This article can be cited as D. Azar, K. Fayad and C. Daoud, A Combined Ant Colony Optimization and Simulated Annealing Algorithm to Assess Stability and Fault-Proneness of Classes Based on Internal Software Quality Attributes, International Journal of Artificial Intelligence, vol. 14, no. 2, pp. 137-156, 2016.

A Combined Ant Colony Optimization and Simulated Annealing Algorithm to Assess Stability and Fault-Proneness of Classes Based on Internal Software Quality Attributes

Danielle Azar[1], **Karl Fayad**[2] **and Charbel Daoud**[3]

Department of Computer Science and Mathematics
Lebanese American University
Byblos, Lebanon 14010

[1]danielle.azar@lau.edu.lb
[2]karl.fayad@lau.edu
[3]charbel.daoud@lau.edu

ABSTRACT

Several machine learning algorithms have been used to assess external quality attributes of software systems. Given a set of metrics that describe internal software attributes (cohesion, complexity, size, etc.), the purpose is to construct a model that can be used to assess external quality attributes (stability, reliability, maintainability, etc.) based on the internal ones. Most of these algorithms result in assessment models that are hard to generalize. As a result, they show a degradation in their assessment performance when used to estimate quality of new software modules. This paper presents a hybrid heuristic to construct software quality estimation models that can be used to predict software quality attributes of new unseen systems prior to re-using them or purchasing them. The technique relies on two heuristics: simulated annealing and ant colony optimization. It learns from the data available in a particular domain guidelines and rules to achieve a particular external software quality. These guidelines are presented as rule-based logical models. We validate our technique on two software quality attributes namely stability and fault-proneness - a sub-attribute of maintainability. We compare our technique to two state-of-the-art algorithms: Neural Networks (NN) and C4.5 as well as to a previously published Ant Colony Optimization algorithm. Results show that our hybrid technique out-performs both C4.5 and ACO in most of the cases. Compared to NN, our algorithm preserves the white-box nature of the predictive models hence, giving not only the classification of a particular module but also guidelines for software engineers to follow in order to reach a particular external quality attribute. Our algorithm gives promising results and is generic enough to apply to any software quality attribute.

Keywords: Prediction, C4.5, rule sets, software quality, metric, search-based software engineering, ant colony optimization, simulated annealing.

ACM Computing Classification System: I.2.1

1 Introduction and related work

Software is expensive. Software quality is very important. Software quality attributes are divided into two categories: 1. Internal attributes such as coupling, cohesion, size, etc. and 2. External quality attributes such as maintainability, stability, re-usability, etc. (Shepperd, 1993). The former class of attributes can be directly measured by looking at source code and software architecture and/or specifications. The latter ones cannot be directly measured. However, the former category can be used as a good indicator of the latter one (Briand, Devanbu and Melo, 1997), (Dallal, 2012) and (Dallal and Morasca, 2014). In fact, practitioners are interested in assessing the external attributes based on the internal ones. Many object-oriented metrics that capture internal software quality attributes have been proposed and extensively used over the past decade (Hendersen-Sellers, 1991), (Coppick and Cheatham, 1992), (Barnes and Swim, 1993), (Chidamber and Kemerer, 1994), (Li and Henry, 1993b), (Li and Henry, 1993a), (Lorenz and Kidd, 1994), (Henderson-Sellers, 1996) and (Briand et al., 1997). Prediction models that establish relationships between internal quality attributes and external quality attributes using such metrics have been widely used in order to assess different external quality attributes of software systems. A variety of statistical techniques are used for this purpose where statistical relationships are established between measures of interal quality attributes and measures of external ones (Abreu and Melo, 1996) and (Khoshgoftaar, Allen, Halstead, Trio and Flass, 1998) and (Subramanyam and Krishnan, 2003). However, such measures are often complex and non-linear (Thwin and Quah, 2005). Regression analysis techniques have also been used to construct models with high accuracy (Briand, W$\ddot{u}$st, Lounis and Ikonomovski, 1999), (Briand, Wust, Daly and Porter, 2000). Machine learning techniques, are well suited for learning such relationships. Examples of such techniques are Neural Networks (Thwin and Quah, 2005), Support Vector Machines, decision tree models and regression tree models (Zhou and Leung, 2007). Our work consists of building predictive models in the form of rule sets where rules are conjunctions of tests on internal quality attributes and classification labels encode the measure of the external quality attribute to predict. Our interest in the models that take the form of rules rises from the fact that these are easy to interpret by practitioners. Moreover, not only do they assess the external quality attribute but they also provide guidelines for practitioners to achieve it. Such rule-based models are tightly related to decision tree models. As a matter of fact, they can be derived from them. Decision tree models have been used as early as 1988 (Selby and Porter, 1990) to predict different software quality attributes such as re-usability (Mao, Sahraoui and Lounis, 1998) of an object-oriented class, or indicators of quality attributes such as cost of rework (Basili, Condon, Emam, Hendrick and Melo, 1997), the average isolation effort (Almeida, Lounis and Melo, 1999), fault-proneness - a software maintainability sub-characteristic (Briand et al., 1997) and (Arisholm, Briand and Fulglerud, 2007), (Arisholm and Briand, 2006), (Lessmann, Baesens, Mues and Pietsch, 2008), etc. In (Lessmann et al., 2008), similar to us, the authors argue that the comprehensibility of the resulting classification models is very important as it illustrates the classification procedure thus giving guidelines on how to improve a certain software quality characteristic (fault-proneness in their case). Arisholm et al. (Arisholm, Briand and Johannessen, 2010) compare different modeling techniques and the impact of selecting different types of measures

