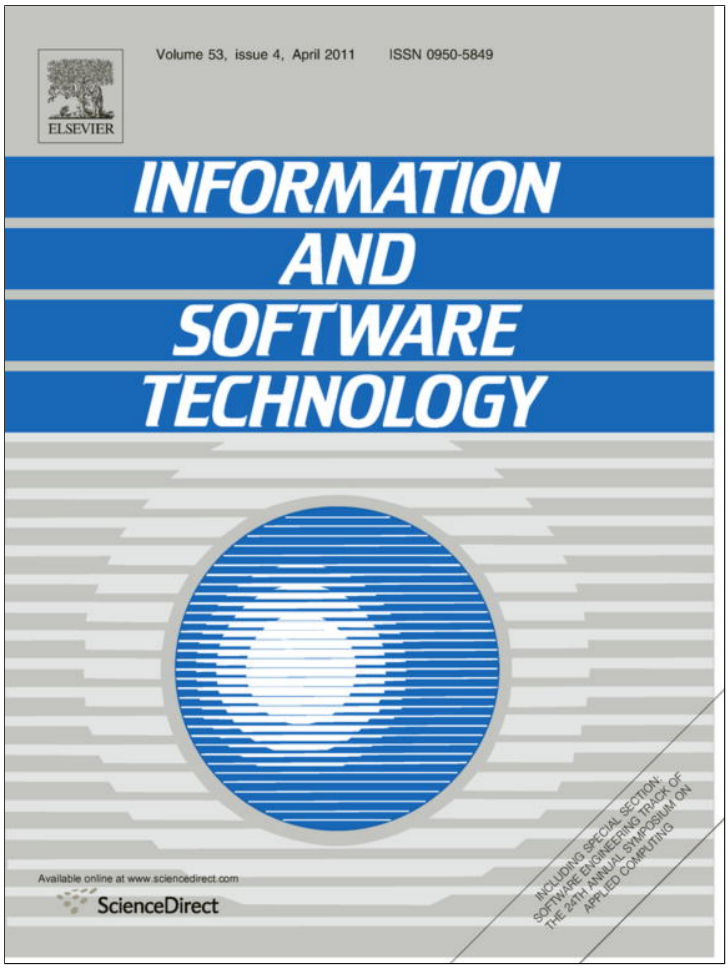

Information and Software Technology 53 (2011) 388–393

Contents lists available at ScienceDirect

Information and Software Technology

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

An ant colony optimization algorithm to improve software quality prediction models: Case of class stability

D. Azar [a,*], J. Vybihal [b]

[a] Department of Computer Science and Mathematics, Lebanese American University, Byblos 1h401 2010, Lebanon
[b] McGill University, School of Computer Science, 3480 University St., Montreal, Quebec, Canada H3A 2A7

ARTICLE INFO

Article history:
Received 23 December 2009
Received in revised form 29 November 2010
Accepted 30 November 2010
Available online 14 December 2010

Keywords:
Software quality
Metric
Search-based software engineering
Ant colony optimization

ABSTRACT

Context: Assessing software quality at the early stages of the design and development process is very difficult since most of the software quality characteristics are not directly measurable. Nonetheless, they can be derived from other measurable attributes. For this purpose, software quality prediction models have been extensively used. However, building accurate prediction models is hard due to the lack of data in the domain of software engineering. As a result, the prediction models built on one data set show a significant deterioration of their accuracy when they are used to classify new, unseen data.
Objective: The objective of this paper is to present an approach that optimizes the accuracy of software quality predictive models when used to classify new data.
Method: This paper presents an adaptive approach that takes already built predictive models and adapts them (one at a time) to new data. We use an ant colony optimization algorithm in the adaptation process. The approach is validated on stability of classes in object-oriented software systems and can easily be used for any other software quality characteristic. It can also be easily extended to work with software quality predictive problems involving more than two classification labels.
Results: Results show that our approach out-performs the machine learning algorithm C4.5 as well as random guessing. It also preserves the expressiveness of the models which provide not only the classification label but also guidelines to attain it.
Conclusion: Our approach is an adaptive one that can be seen as taking predictive models that have already been built from common domain data and adapting them to context-specific data. This is suitable for the domain of software quality since the data is very scarce and hence predictive models built from one data set is hard to generalize and reuse on new data.

