Use of a Genetic Algorithm to Identify Source Code Metrics Which Improves Cognitive Complexity Predictive Models

Rodrigo Vivanco[1,2]
[1] *Biomedical Informatics Group, Institute for Biodiagnostics, National Research Council*
[2]*Department of Computer Science, University of Manitoba, Winnipeg, Canada*
Rodrigo.Vivanco@nrc-cnrc.gc.ca

Abstract

In empirical software engineering predictive models can be used to classify components as overly complex. Such modules could lead to faults, and as such, may be in need of mitigating actions such as refactoring or more exhaustive testing. Source code metrics can be used as input features for a classifier, however, there exist a large number of measures that capture different aspects of coupling, cohesion, inheritance, complexity and size. In a large dimensional feature space some of the metrics may be irrelevant or redundant. Feature selection is the process of identifying a subset of the attributes that improves a classifier's discriminatory performance. This paper presents initial results of a genetic algorithm as a feature subset selection method that enhances a classifier's ability to discover cognitively complex classes that degrade program understanding.

1. Introduction

Producing high quality products in a timely and cost-effective manner is one of the aims in software engineering. Locating modules that are fault prone or show evidence of high cognitive complexity can assist organizations improve product quality and reduce maintenance costs.

Predictive models have been a focus of research in many empirical studies of software systems [1]. Many of these studies attempt to associate a causal relationship between a system's structural metrics with external objective measures of quality such as component faults.

Being able to identify system modules that are likely to be problematic would assist project managers to take corrective actions, such as increased source code inspection, refactoring, assigning the best developers to such modules or performing more exhaustive testing.

Many organizations have adopted object-oriented technologies for product development and numerous studies have been conducted into the formulation of metrics to quantify various aspects of a system's design and implementation. Chidamber and Kemerer [2] introduced six metrics that attempt to capture the coupling, cohesion, inheritance and complexity in object-oriented designs. Briand et al. have identified at least 15 different cohesion and 30 coupling measures in the literature [3, 4].

Many of the proposed metrics measure similar structural aspects of the system. As such, most metrics exhibit various levels of correlation. For example, some complexity measures can be associated with simple size measures such as the number of lines of code. Using correlated measures may decrease the accuracy of multivariate predictive models, especially linear models.

A feature selection process using evolutionary computational methods, in particular a genetic algorithm [5], is proposed as a means of discovering source code metrics that improve the performance of predictive models. Genetic algorithms have been successfully used in the medical field for feature selection [6]. In software engineering they have been used to modify models [7] but not to identify source code metrics to be used in predictive models.

2. Background

Various techniques can be used to reduce a large collection of metrics to a smaller set that exhibit strong discriminatory powers in the identification of fault-prone classes [8]. One approach is to use univariate linear regression analysis to identify potentially useful predictors to utilize in the model; however, it does not take into account the discriminatory power of using combined measures.

If the number of available metrics is not large, stepwise selection/elimination could be used. In

15th IEEE International Conference on Program Comprehension (ICPC'07)
0-7695-2860-0/07 $20.00 © 2007 IEEE

forward stepwise selection, each feature is added to the model, one at a time, and if it improves its performance it is kept, otherwise it is rejected. It could be that a rejected feature would have improved performance in combination with a metric not yet included in the model. But since it has been rejected, it will not be taken into account.

Backward stepwise elimination starts with all the features. Each feature is tentatively removed and the model tested. The feature that has the least positive impact on the model's performance is permanently eliminated from the model. Since the features are removed one at a time, this approach is sensitive to local minima, as the process stops when model performance is not improved.

Another approach is to use principal component analysis (PCA), based on linear combinations of the metrics that maintain the maximum variance of the dataset. With PCA, all the metrics are considered when calculating loading factors. The loading factor of the metrics in one dimension (a principal component) is an indication of the importance of the metrics in explaining the variance.

Metrics with a loading factor above a user-defined threshold are then used with the predictive model. It is assumed that the metrics that explain the maximum variance will result in more accurate models. Setting the loading factor threshold is another limitation of PCA, as it is user dependent.

In the absence of theoretical reasons to choose a particular set of metrics and the observed association of the various metrics, there is a need for an effective, flexible method of feature subset selection in order to augment the performance of predictive models and illuminate the potential theoretical underpinnings of the significant metrics.

An initial study to ascertain source code metrics that improve the detection of cognitively complex classes in a Java-based medical data analysis application has been conducted. Early results shows that a genetic algorithm could be an effective means of feature subset selection for predictive models in software engineering.

3. Cognitive Complexity Dataset

It is theorized that structural properties such as coupling, cohesion, functional complexity and inheritance have an impact on the cognitive complexity of the system [9]. That is, it places a "mental burden" on developers, inspectors, testers and maintainers to understand the system, both at the component level and system level. It is further theorized that classes that exhibit high cognitive complexity result in poor quality components and are also likely to be error prone.