as predictors. They find that the measures and techniques used are highly dependent on the evaluation criteria used to assess the prediction models. In our work, we propose an algorithm to build rule-based prediction models. In our algorithm, we combine two meta-heuristics namely, Ant Colony Optimization (ACO)- a swarm intelligence nature-inspired algorithm- and Simulated Annealing(SA). Previous work has addressed problems in software engineering using meta-heuristics. Typically, meta-heuristics such as simulated annealing, genetic algorithms and genetic programming are used as sampling techniques (Harman, 2007). For example, Pedrycz et al. (Pedrycz and Succi, 2005) represent classifiers as hypberboxes and use genetic algorithms to evolve existing models into new ones. Vivanco (Vivanco, 2007) uses a genetic algorithm to improve a classifier accuracy in identifying problematic components. The approach relies on the selection of metrics that are more likely to improve the performance of predictive models. In (Azar and Precup, 2007) and (Azar, 2010), the authors use genetic algorithms to adapt classification models built from one domain data to unseen data in a different domain. Unlike the current work, the classification models used are initially constructed by a machine learning technique and then adapted to new data sets. Azar et al. (Azar, Harmanani and Korkmaz, 2009) and (Azar and Harmanani, 2011) present a hybrid approach that combines different meta-heuristics such as genetic algorithms, tabu search and simulated annealing to re-combine existing rule sets into new (more accurate) ones. In these also, the initial rule sets are built from one data set and adapted to a different one. In (Azar and Harmanani, 2011), data from different domains including software quality is used and the approach is promising in all different domains. Grosser et al. (Grosser, Sahraoui and Valtchev, 2002) propose a case-based reasoning approach for classifying object-oriented classes as stable or not. In (Azar and Vybihal, 2011), the authors present an ant colony optimization algorithm which adapts a single prediction model to a new set of data. The results were also promising. In this current work, we are inspired from (Azar and Vybihal, 2011) and extend the work to different ant colonies working in parallel on different models. Unlike (Azar and Vybihal, 2011) which adapts already existing models to new data sets, this work proposes an approach that constructs the model from scratch and by learning from different models. The underlying algorithm is a combination of Ant Colony Optimization and Simulated Annealing. Ant Colony Optimization is a population-based meta-heuristic. Such types of heuristics have shown to be successful in several application domains. In (Yazdani, Sadeghi-Ivrigh, Yazdani, Sepas-Moghaddam and Meybodi, 2015), the authors present an algorithm inspired from the behavior of fish to perform global optimization in continuous and stationary environments. In (David, Precup, Petriu, Rădac and Preitl, 2013), the authors propose a Gravitational Search Algorithm (GSA) - a nature-inspired optimization algorithm based on Newtons law of gravity and laws of motion - to design fuzzy control systems with a reduced parametric sensitivity. Previous work in which a swarm intelligence technique has been combined with another meta-heuristic includes (Valdez, Melin and Castillo, 2011) in which the authors present an evolutionary method which combines Particle Swarm Intelligence (PSO) with genetic algorithms for mathematical function optimization and uses fuzzy logic to combine the results. The hybrid approach showed better results than individual methods when tested on a benchmark of mathematical functions. In (Zăvoianu, Bramerdorfer, Lughofer, Silber, Amrhein and Klement, 2013), the authors apply non-dominated sorting genetic algorithm

to obtain high quality Pareto-optimal solutions for three different scenarios. The approach was hybrid as it used artificial neural networks to create the objective functions. To the best of our knowldge, our work is the first to combine Ant Colony Optimization and Simulated annealing to construct software quality prediction models. The closest work to ours which combined the two heuristics is (Rizauddin, 2013) where an algorithm that uses ACO is used to construct classification rules. Like ours, the algorithm in (Rizauddin, 2013), extracts the rules directly from the data set. Unlike ours, each ant in the proposed ACO extracts a rule by adding conditions one by one whereas simulated annealing is used to guide the choice of the condition. The algorithm was benchmarked on seventeen data sets from the UCI repository none of which is from the software quality domain. Results from our work show that our new proposed algorithm out-performs rule sets produced by the state-of-the art machine learning algorithms Neural Networks and C4.5. We also compare it to the algorithm in (Azar and Vybihal, 2011). The remainder of this paper is organized as follows. In Section 2, we introduce the problem. In Section 3, we give an overview of ACO and Simulated Annealing and we present our hybrid algorithm. In Section 5, we explain the set up of the experiments we conducted and we discuss the results. In Section 6, we summarize the work and highlight some interesting venues to explore in the future.

2 Problem Formulation

In this work, the unit that we consider is a class in an object-oriented system. We use binary classification of such classes. A class either possesses a certain software quality attribute or it does not. We concentrate on two software quality attributes namely, stability and fault-proneness. However, the same approach can be used on other quality attributes. A class is considered *unstable* if at least one of the following cases holds:

- A method ceases to exist in a future version of the class.
- A method is created in a future version of the class (it was not in the previous version).
- The header of a method changes between two different versions of the class.