1. Problem statement

Software quality is defined as the degree to which a software component or system meets specified requirements and specifications [37]. It is measured in terms of characteristics such as maintainability, reusability, etc. With the complexity of software systems on the rise, there is an increasing need for measuring such quality characteristics at an early stage of the software development cycle but this is not possible before the system is deployed and used for a certain period of time. However, a software system has attributes that can be used as good indicators of its quality characteristics. For example, in [33], the authors establish a relationship between reusability of software components and their complexity and volume. They show that highly reused components tend to have complexity and volume measures lower than those of less reused components. Reusability is one software quality characteristic that cannot be directly measured and complexity and volume are software attributes that can be measured and used as indicators of it. Several metrics have been proposed for measuring software attributes [9,11,15,17,25,31]. Examples of such metrics are: NOM (number of methods in the class) and NOC (number of children of the class). In this context, we speak of software quality estimation models which can be used to build relationships between the measurable attributes on the one hand and the software quality characteristic of interest on the other. These models can be either statistical models (such as regression models [21,28]) or logical models [1,12]. Our approach deals with logical models. These have been extensively used to predict several software quality characteristics due to their simplicity and intuitive aspect [46]. Moreover, they have a two-fold advantage since they give the prediction itself and they provide guidelines to attain a certain software quality characteristic. These guidelines can be incorporated at the early stages of software development to produce software components with a better quality characteristic.

* Corresponding author. Tel.: +961 9 547254x2408; fax: +961 9 547256.
E-mail address: danielle.azar@lau.edu.lb (D. Azar).

0950-5849/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2010.11.013

Logical models can be either decision trees or rule sets. The latter can be represented by logical expressions which aim at classifying a software component into one of several categories according to some tests on the metrics. However, the data used to build logical models is scarce. The reason is that companies do not systematically collect data and when they do, the data is company-specific. As a result, it becomes hard to generalize, to cross-validate or to reuse the models on new unseen data. In our work, we focus on one sub-characteristic of software maintainability namely stability [26]. During its life-time, software undergoes several changes and modifications. It is important for the software component to remain stable across different versions of the system. A component (a class in an OOP system) is said to be *stable* across two versions of the system, if it keeps its public interface; otherwise the class is *unstable*. In this context, we focus on the syntactic stability of the classes only. i.e. changes in the header of the methods not in the body or the semantics. The class is considered unstable if the use of its method (syntactic) has changed between different versions of the system. Addition and deletion of methods to the public interface of the class also incurs instability. We formulate our problem as a search-based software engineering one, a term first coined by Harman and Jones [24]. Search-based software engineering is a field that formulates software engineering problems as optimization problems and applies meta-heuristics techniques (genetic algorithms, simulated annealing, tabu search, etc.) to solve them [16,34]. The field has shown a lot of promise when applied to problems in project management [3,5,14], prediction in software engineering management [18,29], project planning and quality assessment [2,13,29], and software testing [23,32,41]. In this paper, we present an ant colony optimization (ACO) algorithm that adapts rule-based models built on one data set to new unseen data. This can be seen as adapting models that were built from one domain (possibly common domain) to context-specific data (could be company-specific data). The remainder of this paper is organized as follows. In Section 2, we give an overview of related work in the area of software quality prediction. In Section 3, we formulate the problem at hand and introduce the notation borrowed from the machine learning domain. In Section 4, we give a brief overview of ant colony optimization algorithms. In Section 5, we describe the ant colony optimization algorithm that we propose for the problem. In Section 6, we describe the experiments we performed and the obtained results. In Section 7, we conclude with a summary of the technique and a description of possible future work.