Expert judgment on the cognitive complexity of components, for example: a ranking on how difficult a module would be to understand and modify, could be used when developing a predictive model. For this study, two developers (both with over 5 years of OO development experience) were asked to inspect a Java-based application used for the analysis of functional MRI datasets [10].

Both developers were involved in the design, development and maintenance of various aspects of the system over a two-year period. The developers were asked to examine all classes and rank each one as low, medium or high for the purpose of future maintenance.

They were not given any specific instructions on how to rank the classes but to use their tacit knowledge and experience with OO system and developing this particular application. Software objects labeled as low were considered easy to understand, while a high ranking implied difficulty to fully comprehend and would take considerably much more effort to maintain.

The scores of 365 classes were averaged, with 247 classes labeled as low, 71 medium and 47 high. Source code measurements, 63 metrics for each class, were computed using a commercial source code inspection application.

A linear discriminant analysis (LDA) classifier was used as the multivariate linear model. LDA is a conventional supervised classification strategy used to determine linear decision boundaries between groups while taking into account between-group and within-group variances [11].

4. Genetic Algorithms

Genetic algorithms (GA), based on the evolutionary mechanics of survival of the fittest, are used in optimization and search problems where a clear analytical solution is not readily available. With the large number of available source code metrics and their various confounding effects, a GA can be used to discover a combination of metrics that improve the predictive performance of classifiers.

A genetic algorithm heuristically searches for an optimal set of solutions to a problem by simulating evolution. In GA, a solution (a set of software metrics) is encoded in a gene and a collection of solutions (genes) makes up a population. Combining the genome of selected parents followed by random mutations modifies a population of solutions.

A fitness value, how well the solution encoded in the gene solves the problem, is associated with a gene. Genetic algorithms are based on the process of

Darwinian evolution; over many generations, the "fittest" individuals tend to dominate the population. In the context of this study, the fittest individual results in the best classifier performance.

The simplest way to represent a gene is to use a string of bits, where '0' means the bit is off and '1' means the bit is on. For this problem domain, the N metrics were encoded in an N-bit mask vector. A zero bit means the metric is not to be used with the classifier. All the metrics with a corresponding bit set to '1' constitute the metrics sub-set to evaluate with the fitness function (LDA predictive model).

Various GA parameters can be adjusted that influence the effectiveness and speed of the solution search. The number of genes in a population, the mutation rate, the number of genes to keep for the next generation, and the number of generations are some of most common parameters. The number of genes in the population and the number of generations will have the largest impact on computational speed and in avoiding local minima. Larger populations are more robust, but take more time to evolve. The most computationally intensive aspect of a GA is the calculation of a gene's fitness value (how well the predictive model performs).

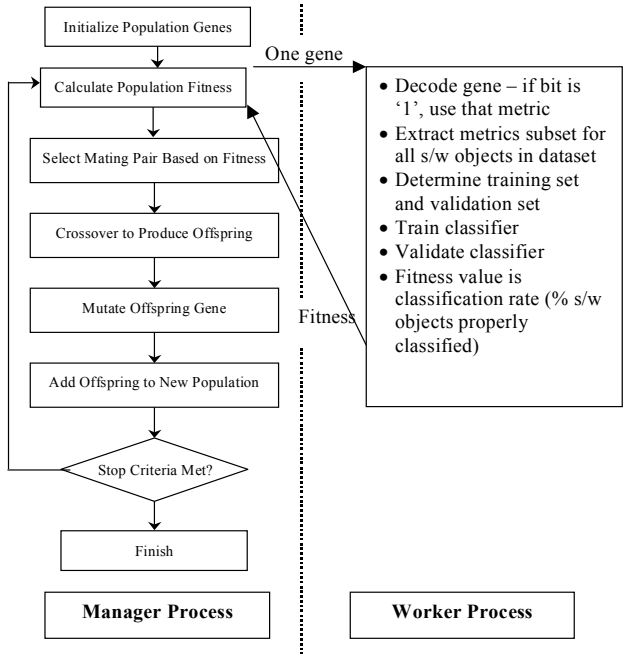

Figure 1. Manager/worker parallel algorithm for a GA.

Genetic algorithms lend themselves naturally to parallelization [12]. The fitness function for one gene is usually executed independently of other genes in the population. That means one process can calculate the fitness function for one gene in a population, while another process does the same for another gene in parallel. Once all the genes in the current population are evaluated, the new population is evolved and the process is repeated for the next generation.

Using a parallel GA greatly enhances the efficiency of the search and larger populations over longer generations can be used. Figure 1 illustrates the worker/manager parallel GA that has been implemented to search for optimal metrics.

5. Initial Results

The expert ranking of the Java classes generated three cognitive complexity groups: low, medium and high. To establish a baseline, all the available metrics were used with LDA using the leave-one-out validation method. Then, a well-known subset of the features, the CK metric suite, was used to determine if the model would improve. Finally, the GA algorithm (100 genes in the population, running for 50 generations) was used to find alternate metrics subsets.