If none of the above cases holds, the class is said to be *stable*[1].
We define a class to be *fault-prone* if it contains one or more errors. Most faults in a software system are found in a few of the system's components (Moller and Paulish, 1993), (Kaaniche and Kanoun, 1996) and (Fenton and Neil, 1999). Fault-proneness of a class has a direct impact on the maintainability and hence, fault-proneness is seen as a sub-characteristic of maintainability.
The classification models are rule-based models derived from decision trees. Each model is a set of rules and a default classification label. A rule is a conjunction of tests on internal quality attributes and a classification label (measuring the external attribute). One example of a rule is: *if NOM* ≤ 20 *and NOC* ≤ 5 *and LOC* ≤ 1000 *then class label* $= 1$ which classifies a class with number of methods less than or equal to 20 and number of children less than or equal

[1]We consider syntactic stability of the class only and we ignore the body and the semantic.

Rule1:	$NOM \leq 20 \wedge NOC \leq 5 \wedge LOC \leq 1000 \rightarrow 0$
Rule2:	$NOM \leq 20 \wedge NOC > 5 \rightarrow 0$
Rule3:	$DIT \leq 5 \wedge NOC \leq 5 \rightarrow 1$
Default class:	1

Figure 1: Rule set composed of three rules and a default classification label

to 5 and number of lines of code less than or equal to 1000 as stable (stable=1, unstable=0). Because the rule set may not cover all instances in a particular data set, it is augmented with a default classification label which applies to all cases that are not covered by any of the rules. Figure 2 is an example of a rule set composed of three rules and a default classification label.

2.1 Performance Measures

We evaluate the performance of a classification system S using the traditional measure of accuracy shown in Equation 2.1. This is extracted from the contingency matrix shown in Figure 2 where n_{ij} is the number of cases in the data set which have classification label i but have been classified with label j. In all our experiments, we assume equal penalty for misclassified cases.

$$Accuracy(S) = \frac{\sum_{i=1}^{k} n_{ii}}{\sum_{i=1}^{k} \sum_{j=1}^{k} n_{ij}} \tag{2.1}$$

		Predicted Label			
		c_1	c_2	$\ldots$	c_k
real label	c_1	n_{11}	n_{12}	$\ldots$	n_{1k}
	c_2	n_{21}	n_{22}	$\ldots$	n_{2k}
	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
	c_k	n_{k1}	n_{k2}	$\ldots$	n_{kk}

Figure 2: Contingency matrix. n_{ij} is the number of cases with real label c_i and predicted label c_j.

3 Overview of ACO and Simulated Annealing

In this section, we give an overview of Ant Colony Optimization and Simulated annealing and then we explain how we combined them in one hybrid heuristic.

3.1 ACO

Ant Colony Optimization (ACO) has been sucessfully used for solving hard combinatorial problems. It is a meta-heuristic technique originally introduced by Dorigo (Dorigo and Caro, 1999) and (Dorigo and Stutzle, 2004) and inspired from the behavior of ants in search for food. Ants

communicate through the environment by depositing pheromone on the route they follow in their search for food. The more a path is explored, the more pheromone it receives which attracts other ants. The two main phases of the ACO algorithm are the construction of the solution and the update of the deposited pheromone. Several ants explore the search space in parallel. The choice that the ant makes between several moves is based on the intensity of the pheromone found on the trail and a heuristic function which encodes information about the desirability of the move. At each iteration, pheromone evaporates to push ants to explore new search sub-spaces and thus avoid the convergence of all ants to the same search neighborhood. Figure 3 shows the pseudocode of ACO.

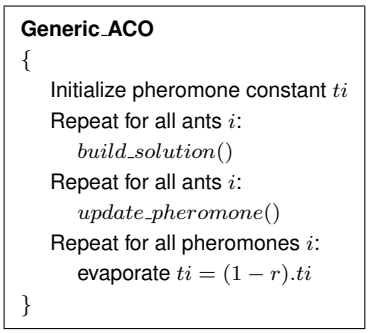

Figure 3: Generic ACO Pseudocode

3.2 Simulated Annealing

Simulated Annealing(SA) is another meta-heuristic technique first introduced in (Metropolis, Rosenbluth, Rosenbluth, Teller and Teller, 1953). It simulates the process of making strong glass whereby the element is heated quickly to very high temperatures and then cooled down slowly. When the glass is heated to a very high temperature, it becomes liquid and the atoms move relatively freely. Subsequently, the temperature of the glass is lowered slowly and, at each temperature, the atoms can move just enough to begin adopting the most stable orientation. This slow cooling is known as *annealing* and results in crystallization. In simulated annealing, the goal is to optimize a solution. At each iteration of the algorithm, a solution is changed and compared to the previous one. Based on an objective function, the algorithm either accepts the new solution or rejects it. In early iterations, changes happen more frequently then they start reducing as further iterations proceed. SA allows the solution to degrade at times in order to escape local optima. Figure 4 shows the pseudocode of SA.