2. Related work

Machine Learning techniques have been widely used to build prediction models in the domain of software quality. In particular, Porter and Selby [36] have used machine learning to build decision trees as early as 1988. Mao et al. [33] use C4.5 [38] to build models that predict reusability of a class in an object-oriented software system based on the three attributes inheritance, coupling and complexity. In [10], the authors use C4.5 to build models that estimate the cost of rework in a library of reusable components. De Almeida et al. [4] use C4.5 to build rule sets that predict the average isolation effort and the average effort. Briand et al. [11] investigate the relationship between coupling and cohesion measures defined at the level of a class in an object-oriented system on one hand and fault-proneness on the other. More recent work was done by Arisholm et al. [6] and Briand and Arisholm [22] in the domain of prediction of fault-proneness. Work in the same domain was done by Khosgoftaar et al. [28] and Jiang et al. [27]. Lessman et al. [30] benchmarked several classification algorithms on the problem of predicting software fault-proneness-another software maintainability sub-characteristic, and found very little difference in their performance. Similar to us, they argue that the comprehensibility of the resulting classification models is very important as it illustrates the classification procedure thus improving our understanding of the software quality characteristic that we are studying (stability in our case, fault-proneness in their case). However, Lessman et al. [30] benchmark classification algorithms on multiple data sets hence giving a wider application to the resulting classification models. Arisholm et al. [7] compare different modeling techniques in a systematic way and the impact of selecting different types of measures as predictors. They find that the measures and techniques quality is highly dependent on the evaluation criteria used to assess the prediction models. In the context of stability of classes in an OO system, Tsantalis et al. [43] propose a methodology for assessing the probability that a class changes in future generations. They use logistic regression to show that the proposed probability measure is useful in predicting the change proneness. Grosser et al. [44] propose a case-based reasoning approach for the same problem. In their work, they consider each software component as a point in a multi-dimensional space where each metric represents one dimension. A distance function is then defined to compute the similarity between components and derive stability from it. Various search-based software engineering approaches have been also proposed in the domain of software quality in general. For example, Pedrycz and Succic [35] represent classifiers as hypberboxes and uses genetic algorithms to evolve existing models into new ones. Similar to our approach, this technique preserves the comprehensibility of the classifiers and can be easily extended to a problem with multiple classification labels. Vivanco [40] 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. Azar et al. [8,45] present a hybrid approach combining different meta-heuristics such as genetic algorithms, tabu search and simulated annealing to re-combine existing rule sets and create new ones. The attempt was successful and the obtained results promising. In this work, we present a different approach which consists of adapting a single model at a time to a new set of data. The idea behind this approach is to see how much we can optimize one model. The technique relies on ant colony optimization (ACO). Results show that our ACO out-performs C4.5 and random guessing.

3. Problem formulation and objective

The prediction problem that we are dealing with is a binary classification problem. The predictive model used for classification takes the form of a function f that takes as input a vector of attributes $(a_1, a_2, \ldots, a_n)$ and outputs a classification label y_i. In the context of our work, the attributes a_1, a_2, …, a_n are metrics (such as number of methods, number of children, etc.) considered relevant to the software quality factor being predicted (stability). The classification label y_i represents this software quality factor. In this problem, $y_i \in \{0(stable), 1(unstable)\}$. Each vector of attributes describes a component i.e. a class in an object-oriented software system. The objective is to find a function with a low error rate. The evaluation of f is done on a data set D. The latter has the form $D = \{(v_1, y_1), \ldots, (v_m, y_m)\}$ where each v_i is a vector of attributes and y_i is the classification label. D is partitioned into two parts, the *training set* D_{train} and the *testing set* D_{test}. Most learning algorithms take the training set as input and search the space of classifiers for one that minimizes the error on D_{train}. The output classifier is then evaluated (tested) on D_{test}. Examples of learning algorithms that use this principle are the back propagation algorithm for feed forward neural nets [39] and C4.5. The latter builds

classifiers in the form of decision trees. These can be transformed later on to rule sets. We call the latter *rule-based classifiers* or *rule-based models*. A rule-based classifier is a disjunction of conjunctive rules and a default classification label. The following example illustrates a rule-based classifier that predicts the stability of a component based on the metrics *NOM* (number of methods) and *CUB* (number of used classes).

*Rule*1: $NOM > 10 \rightarrow 1$
*Rule*2: $CUB \leqslant 2 \wedge NOM \leqslant 4 \rightarrow 0$
Defaultclass: 1

This classifier consists of two rules and a default classification label. The first rule classifies a class with number of methods (NOM) greater than 10 as unstable (classification label equal to 1). The second rule classifies a class with the number of used classes (CUB) less than or equal to 2 and number of methods (NOM) less than or equal to 4 as stable (0). The classification is sequential i.e. the first rule (from the top) whose left hand side is satisfied by a case classifies the case. If no such rule exists, the default classification label is used to classify the case (default classification 1).