	Low	Medium	High
All Metrics	92%	43%	63%
CK Metrics	97%	28%	71%
GA Metrics	93%	56%	81%

Table 1. Classifier performance for three groups.

Table 1 shows the percent correct classification for each ranking for the various metrics subsets. It is clear that identifying low cognitive complexity is not difficult, but distinguishing between medium and high complexity is challenging.

Identifying modules that may need mitigation actions is important for organizations. Using all the available metrics, only 63% of the modules were properly classified as difficult to understand. The CK metrics suite performed better at 71% but the GA algorithm, with a subset of 34 metrics, had the highest performance at 81%.

	Low/Medium	High
All Metrics	95%	63%
CK Metrics	97%	70%
GA Metrics	97%	83%

Table 2. Classifier performance for two groups.

Since correctly identifying problem modules is more important than overall classifier performance, the dataset was divided into two groups, high complexity modules and the low/medium complexity. With this

grouping, using all the metrics or the CK subset did not result in improved performance, but the GA was able to increase slightly to 83% with 23 metrics, as shown in Table 2.

6. Summary and Future Work

The main contribution of this research is a new approach for the selection of object-oriented source code metrics with respect to predictive models. A parallel GA prototype that performs feature subset selection of metrics has been developed with Scopira [13]. Once the tool is fully developed it will be made available to the research community.

This research is currently in the initial stages. The parallel GA has been developed and tested on proprietary Java source code using an LDA predictive model. A neural net (NN) classifier will also be used as a non-linear predictive model. A comparison of the efficacy and metrics used by linear (LDA) and non-linear (NN) will be made.

Expert judgment on the cognitive complexity of classes was used in this study. However, obtaining expert judgment is not trivial is susceptible to confounding effects such as developer experience and familiarity with the code.

The number of faults may be used as a proxy to cognitive complexity and will be considered in future work. Publicly available metrics datasets [14] will be investigated as this allows for comparison by researchers using other techniques.

The author sees this research area as a step towards developing tools to assist developers evaluate the quality of software products for a given set of objectives, such as cognitive complexity, fault-proneness or any external measure an organization deems important. By using search-based methods such as genetic algorithms, classifiers can be developed that are tuned to a particular organization's practices.

These tools can be used to identify metrics and generate quality models using training sets and objectives acceptable to a particular organization. In essence, the predictive models can then automate an organization's source code inspections to evaluate the quality of components.

8. References

[1] L. Briand, E. Arisholm, S. Counsell, F. Houdek and P. Thevenod-Fosse, "Empirical Studies of Object-Oriented Artifacts, Methods, and Processes: State of the Art and Future Directions", *Empirical Software Engineering*, 4, 4(1999), pp. 387-404.

[2] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, 20, 6 (1994), pp. 476-493.

[3] L.C. Briand, J.W. Daly and J. Wust, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems", *Emp. Software Engineering*, 3 (1998), pp. 65-117.

[4] L.C. Briand, J.W. Daly and J. Wust, "A Unified Framework for Coupling Measurement in Object-Oriented Systems", *IEEE Transactions on Software Engineering*, 25, 1 (1999), pp. 91-121.

[5] D. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, 1989.

[6] M.L. Raymer, W.F. Punch, E.D. Goodman, L.A. Kuhn and A.K. Jain, "Dimensionality Reduction Using Genetic Algorithms", *IEEE Trans. on Evolutionary Computation*, 4, 2 (2000), pp. 164-171.

[7] D. Azar, D. Precup, S. Bouktif, B. Kegl and H. Sahraoui, "Combining and Adapting Software Quality Predictive Models by Genetic Algorithms", *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE'02)*, 2002.

[8] L.C. Briand and J. Wuest, "Empirical Studies of Quality Models in Object-Oriented Systems", *Advances in Computers*, 56 (2002), pp. 97-166.

[9] K. El Emam, "Object-Oriented Metrics: A Review of Theory and Practice", in Advances in Software Engineering, editors, H. Erdogmus, O. Tanir, Springer, 2002.

[10] N.J. Pizzi, R. Vivanco, R.L. Somorjai, "EvIdent: a functional magnetic resonance image analysis system", *Artificial Intelligence in Medicine,* 21**(**2001**),** pp. 263-269.

[11] C.D. Duda P.E. Hart and D.G. Stork, *Pattern Classification 2nd Edition*, John Wiley & Sons, 2000.

[12] E. Cantu-Paz, *Effective and Accurate Parallel Genetic Algorithms***,** Kluger Academic Publishers, 2000.

[13] A. Demko, R. Vivanco, N.J. Pizzi, "Scopira: An Open Source C++ Framework for Biomedical Data Analysis Applications - A Research Project Report", in P*roceedings of Object-Oriented Programming, Systems, Languages and Applications*, San Diego, October, 2005.

[14] PROMISE Repository of empirical software engineering data: html:www.promisedata.org/