4 Proposed Annealing-ACO

The algorithm has two main stages: the exploration stage and the intensification stage.
Exploration Stage: The algorithm starts by constructing rule sets from scratch at random. For this, metrics are picked randomly from the list of available ones and combined with values and operators which are also picked at random. Such rule sets are represented as matrices in the heuristic. Given rule set R of r rules with c being the maximum number of conditions in a rule,

Simulated_Annealing

{

Given: An initial solution S_0, an initial temperature T_0, a fraction K, $0 \leq K \leq 1$

Set initial set of solutions $X = \{S_0\}$

Repeat for a certain number of iterations:

Set $T = T_0$;

Repeat for each temperature T_i:

Repeat for n iterations:

Select randomly a solution S_i from the set X

S'_i=Perturb (S_i)

Calculate :

ΔE = evaluation (S'_i) - evaluation (S_i)

if ($\Delta E > 0$)

accept S'_i

else:

generate random number r, $0 \leq r \leq 1$

if ($r < e^{\frac{-\Delta E}{T}}$)

accept S'_i

else

reject S'_i

$T_{i+1} = K * T_i$

Return best solution in X

}

Figure 4: Simulated Annealing Pseudocode

NOM <= 20	DIT < 3	NOC <= 4	1
NOC > 5			0
			1

Figure 5: Matrix representing a rule set which consists of two rules and a default classification label.

R is represented as a $N \times M$ array where $N = 1 + r$ and $M = 1 + c$. Each row in the array stores one rule in R (one condition in each cell). The last column stores the classification label of a rule. The last row in the array is left empty except for the last column which stores the default classification label of the rule set. Figure 5 illustrates this representation. Traditionally in ACO, ants march on a graph where nodes encode sub-solutions and edges operations that help in the construction of the solution. In our ACO, ants march on the constructed two-dimensional matrix that describes a rule set. Ants perform what we call an $S - march$ i.e. they move from one cell on the matrix vertically to the adjacent one until they hit a border at which point they move to the node horizontally adjacent, etc.[2] An ant starts at a random corner in the matrix and walks in a vertical march. At each cell, an ant applies simulated annealing to change the underlying condition. The change consists of modifying with a certain probability the metric, the operator or the value of the condition. In the latter case, the value is changed to one picked randomly from the set of cutpoints of the attribute[3]. During the early iterations

[2]Ants might divert from their S-march according to pheromone. This will be described later on.

[3]To compute the cutpoint of an attribute, the data is sorted by the attribute and the average of two values where

of SA, modifications are accepted with a high probability even if they result in a deteriorated objective function (f) value. This probability decreases gradually at subsequent iterations. Such a scheme allows the ants to explore different regions of the search space at early stages prior to converging to promising ones. During its search process, an ant releases pheromone with intensity proportional to the quality of the underlying metric. The quality of a metric is computed based on Equation 4.1. The objective function includes the measurement we aim to optimize (accuracy) as well as the Youden's J_index which measures the average accuracy per class label (Equation 4.2). Both measures (accurach and J_index) are computed based on the training data. The reason we include the latter measure is to avoid the algorithm from converging to the majority classifier which has a very high accuracy on imbalanced data sets but low J_index. A metric is considered good if it results in a higher f value. If the modification is accepted by the underlying simulated annealing process, the ant deposits pheromone on the underlying cell. The metrics with the highest pheromone values are chosen with a higher probability. At each iteration, pheromone evaporates.

$$f = \alpha.accuracy + \beta.J_{index} \tag{4.1}$$

$$J_{index} = \frac{1}{k}\sum_{i=1}^{k}\frac{n_{ii}}{\sum_{j=1}^{k} n_{ij}} \tag{4.2}$$

Intensification Stage: At this stage, the algorithm seeks further specific improvements by modifying the operators and values of conditions only. The modifications happen through Simulated Annealing. The same objective function (f) is used. Algorithm 6 shows the pseudocode of our instantiated SA-ACO. It is possible that redundant conditions appear within rules as ants are constructing solutions. In this case, one of the two conditions is removed keeping the more general one[4]. In case of redundancy in rules, only one is kept.

5 Experimental Results and Discussion

5.1 Data Sets

We have constructed our own data sets by extracting metrics from publicly available software systems[5] and we also used other ready-made data sets that are available in public repositories. In total, we used six data sets, four of which describe stability of classes in an object-oriented framework and the other two describe fault-proneness as a sub-attribute of class maintainability. Table 1 summarizes the data sets and tables 2, 3, 4 and 5 , summarize the metrics describing the data sets[6]. Detailed description of these metrics can be found in (Barnes and Swim, 1993), (Chidamber and Kemerer, 1994), (Henderson-Sellers, 1996), (Coppick and Cheatham, 1992).

the classification flips is recorded as a cutpoint

[4] $NOM > 5$ and $NOM > 9$ are two redundant conditions. $NOM > 5$ is removed.

[5] Data sets can be obtained by emailing the primary author at danielle.azar@lau.edu.lb.