In this work, we consider that all mis-classifications incur equal cost and thus, we evaluate the performance of a model (rule set) using its accuracy (percentage of cases correctly classified) (Eq. (1)). This measure is extracted from the confusion matrix shown in Table 1. In this table, n_{ij} is the number of cases in D that are classified by R as having classification label j while they actually have classification label i.

$$C(R) = \frac{\sum_{i=1}^{k} n_{ii}}{\sum_{i=1}^{k}\sum_{j=1}^{k} n_{ij}} \quad (1)$$

4. Overview of ant colony optimization algorithms

Ant colony optimization (ACO) algorithms have been widely used for optimization problems [19,20]. They are cooperative search algorithms inspired by the behavior of ants in search for food. In real life, ants explore routes from the nest to the food and leave pheromone traces on their path. Foraging ants can exploit this chemical substance to know which route is more likely to take them to the food in a shorter time. The larger the amount of detected pheromone, the most likely the foraging ants will take the route. During the process, pheromone evaporates. In artificial ant systems, ants build solutions to the problem by collecting information about problem characteristics. This information is encoded in pheromone. Hence, the ant is somehow guided by the pheromone and modifies the representation of the problem through the pheromone as well. The other ants can see this modification. During the whole process of solution building, pheromone evaporates. This helps avoid the convergence of all ants to one solution. In Fig. 1, we show the pseudo-code of a generic ACO and in the next section, we explain how we instantiate it to our problem.

Table 1
Confusion matrix of decision function f. n_{ij} is the number of cases with real label c_i classified by f as c_j.

Real label	Predicted Label			
	c_1	c_2	...	c_k
c_1	n_{11}	n_{12}	...	n_{1k}
c_2	n_{21}	n_{22}	...	n_{2k}
⋮	⋮	⋮	⋱	⋮
c_k	n_{k1}	n_{k2}	...	n_{kk}

5. The proposed ACO algorithm

ACO starts by taking one rule set at a time, applying small changes to it and evaluating the effect of these changes. This is repeated for every rule set separately. The core of the algorithm is in its *build_solution* procedure (Fig. 1). A rule set is represented to the ACO as an $n \times m$ matrix M where n is one plus the number of rules in the rule set and m is one plus the number of conditions in the longest rule in the rule set (Table 2). Row i of M represents the ith rule in the rule set (the first rule being the topmost rule) if $i \leqslant n-1$ or the default classification label if $i = n$.[1] Cell j in row i represents the jth condition in rule i if $j \leqslant m-1$ or the classification label of rule i if $j = m$. Since the rules do not all have the same length, some cells in the matrix are left empty.

The algorithm starts by creating several ants at random locations on the matrix.[2] The ants start moving downwards (Fig. 2). At each non empty cell, the ant *perturbs* the content of the cell. The *perturb* operation consists of changing the underlying cell according to its contents (Fig. 1). If the cell encodes a condition, then the *perturb* operation changes either the operator or the value in the cell (this is decided randomly with a 50% chance for the operator change to happen and 50% chance for the value to be changed). In the former case, if the operator is $\leqslant$, it is changed to $>$ and vice versa. In the latter case, the value of the condition is changed to a different one picked randomly from the set of cutpoint values for the metric in the training set. A cutpoint value for a metric is computed as the median of two values in the training data set at which the classification label changes. For this, the training data set is sorted by each metric at a time. The borders at which the classification label changes are noted and the median of the two border values is computed and inserted in the cutpoint set for this metric. We chose the cutpoint values of a metric because these are the values that are most likely to incur a change in the classification label. In the case where the cell encodes a classification label, then the *perturb* operation changes this to a different classification label (in our case, this changes 0–1 and vice versa). If the cell is an empty cell, the ant does not perform any operation on it and continues its march. When the ant hits the lowest row in the matrix, it moves to the adjacent cell to the right of the current one and starts moving upwards.When the ant reaches the topmost row, it moves to the right cell again and continues walking downwards. When it reaches the rightmost lowest cell in the matrix, it walks toward the cell on its left hand side and then upwards again. This ensures that every time the ant reaches a border, it goes to the longest unvisited adjacent cell. This results in the ant walking in an *S-shape*. We refer to this march as the *S-march*.

