classification modeling using the SPRINT decision tree algorithm. *International Journal on Artificial Intelligence Tools, 12*(03), 207–225.

Khoshgoftaar, T. M., Xiao, Y., & Gao, K. (2014). Software quality assessment using a multi-strategy classifier. *Information Sciences, 259*, 555–570.

Koru, A. G., Emam, K. E., Zhang, D., Liu, H., & Mathew, D. (2008). Theory of relative defect proneness. *Empirical Software Engineering, 13*(5), 473. doi:10.1007/s10664-008-9080-x. https://doi.org/10.1007/s10664-008-9080-x.

Kudo, M., & Sklansky, J. (2000). Comparison of algorithms that select features for pattern classifiers. *Pattern Recognition, 33*(1), 25–41.

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

Li, W., & Henry, S. (1993). Object-oriented metrics that predict maintainability. *Journal of systems and software, 23*(2), 111–122.

Lipton, Z. C. (2015). A critical review of recurrent neural networks for sequence learning. CoRR, abs/1506.00019. http://arxiv.org/abs/1506.00019.

Liu, H., & Motoda, H. (2012). *Feature selection for knowledge discovery and data mining (454)*. Springer Science & Business Media.

Maggiori, E., Charpiat, G., Tarabalka, Y., & Alliez, P. (2017). Recurrent neural networks to correct satellite image classification maps. *IEEE Transactions on Geoscience and Remote Sensing, 55*(9), 4962–4971. doi:10.1109/TGRS.2017.2697453.

Malhotra, R. (2014). Comparative analysis of statistical and machine learning methods for predicting faulty modules. *Applied Soft Computing, 21*, 286–297.

Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing, 27*, 504–518.

Maniezzo, A. (1992). Distributed optimization by ant colonies. *Toward a practice of autonomous systems: proceedings of the first European conference on artificial life (134)*.

Maren, A. J., Harston, C. T., & Pap, R. M. (2014). *Handbook of neural computing applications*. Academic press.

M.D.P (2017). NASA. http://nasa-softwaredefectdatasets.wikispaces.com/. Last [Accessed 24 November 2017].

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

Miholca, D. L., Czibula, G., & Czibula, I. G. (2018). A novel approach for software defect prediction through hybridizing gradual relational association rules with artificial neural networks. *Information Sciences, 441*, 152–170.

Mohan, B. C., & Baskaran, R. (2012). A survey: Ant colony optimization based recent research and implementation on several engineering domain. *Expert Systems with Applications, 39*(4), 4618–4627. doi:10.1016/j.eswa.2011.09.076. http://www.sciencedirect.com/science/article/pii/S0957417411013996.

Okutan, A., & Yıldız, O. T. (2014). Software defect prediction using bayesian networks. *Empirical Software Engineering, 19*(1), 154–181. doi:10.1007/s10664-012-9218-8. https://doi.org/10.1007/s10664-012-9218-8.

Oliveira, J. C. M., Pontes, K. V., Sartori, I., & Embiruçu, M. (2017). Fault detection and diagnosis in dynamic systems using weightless neural networks. *Expert Systems with Applications, 84*, 200–219.

Porter, A. A., & Selby, R. W. (1990). Empirically guided software development using metric-based classification trees. *IEEE software, 7*(2), 46–54.

Prugel-Bennett, A. (2010). Benefits of a population: Five mechanisms that advantage population-based algorithms. *IEEE Transactions on Evolutionary Computation, 14*(4), 500–517. doi:10.1109/TEVC.2009.2039139.

Qin, S., Yang, X., Xue, X., & Song, J. (2017). A one-layer recurrent neural network for pseudoconvex optimization problems with equality and inequality constraints. *IEEE Transactions on Cybernetics, 47*(10), 3063–3074. doi:10.1109/TCYB.2016.2567449.

Radjenovic, D., Hericko, M., Torkar, R., & Zivkovic, A. (2013). Software fault prediction metrics: A systematic literature review. *Information and Software Technology, 55*(8), 1397–1418. doi:10.1016/j.infsof.2013.02.009. http://www.sciencedirect.com/science/article/pii/S0950584913000426.