[6] CM1 and JM1 can be found online at *promise.site.uottawa.ca/SERepository/datasets-page.html* while the remaining data sets were built in-house.

```
Instantiated ACO
{
    Initialize number of iterations constant total_itr
    Initialize pheromone constant p
    Initialize temperature constant t
    Initialize cooling rate constant c
    Initialize evaporation rate constant e
    Initialize ant seek radius constant s
    Initialize pheromone intensity p_i
    Repeat until done:
        if(current_itr < total_itr/2)
          stage = 1
        else if(current_itr < 3 * total_itr/4)
          stage = 2
        if(stage = 1)
          s = 0
        else if(stage = 2)
        s = x
        Repeat for all ants j:
            build_solution(s, stage)
            if (stage ≠ 1 && improvement)
                p = p + p_i//release pheromone
        p = p * (1 − e)//evaporate pheromone
        t = t * (1 − c) //update temperature

}
```

```
build_solution(seek_radius, stage)
{
    if(stage = 1)
      r = random()
      if (r < 0.1)
        add_rule();
      if(r < 0.3)
        remove_rule();
      if(r < 0.55)
        add_condition();
      if(r < 0.8)
        remove_condition();
      else
        revive_best_ruleset();
    (S', attribPerturb) = perturb_cell(matrix, row, col)
    f' = evaluate_objective_function(S')
    r = random()
    δ = f' − f
    if(δ > 0)
      acceptance = 1.0
    else
      acceptance = e^(−δ/t)
    if(r < acceptance)
      R = R'
      f = f'
    if(attribPerturb)
      update_pheromone(attribute) // release pheromone on attribute at [row][col]
    if(f' > fBest)
      R* = R'
      fBest = f'
    march(seek_radius)
}
```

```
perturb_cell(Matrix m,int row,int col)
{
    attribPerturbation = false
    if(cell_encodes_class(m,row,col))
      modify_class(m,row,col)
    else // cell encodes condition
      r random()
      if(stage == 1&&r < 0.1)
        modify_attribute(m,row,col)
        a = true
      else if(r < 0.5)
            modify_operator(m,row,col)
      else
            modify_value(m,row,col)
    return (m,attribPerturb)

}
```

```
march(seek_radius)
{
    found=getLargestPheromone(seek_radius);
    if(found)
      go to cell
    else
      smarch() //go to adjacent cell

}
```

Figure 6: Pseudocode of instantiated SA-ACO.

Table 1: Data sets

Dataset	Size	Metrics	Classes (%1-%0)	Software Quality Attribute
Geotools	1027	12	991-36 (96.5%-3.5%)	stability
Weka	2851	12	1878-973 (66%-34%)	stability
STAB1	2920	19	2481-439 (85%-15%)	stability
STAB2	690	22	455-235 (66%-34%)	stability
JM1	7253	21	2013-5240 (28%-72%)	fault-proneness
CM1	332	21	21-311 (6%-94%)	fault-proneness

5.2 Experimental Setup and Results

We performed our experiments on an iCore 7 CPU with 8 GB memory running Windows 7 Pro 64-bit. Table 7 summarizes the time each experiment took (30 runs).

To validate our approach, we use 10-fold cross-validation whereby each data set is divided into ten folds of roughly equal size. Each run consists of 10 iterations. At each iteration, the union of 9 folds is taken as the training set to compute the objective function as well as the quality of the metrics and the remaining fold is kept as a testing set. This is repeated 10 times, each time a different fold is kept as a testing set. At the end of each run, we average over the 10 iterations. Since the algorithm involves several parameters, we ran it with different parameter values. We report the results obtained with the best ones(Table 6). In particular, the number of iterations was chosen after we saw that there was no improvement beyond this chosen point or that the improvement was very minimal compared to the run time.

Because SA-ACO includes a stochastic element, we ran each experiment 30 times and we averaged over the thirty runs. To validate our technique, we compare it to Neural Networks (NN), C4.5, and the ACO published in (Azar and Vybihal, 2011). In Table 8, we show the results we obtained on each data set and in Table 9, we summarize the performance of all algorithms over all the data sets.

Looking at the data sets individually (Table 8), we can see that on Geotools, SA-ACO reaches an accuracy above 96% on both the testing and the training data - results very close to those obtained with NN. SA-ACO out-performs C4.5 by about 18% on both the training and the testing data and ACO by 12%. On Weka, SA-ACO reaches a testing accuracy close to that of NN and out-performs C4.5 and ACO by around 14% and 9% respectively on the training set and 12% and 7% on the testing set. On STAB1, SA-ACO outperforms NN by 12% on the testing data and C4.5 and ACO by 13% and 5% respectively on the training data and 12% and

Table 2: STAB1 Metrics

Name	Description
LCOM	lack of cohesion methods
COH	cohesion
COM	cohesion metric
COMI	cohesion metric inverse
OCMAIC	other class method attribute import coupling
OCMAEC	other class method attribute export coupling
CUB	number of classes used by a class
NOC	number of children
NOP	number of parents
DIT	depth of inheritance
MDS	message domain size
CHM	class hierarchy metric
NOM	number of methods
WMC	weighted methods per class
WMCLOC	LOC weighted methods per class
MCC	McCabe's complexity weighted method per class
DEPCC	operation access metric
NPPM	number of public and protected method in a class
Stress	amount of work on the class

4% respectively on the testing data. On STAB2, SA-ACO reaches an accuracy around 4% less than that of NN but out-performs C4.5 and ACO by around 12% and 9% respectively on the training data and 8% and 5% on the testing data. On CM1, our algorithm reaches results as good as those of NN and out-performs C4.5 by more than 26% on the training data and by 10% ACO. On the testing set, it outperforms C4.5 by more than 24% and ACO by about 8%. On JM1, SA-ACO reaches a training accuracy that is 1% less than that of NN and a testing accuracy that is 2% less. However, it out-performs C4.5 and ACO but only slightly on both the training and testing sets.