The described S-march results in an ant perturbing only slightly a rule by perturbing one condition at a time (for each objective function evaluation) or the classification label only. This way, we can assess how much effect each condition has on the rule in which it lies and hence on the whole rule set. The ant leaves pheromone on the modified cell if the perturbation results in an improvement(higher objective function value). This indicates that the change is good thus attracting other ants to this cell. The improvement is calculated according to an objective function f (Eq. (2)). Since the goal is to improve the accuracy of the underlying rule set R, f is set equal to $C(R)$ (Eq. (1)). The new (perturbed) rule set replaces the current one only if it gives a higher objective function value. Otherwise, ants continue their march to find better perturbations.

$$f = C(R) \quad (2)$$

[1] In this case, only the rightmost cell in the row is filled with the classification label.
[2] The number of ants is a parameter that is entered at the onset of the experiments.

```
Generic_ACO
{
   Initialize pheromone constant ti
   Repeat for all ants i:
      build_solution()
   Repeat for all ants i:
      update_pheromone()
   Repeat for all pheromones i:
      evaporate ti = (1 − r).ti
}

build_solution()
{
   R′=perturb_cell(m, row, col)
   f′=evaluate objective function f(R′)
   if (f′ > f* )
      R* = R′
      f* = f′
   list=detectMarker()
   if (marker_list ≠ empty)
      get largest pheromone within 3 cell range
      go straight to pheromone cell
   else
      smarch()
}

update_pheromone()
{
   if (improvement=TRUE)
      release pheromone (intensity=3)
}

perturb_cell(Matrix m, int i, int j)
{
   if(cell_empty(m, i, j)=FALSE)
     Matrix m'
     if (encodes_condition(m, i, j)=TRUE)
        r=generate_random()
        if (r < 0.5)
           m′=modify_operator(m, i, j)
        else
           m′=modify_value(m, i, j)
     else
        if (encodes_class_label(m, i, j)=TRUE)
           m′=flip_class_label(m, i, j)
     retrun m′
   else
     retrun m
}
```

Fig. 1. Pseudocode of the generic ACO and its adaptation to the problem. The procedures *build_solution*() and *update_pheromone*() are the core of ACO. *perturb_cell*() is the core of *build_solution*().

Table 2
Matrix representation of the rule set: *Rule*1: *NOC* ⩽ 5 ∧ *NOM* ⩽ 3 → 0; *Rule*2: *NPA* > 6 ∧ *NOM* > 10 ∧ *SIX* > 3 → 1; *Default class* 1 in the ACO algorithm.

NOC ⩽ 5	*NOM* ⩽ 3	–	0
NPA > 6	*NOM* > 10	*SIX* > 3	1
–	–	–	1

$NOC \leq 5$	$NOM \leq 3$	-	0
$NPA > 6$	$NOM > 10$	$SIX > 3$	1
-	-	-	1

Fig. 2. The S-march. Solid lines indicate the path of the ant before it hits the lowest rightmost cell and reverses directions (returning path is shown by dashed lines).

While walking, the ant checks for pheromone. If it detects any, it diverts from its *S-march* and goes directly to the pheromone (allowing itself diagonal moves). It is possible for the ant to detect several pheromone signals from different sources. In such a case, it moves to the one that has the highest intensity breaking ties randomly. The ant can detect pheromone up to three cells away only (we found three to be a good number given the relatively small size of the rule sets. As a matter of fact, very few rule sets have more than three conditions per rule). If, at any point during the walk, the ant encounters another one, it steps aside. For this, it checks the right cell first and moves to it if it lies within the boundaries of the matrix and is empty. Otherwise, the ant moves to the left cell.[3] The ants repeat this *march-perturb-release_pheromone* cycle a certain number of times (specified as a parameter to the algorithm). At the end of each iteration, the pheromone intensity is decreased to avoid the premature convergence of all ants to a subset of the conditions only.

[3] If the ant cannot step aside, it waits for one of the other ants to free its cell.

6. Experimentation and results

The data used to validate our technique was collected from open source. It describes stability of classes in an object-oriented software system and is fairly balanced. Table 3 shows the software systems used to build classification models with C4.5 and Tables 4–7 show the metrics that were extracted from these systems. In summary, they are metrics that measure four software attributes: size, cohesion, class coupling and inheritance. Detailed description of these metrics can be found in [9,15,17,25]. Fifteen different subsets of metrics were created by combining one, two, three or four of these groups of metrics. This setup helps assess the relationships between the software attribute (or a combination of them) on the one hand and the stability of the software system on the other. For example, cohesion and coupling metrics were combined to show the relationship between these two software quality attributes on one hand and stability on the other hand. These subsets were used with the nine chosen software systems to create 135 data sets. C4.5 was used to construct a decision tree from each data set. Constant classifiers and classifiers with an error rate higher than 40% were eliminated and 23 retained. The decision trees were then converted to rule sets by C4.5. These can be thought of as

Table 3
Software systems used to build classifiers with C4.5.