Radjenović, D., Heričko, M., Torkar, R., & Živkovič, A. (2013). Software fault prediction metrics: A systematic literature review. *Information and Software Technology, 55*(8), 1397–1418.

Rathore, S. S., & Kumar, S. (2017a). A decision tree logic based recommendation system to select software fault prediction techniques. *Computing, 99*(3), 255–285.

Rathore, S. S., & Kumar, S. (2017b). Towards an ensemble based system for predicting the number of software faults. *Expert Systems with Applications, 82*, 357–382.

Rathore, S. S., & Kumar, S. (2017c). Linear and non-linear heterogeneous ensemble methods to predict the number of faults in software systems. *Knowledge-Based Systems, 119*, 232–256.

Rauber, T. W., de Assis Boldt, F., & Varejão, F. M. (2015). Heterogeneous feature models and feature selection applied to bearing fault diagnosis. *IEEE Transactions on Industrial Electronics, 62*(1), 637–646.

Royce, W. W. (1987). Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th international conference on software engineering (328–338)*.

Ruiz, L., Rueda, R., Cuéllar, M., & Pegalajar, M. (2018). Energy consumption forecasting based on elman neural networks with evolutive optimization. *Expert Systems with Applications, 92*(Supplement C), 380–389. doi:10.1016/j.eswa.2017.09.059.

Senjyu, T., Takara, H., Uezato, K., & Funabashi, T. (2002). One-hour-ahead load forecasting using neural network. *IEEE Transactions on Power Systems, 17*(1), 113–118.

Shatnawi, R. (2017). The application of ROC analysis in threshold identification, data imbalance and metrics selection for software fault prediction. *Innovations in Systems and Software Engineering, 13*(2), 201–217. doi:10.1007/s11334-017-0295-0. https://doi.org/10.1007/s11334-017-0295-0.

Sheskin, D. J. (2003). *Handbook of parametric and nonparametric statistical procedures*. CRC Press.

Stavru, S. (2014). *A critical examination of recent industrial surveys on agile method usage* (94, pp. 87–97).

Talbi, E. G. (2009). *Metaheuristics: From design to implementation (74)*. John wiley & sons.

Tera-Promise (2017). http://openscience.us/repo. Last accessed 24 November 2017.

Thwin, M. M. T., & Quah, T. S. (2005). Application of neural networks for software quality prediction using object-oriented metrics. *Journal of Systems and Software, 76*(2), 147–156.

Van Laarhoven, P. J., & Aarts, E. H. (1987). Simulated annealing. *Simulated annealing: Theory and applications (7–15)*. Springer

Xing, F., Guo, P., & Lyu, M. R. (2005). A novel method for early software quality prediction based on support vector machine. *Software reliability engineering, 2005. ISSRE 2005. 16th IEEE international symposium on (10–pp)*.

Yuan, X., Khoshgoftaar, T. M., Allen, E. B., & Ganesan, K. (2000). An application of fuzzy clustering to software quality prediction. *Application-specific systems and software engineering technology, 2000. proceedings. 3rd IEEE symposium on (85–90)*.

Zhang, F., Mockus, A., Keivanloo, I., & Zou, Y. (2016). Towards building a universal defect prediction model with rank transformed predictors. *Empirical Software Engineering, 21*(5), 2107–2145. doi:10.1007/s10664-015-9396-2. https://doi.org/10.1007/s10664-015-9396-2.

Zhang, X. Y., Yin, F., Zhang, Y. M., Liu, C. L., & Bengio, Y. (2017). Drawing and recognizing chinese characters with recurrent neural network. *IEEE Transactions on Pattern Analysis and Machine Intelligence, PP*(99), 1. doi:10.1109/TPAMI.2017.2695539.

Zimmermann, T., Nagappan, N., & Zeller, A. (2008). Predicting bugs from history. *Software Evolution, 4*(1), 69–88.