In general, we can see from Table 9 that SA-ACO reaches results very close to NN and out-performs C4.5 and ACO by around 14% and 8% respectively on both the training and testing data. The small standard deviation of our algorithm proves its stability. A pair-wise statistical test comparing SA-ACO to each of the algorithms, shows that, except for JM1, the results obtained with SA-ACO are significantly better than C4.5 and ACO (two sample t-test, $p \leq 0.05$). On JM1, the difference in performance is not significant.

Table 3: JM1 and CM1 Metrics

Name	Description
loc	McCabe's line count of code
v(g)	McCabe "cyclomatic complexity"
ev(g)	McCabe "essential complexity"
iv(g)	McCabe "design complexity"
n	Halstead total operators + operands
v	Halstead "volume"
l	Halstead "program length"
d	Halstead "difficulty"
i	Halstead "intelligence"
e	Halstead "effort"
b	Halstead
t	Halstead's time estimator
lOCode	Halstead's line count
lOComment	Halstead's count of lines of comments
lOBlank	Halstead's count of blank lines
lOCodeAndComment	Halstead's count of lines of comments and code
uniq_Op	unique operators
uniq_Opnd	unique operands
total_Op	total operators
total_Opnd	total operands
branchCount	of the flow graph
defects	false,true - class has/has not one or more reported defects

Table 4: STAB2 Metrics

Name	Description
LCOM	lack of cohesion methods
COH	cohesion
COM	cohesion metric
COMI	cohesion metric inverse
OCMAIC	other class method attribute import coupling
OCMAEC	other class method attribute export coupling
CUB	number of classes used by a class
CUBF	number of classes used by a member function
NOC	number of children
NOP	number of parents
NON	number of nested classes
NOCONT	number of containing classes
DIT	depth of inheritance
MDS	message domain size
CHM	class hierarchy metric
NOM	number of methods
WMC	weighted methods per class
WMCLOC	LOC weighted methods per class
MCC	McCabe's complexity weighted method per class
DEPCC	operation access metric
NPPM	number of public and protected method in a class
NPA	number of public attributes

Table 5: Geotools and Weka Metrics

Name	**Description**
Lines	Number of lines
Statements	Number of computational statements (including if, while, etc.)
%Branches	Percentage of statements that cause a break in the sequential execution of statements (such as if, for, etc.)
Method calls	The total number of statements found inside methods divided by the number of methods found in the file).
%Comments	Percentage of lines with comments
Classes	number of classes and interfaces
Methods/Class	average number of methods per class (total method count divided by the total class count)
Avg Stmts/Method	average number of statements per method (The total number of statements inside methods divided by the number of methods).
Max Complexity	maximum complexity of the class (The complexity value of the most complex method in a source file.)
Max Depth	maximum depth in inheritance tree (The maximum nested block depth found. At the start of each source file the block level is zero. Depths up to 9 are recorded and all statements at deeper levels are counted as depth 9.)
Avg Depth	Average block depth: The average nested block depth weighted by depth.
Avg Complexity	Average complexity for each method It is computed as a simple arithmetic average of all complexity values measured for a source file.

Table 6: Parameters

ACO						
#of ants	#of iterations	Pheromone intensity	Pheromone seek radius	Pheromone evaporation rate	α	β
8	1000	1	1	0.7	0.5	0.5
SA						
		Initial Temperature	**Cooling Rate**	**Iterations**		
		1000	10^a	1000		

Table 7: Run time on each data set

Dataset	**Time(min)**
Geotools	22
Weka	90
STAB1	100
STAB2	20
CM1	15
JM1	240

Table 8: Average accuracy per data set

Dataset	**Algorithm**	**Training Accuracy(%)(stdv)**	**Testing Accuracy(%)(stdv)**
Geotools	NN	96.91(0.05)	95.91(0.38)
	C4.5	78.59 (4.26)	78.59(4.76)
	ACO	84.27 (3.36)	84.23(3.38)
	SA-ACO	96.83(0.13)	96.37(0.41)
Weka	NN	68.84 (0.72)	66.33 (2.52)
	C4.5	53.89(4.11)	53.88(4.27)
	ACO	58.80 (2.22)	58.62 (2.19)
	SA-ACO	67.70(0.73)	66.05(1.25)
STAB1	NN	90.12(1.07)	73.46(14.23)
	C4.5	73.56(14.17)	73.46(14.23)
	ACO	81.42 (0.44)	81.40 (0.58)
	SA-ACO	86.40(0.82)	85.64(1.07)
STAB2	NN	79.00 (2.16)	74.20 (2.66)
	C4.5	62.42(7.81)	62.29(8.48)
	ACO	65.66 (2.61)	65.41 (2.42)
	SA-ACO	74.65(1.47)	70.65(4.02)
CM1	NN	94.71 (0.33)	92.83 (1.35)
	C4.5	68.21(40.52)	68.22(40.57)
	ACO	84.90 (4.51)	84.72 (4.65)
	SA-ACO	94.92(0.45)	92.38(2.07)
JM1	NN	75.63 (0.31)	76.00 (2.28)
	C4.5	73.64(1.26)	73.64(1.47)
	ACO	73.47 (0.75)	73.39 (0.79)
	SA-ACO	74.62(0.32)	74.10(0.61)