Software System	# of versions	# of classes	Location
Bean browser	6	388–392	alternativeTo
Free	9	46–93	Oracle
Javamapper	2	18–19	SourceForge
Jchempaint	2	84	SourceForge
Jigsaw	4	846–958	SourceForge
Jlex	4	20–23	Princeton University
Jms	2	106	SourceForge
Voji	4	16–39	SourceForge

Table 4
Cohesion metrics.

Name	Description
LCOM	Lack of cohesion on methods
COH	Cohesion
COM	Cohesion metric
COMI	Cohesion metric inverse

Table 5
Coupling metrics.

Name	Description
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

Table 6
Inheritance metrics.

Name	Description
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

Table 7
Size metrics.

Name	Description
NOM	Number of methods
WMC	Weighted methods per class
WMC_LOC	LOC weighted methods per class
MCC	McCabe's complexity weighted methods per class
DEPCC	Operation access metric
NPPM	Number of public and protected methods in a class
NPA	Number of public attributes

simulating classification models built from common domain data. They are adapted by ACO to the systems shown in Table 8. This set-up helps assess the performance of the models on new data. The new data can simulate company-specific data.

We performed several experiments with the algorithm changing number of ants and number of iterations for each ant. We show the results we obtained by setting the number of ants to 4 and the number of iterations to 30 for all experiments (Table 9). We chose these values after several trials showed that an increase in these numbers increased the running time of the algorithm and had very little effect (if any) on the quality of the obtained results. This shows that the algorithm is robust to such parameters variations. We believe that this is also due to the relatively small size of the rule sets and hence of the matrix that the ants march on. To accurately estimate the accuracy of the rule sets, we used 10-fold cross-validation [42] (10 is a commonly used number) on the data set in Table 8. With this technique, the data set is split into 10 folds of roughly equal size. The algorithm is seeded with one rule set at a time, trained on the union of nine folds and tested on the remaining fold. This is repeated 10 times using each time a different fold as a testing set. Also, in order to account for the element of randomness in the ACO algorithm, we repeat each experiment 30 times and we report the average results over the 30 runs. Table 10 shows the average, maximum and minimum accuracies obtained with each of the algorithms on the testing data. We can see from the tables that ACO scores the highest results of all in its minimum, maximum and average accuracies. It also shows a small standard deviation indicating that it is fairly stable. In order to test ACO in a more global fashion, we also built random rule sets on the same data set and compared our ACO to it. For this purpose, we created 30 random rule sets by randomly selecting attributes and values from the data set and forming conditions and rules of random size. Table 10 shows that our approach out-performs significantly random guessing and that random guessing is very unstable (large standard deviation).

Table 8
Software systems used to train and test the heuristics.

Software system	# of versions	# of classes	Location
Jedit	2	464–468	SourceForge
Jetty	6	229–285	SourceForge

Table 9
ACO parameters.

Number of ants	Number of iterations per ant
4	30

Table 10
Average, minimum, maximum and standard deviation of testing accuracy (percentages) of ACO, C4.5 and random guessing.

	Avg.	Min.	Max.	stdv.
ACO	64	56	72	0.04
C4.5	58	34	68	0.1
Random guessing	28	14	60	30

In order to test for the significance of our results, we use the Wilcoxon signed rank test to compare ACO to C4.5. Results show a $z - ratio$ of 2.8 indicating a significance level of 0.005 proving that it is unlikely that the observed results are due to chance. We do not include random guessing in our comparisons since it is unstable showing a very large variance in the obtained results.