Table 9: Average accuracy over all data sets

	Training Accuracy(%)(stdv)	**Testing Accuracy(%)(stdv)**
NN	84.20(0.77)	82.40(1.99)
C4.5	68.37(12.02)	68.35(12.30)
ACO	74.75(2.32)	74.63(2.34)
SA-ACO	82.52(0.65)	80.87(1.58)

6 Conclusion and Future Work

This paper proposes a hybrid algorithm that combines ACO, a swarm intelligence algorithm, and simulated annealing to search for a good software quality estimation model. Tested on data that describes stability of classes in an object-oriented paradigm and fault-proneness, our algorithm performs significantly better than C4.5 and ACO published in (Azar and Vybihal, 2011). Compared to NN, SA-ACO reaches an accuracy that is similar in most of the cases but it has the advantage of preserving the white-box nature of the prediction models. This is important for experts in the field as they can learn guidelines to attain the desired software quality characteristic at the design and implementation stage of the system. The approach is also generic enough to accommodate any software quality attribute. The classification problem that we tackle is binary. The algorithm can easily evolve to accommodate multi-classification label problems.

7 Acknowledgements

This work was supported by a grant from the School of Arts and Sciences Research and Development Council at the Lebanese American University.

References

Abreu, F. and Melo, W. 1996. Evaluating the Impact of Object-Oriented Design on Software Quality, *Proceedings of the 3rd International Symposium on Software Metrics*, IEEE, Berlin, Germany, p. 90.

Almeida, M. D., Lounis, H. and Melo, W. 1999. An investigation on the use of machine learned models for estimating software correctability, *International Journal of Software Engineering and Knowledge Engineering, Spcial Issue on Knowledge Discovery from Empirical Software Engineering Data* pp. 565–593.

Arisholm, E. and Briand, L. 2006. Predicting fault-prone components in a java legacy system, *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering* , ACM, Rio de Janeiro, Brazil, pp. 8–17.

Arisholm, E., Briand, L. and Fulglerud, M. 2007. Data mining techniques for building fault-proneness models in telecom java software: An experiment, *Proceedings of the 5th IEEE International Symposium on Software Reliability Engineering(ISSRE 2007)*, IEEE, Monterey, CA, USA, pp. 215–224.

Arisholm, E., Briand, L. and Johannessen, E. 2010. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models, *The Journal of Systems and Software* **83, no. 1**: 2–17.

Azar, D. 2010. A genetic algorithm for improving accuracy of software quality predictive models: A search-based software engineering approach, *International Journal of Computational Intelligence and Applications* **9**(2): 125–136.

Azar, D. and Harmanani, H. 2011. Heuristic approaches for optimizing the performance of rule-based classifiers, *The 12th IEEE International Conference on Information Reuse and Integration (IRI)*, IEEE, Las Vegas, USA, pp. 25–31.

Azar, D., Harmanani, H. and Korkmaz, R. 2009. A hybrid hueristic to optimize rule-based software quality estimation models, *Software and Information Technology* **51**(9): 1365–1376.

Azar, D. and Precup, D. 2007. An adaptive approach to optimize software component quality predictive models: Case of stability, *New Technologies, Mobility and Security*, Springer, pp. 297–310.

Azar, D. and Vybihal, J. 2011. An ant colony optimization algorithm to improve software quality prediction models: Case of class stability, *Journal of Information and Software Technology* **53**: 388–393.

Barnes, G. M. and Swim, B. R. 1993. Inheriting software metrics, *Journal of Object-Oriented Programming* **6**(7): 27–34.

Basili, V., Condon, K., Emam, K. E., Hendrick, R. and Melo, W. 1997. Characterizing and modeling the cost of rework in a library of reusable software components, *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, USA, pp. 282–291.

Briand, L., Devanbu, P. and Melo, W. 1997. An investigation into coupling measures for C++, *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, USA, pp. 412–421.

Briand, L., W$\ddot{u}$st, J., Lounis, H. and Ikonomovski, S. 1999. Investigating quality factors in object-oriented designs: an industrial case study, *Proceeding of the 21st International Conference on Software Engineering,*, ACM, Los Angeles, CA, USA, pp. 345–354.

Briand, L., Wust, J., Daly, J. and Porter, V. 2000. Exploring the relationships between design measures and software quality in object-oriented sytems, *Jounal of Systems and Software* **51**(245): 245–273.

Chidamber, S. R. and Kemerer, C. F. 1994. A Metrics Suite for Object Oriented Design, *IEEE Transactions on Software Engineering* **20**(6): 476–493.

Coppick, J. and Cheatham, T. 1992. Software metrics for object-oriented systems, *Proceedings of the ACM Computer Science Conference*, Kansas City, MO, USA, pp. 317–322.

Dallal, J. A. 2012. Fault prediction and the discrimative powers of connectivity-based object-oriented class cohesion metrics, *Information and Software Technology* **85**(5): 1042–1057.

Dallal, J. A. and Morasca, S. 2014. Predicting object-oriented class reuse-proneness using internal quality attributes, *Empirical Software Engineering* **19**: 775–821.