7. Conclusion and future work

In this paper, we proposed an ant colony optimization algorithm to optimize existing software quality estimation models by adapting each, separately, to new unseen software systems. This can be seen as taking models built from common domain knowledge and adapting them to some context-specific (company) data. This is important in the case of software engineering because data is scarce and seldom made public. We have conducted experiments on a data set describing the stability of classes in an object-oriented system. ACO out-performs both C4.5 and random guessing. The approach also preserves the white-box nature of the obtained models. These provide the classification label as well as its source. We point out that the data set that we used is noisy and hence expect ACO to improve the results even more on a data set with less noise. It is certainly interesting to test our approach on data related to different software quality characteristics (such as reusability). Our ACO is limited by the size of the initial rule sets it is seeded with. In future work, we plan to design the algorithm in such a way that it allows the creation of rule sets of different complexity than the initial ones. Another idea would be to modify the algorithm in such a way that it uses several rule sets at the same

time. It would then be interesting to see how the performance of ACO varies with the number of rule sets to combine. Also, the problem that we are dealing with in this work is a binary classification of classes. In other words, a class is considered as stable or unstable depending on modifications in its public interface. Adding more information to the classification that would relate it to the degree of change in a class would be an interesting approach for future work.

Acknowledgement

This work was supported in part by a grant from the Lebanese National Council for Scientific Research (CNRS), and from the Research Council at the Lebanese American University.

References

[1] A. Agresti, Categorical Data Analysis, John Wiley and Sons, 1990.
[2] J. Aguilar-Ruiz, I. Ramos, J.C. Riquelme, M. Toro, An evolutionary approach to estimating software development projects, Information and Software Technology 43 (14) (2001) 875–882.
[3] E. Alba, F.B. e. AbreuChicano, Software project management with GAs, Information Sciences 177 (11) (2007) 2380–2401.
[4] M.A. De Almeida, H. Lounis, W. Melo, 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, October 1999.
[5] G. Antoniol, M. Di Penta, M. Harman, Search-based techniques for optimizing software project resource allocation, in: Proceedings of the 2004 Conference on Genetic and Evolutionary Computation (GECCO '04), vol. 3103, 2004, pp. 1425–1426.
[6] E. Arisholm, L. Briand, M. Fulglerud, Data mining techniques for building fault-proneness models in telecom java software: an experiment, in: Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE 2007), IEEE, 2007, pp. 215–224.
[7] E. Arisholm, L. Briand, E.B. Johannessen, A systematic and comprehensive investigation of methods to build and evaluate fault prediction models, The Journal of Systems and Software 83 (1) (2010) 2–17.
[8] D. Azar, H. Harmanani, R. Korkmaz, A hybrid hueristic to optimize rule-based software quality estimation models, Information and Software Technology (2009).
[9] G.M. Barnes, B.R. Swim, Inheriting software metrics, JOOP 6 (7) (1993) 27–34.
[10] V. Basili, K. Condon, K. El Emam, R.B. Hendrick, W.L. Melo, Characterizing and modeling the cost of rework in a library of reusable software components, in: Proceedings of the 19th International Conference on Software Engeineering, Boston, MA, May 1997, pp. 282–291.
[11] L. Briand, P. Devanbu, W. Melo, An investigation into coupling measures for C++, 1997, in: Proceedings of the 19th International Conference on Software Engineering, 1997.
[12] L.C. Briand, W.M. Thomas, C.J. Hetmanski, Modeling and managing risk early in software development, in: Proceedings of the 15th International Conference on Software Engineering, 1993, pp. 55–65.
[13] C.J. Burgess, M. Lefley, Can genetic programming improve software effort estimation? a comparative evaluation, Information and Software Technology 43 (14) (2001) 863–873.
[14] C.K. Chang, C. Chao, T.T. Nguyen, M. Christensen, Software project management net: a new methodology on software management, in: Proceedings of the 22nd Annual International Computer Software and Applications Conference (COMPSAC '98), 1998, pp. 534–539.
[15] S. Chidamber, C. Kemerer, A metrics suite for object-oriented design, IEEE Transactions on Software Engineering 20 (1994) 476–493.
[16] J. Clark, J.J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, M. Shepperd, Reformulating software engineering as a search problem, IEE Proceedings – Software 150 (3) (2003) 161–175.
[17] J.C. Coppick, T.J. Cheatham, Software metrics for object-oriented systems, in: In —'em CSC '92 Proceedings, 1992, pp. 317–322.
[18] J.J. Dolado, On the problem of the software cost function, Information and Software Technology 43 (1) (2001) 61–72.
[19] M. Dorigo, G.D. Caro, Ant colony optimization: a new meta-heuristic, in: Congress of Evolutionary Computation, vol. 2, 1999, pp. 1470–1477.
[20] M. Dorigo, T. Stutzle, Ant Colony Optimization, MIT press, 2004.
[21] F.B.E. Abreu, W.L. Melo, Evaluating the impact of object-oriented design on software quality, in: Proceedings of the 3rd International Symposium on Software Metrics, IEEE, 1996, p. 90.
[22] L.C. Briand, E. Arisholm, Predicting fault-prone components in a JAVA legacy system, in: IESE'06, 2006.
[23] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, M. Roper, Testability transformation, IEEE Transactions on Software Engineering 30 (1) (2004) 3–16.
[24] M. Harman, B. Jones, Search based software engineering, Journal of Information and Software Technology 43 (14) (2001) 833–839.
[25] B. Henderson-Sellers, Some metrics for object-oriented software engineering, in: The First IEEE International Conference on New Technology and Mobile Security, 2007.
[26] ISO9126, Software product evaluation-quality characteristics and guidelines for their use, ISO/IEC Standard-ISO9126, 1991.
[27] Y. Jiang, B. Cukic, T. Menzies, Fault prediction using early lifecycle data, in: The 18th IEEE International Symposium on Software Reliability, IEEE, 2007.
[28] T.M. Khoshgoftaar, Allen EB, J. Deng, Using regression trees to classify fault-prone software modules, IEEE Transactions on Reliability 51 (2002) 455–462.
[29] C. Kirsopp, M. Shepperd, J. Hart, Search heuristics, case-based reasoning and software project effort prediction, in: Proceedings of the 2002 Conference on Genetic and Evolutionary Computation (GECCO '02), 2002, pp. 1367–1374.
[30] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: a proposed framework and novel findings, IEEE Transactions on Software Engineering 34 (2008) 485–496.
[31] W. Li, S. Henry, Maintenance metrics for the object-oriented paradigm, in: Proceedings of the First International Software Metrics Symposium, 1993.
[32] Z. Li, M. Harman, R.M. Hierons, Meta-heuristic search algorithms for regression test case prioritization, IEEE Transactions on Software Engineering 33 (4) (2007) 225–237.
[33] Y. Mao, H.A. Sahraoui, H. Lounis, Reusability hypothesis verification using machine learning techniques: a case study, in: IEEE Automated Software Engineering Conference, 1998.
[34] P. McMinn, Search-based software test data generation: a survey, Software Testing, Verification, and Reliability 14 (2) (2004) 105–156.
[35] W. Pedrycza, G. Succic, Genetic granular classifiers in modeling software quality genetic granular classifiers in modeling software quality, Journal of Systems and Software 76 (3) (2005) 277–285.
[36] A. Porter, R. Selby, Learning from examples: generation and evaluation of decision trees for software resource analysis, Software Engineering 14 (12) (1988) 1743–1757.
[37] R.S. Pressman, Making Software Engineering Happen, A Guide for Instituting the Technology, Prentice-Hall, 1988.
[38] J.R. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufman, 1993.
[39] G.D.E. Rumelhart, Hinton, R. Williams, Learning representations by back-propagation errors, Nature 323 (1986).
[40] R. Vivanco, Improving predictive models of software quality using an evolutionary computational approach improving predictive models of software quality using an evolutionary computational approach, in: Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2007), 2007, pp. 503–504.
[41] J. Wegener, A. Baresel, H. Sthamer, Evolutionary test environment for automatic structural testing, Information and Software Technology 43 (14) (2001) 841–854.
[42] R. Kohavi, A study of cross-validation and bootstrap for accuracy estimation and model selection, in: Proceedings of the Fourteenth International Conference Artificial Intelligence (IJCAI 1995), Morgan Kaufmann, 1995, pp. 1137–1143.
[43] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, Predicting the probability of change in object-oriented systems, in: IEEE Transactions on Software Engineering, IEEE Computer Society, 2005, pp. 601–614.
[44] D. Grosser, H.A. Sahraoui, P. Valtchev, Predicting software stability using case-based reasoning, in: 17th International Conference on Automated Software Engineering, 2005, pp. 295–298.
[45] D. Azar, H. Harmanani, R. Korkmaz, Predicting stability of classes in an object-oriented system, Journal of Computational Methods in Science and Engineering (2010).
[46] T. Fawcett, Using rule sets to maximize ROC performance, in: IEEE international Conference on Data Mining, IEEE Computer Society, 2001, pp. 131–138.