David, R.-C., Precup, R.-E., Petriu, E. M., Rădac, M.-B. and Preitl, S. 2013. Gravitational search algorithm-based design of fuzzy control systems with a reduced parametric sensitivity, *Information Sciences* **247**: 154–173.

Dorigo, M. and Caro, G. 1999. Ant colony optimization: a new meta-heuristic, *Congress of Evolutionary Computation*, Vol. 2, Washington D.C., USA, pp. 1470–1477.

Dorigo, M. and Stutzle, T. 2004. *Ant Colony Optimization*, MIT Press.

Fenton, N. and Neil, M. 1999. A critique of software defect prediction models, *IEEE Transactions on Software Engineering* **25**(5): 676–689.

Grosser, D., Sahraoui, H. and Valtchev, P. 2002. Predicting software stability using case-based reasoning, *Procceedings of the 17th International Conference on Automated Software Engineering*, IEEE Computer Society, Washington, DC, Edinburgh, U.K., pp. 295–298.

Harman, D. 2007. The current state and future of search based software engineering, *Proceedings of the Conference on the Future of Software Engineering*, IEEE Computer Society, Limerick, Ireland, pp. 342–357.

Hendersen-Sellers, B. 1991. Some metrics for object-oriented software engineering, *Proceedings of the 5th International Conference on Technology of Object-Oriented Languages and System*, Santa Barbara, CA, USA, pp. 131–139.

Henderson-Sellers, B. 1996. *Object-Oriented Metrics: Measures of Complexity*, Prentice Hall.

Kaaniche, M. and Kanoun, K. 1996. Reliability of a commercial telecommunications system,, *Proceedings of the International Symposium on Software Reliability Engineering,*, New York, USA, pp. 207–228.

Khoshgoftaar, T. M., Allen, E. B., Halstead, R., Trio, G. P. and Flass, R. M. 1998. Using Process History to Predict Software Quality, *Computer* **31**(4): 66–72.

Lessmann, S., Baesens, B., Mues, C. and Pietsch, S. 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings, *IEEE Transactions on Software Engineering* **34**: 485–496.

Li, W. and Henry, S. 1993a. Maintenance metrics for the object-oriented paradigm, *Proceedings of the First International Software Metrics Symposium*, Baltimore, Maryland, USA, pp. 52–60.

Li, W. and Henry, S. 1993b. Object-oriented metrics that predict maintainability, *Journal of Systems and Software* **23**: 111–122.

Lorenz, M. and Kidd, J. 1994. *Object-Oriented Software Metrics: A Practical Approach*, Prentice Hall.

Mao, Y., Sahraoui, H. and Lounis, H. 1998. Reusability hypothesis verification using machine learning techniques: a case study, *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, IEEE, Honolulu, Hawaii, USA, pp. 84–93.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H. and Teller, E. 1953. Equation of State Calculations by Fast Computing Machines, *The Journal of Chemical Analysis* **21**(6).

Moller, K.-H. and Paulish, D.-J. 1993. An empirical investigation of software fault distribution, *Proceedings of the First International Software Metrics Symposium*, Baltimore, Maryland, pp. 82–90.

Pedrycz, A. and Succi, G. 2005. Genetic granular classifiers in modeling software quality, *Journal of Systems and Software* **76**(3): 277–285.

Rizauddin, S. 2013. *A Hybrid of Ant Colony Optimization Algorithm and Simulated Annealing for Classification Rules*, PhD thesis, Universiti Utara Malaysia.

Selby, R. and Porter, W. 1990. Empirically Guided Software Development Using Metric-Based Classification Trees, *IEEE Software* **7**(2): 46–54.

Shepperd, M. J. 1993. *Software Engineering Metrics*, Vol. 1, McGraw-Hill.

Subramanyam, R. and Krishnan, M. 2003. Empirical analysis of ck measures for object- oriented design complexity: implications for software defects, *IEEE Transactions on Software Engineering,* **29**(4): 297–310.

Thwin, M. M. T. and 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.

Valdez, F., Melin, P. and Castillo, O. 2011. An improved evolutionary method with fuzzy logic for combining particle swarm optimization and genetic algorithms, *Applied Soft Computing* **11**(2): 2625–2632.

Vivanco, R. 2007. Improving predictive models of software quality using an evolutionary computational approach improving predictive models of software quality using an evolutionary computational approach, *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2007)*, Paris, France, pp. 503–504.

Yazdani, D., Sadeghi-Ivrigh, S., Yazdani, D., Sepas-Moghaddam, A. and Meybodi, M. R. 2015. Fish swarm search algorithm: A new algorithm for global optimization, *International Journal of Artificial Intelligence* **13**(2): 17–45.

Zăvoianu, A.-C., Bramerdorfer, G., Lughofer, E., Silber, S., Amrhein, W. and Klement, E. P. 2013. Hybridization of multi-objective evolutionary algorithms and artificial neural networks for optimizing the performance of electrical drives, *Engineering Applications of Artificial Intelligence* **26**(8): 1781–1794.

Zhou, Y. and Leung, H. 2007. Predicting object-oriented software maintainability using multivariate adaptive regression splines, *Journal of Systems and Software* **80**(8): 1349–1361.