Identifying Effective Software Metrics Using Genetic Algorithms

R.A. Vivanco, N.J. Pizzi
Institute for Biodiagnostics, National Research Council Canada
435 Ellice Avenue, Winnipeg MB R3B 1Y6
Rodrigo.Vivanco@nrc-cnrc.gc.ca

Abstract

Various software metrics may be used to quantify object-oriented source code characteristics in order to assess the quality of the software. This type of software quality assessment may be viewed as a problem of classification: given a set of objects with known features (software metrics) and group labels (quality rankings), design a classifier that can predict the quality rankings of new objects using only the software metrics.

We have obtained a variety of software measures for a Java application used for biomedical data analysis. A system architect has ranked the quality of the objects as low, medium-low, medium or high with respect to maintainability. A commercial program was used to parse the source code identifying 16 metrics. A genetic algorithm (GA) was implemented to determine which subset of the various software metrics gave the best match to the quality ranking specified by the expert. By selecting the optimum metrics for determining object quality, GA-based feature selection offers an insight into which software characteristics developers should try to optimize.

Keywords: Software metrics, evolutionary computing, genetic algorithms.

1. INTRODUCTION

Project leaders commonly use software metrics as a measure of the overall quality of the design and the eventual implementation of systems. The ability to predict the quality of a software object from a set of software metrics is in essence a problem of classification. Given a set of objects with known features (software metrics) and group labels (quality rankings), design a classifier that can predict the quality of new objects using only the computed metrics. For purposes of training the classifier, the group labels representing quality could be determined by a panel of experts (system architects) whereas the software metrics may be assembled automatically.

Software metrics are used to quantitatively map a set of numerical values, such as the number of lines of code in a file or the number of methods in a class, to a subjective measure of quality, in terms of the apparent complexity, maintainability and usability. Many such software metrics have been proposed in the literature [1], some metrics are based on experience and intuition, others derived from a more theoretical basis.

Not all metrics provide the same classification power, and a previously unforeseen combination of metrics may capture the insight that an expert uses when attributing a quality ranking to a software object. We propose using a genetic algorithm feature selection procedure to indicate the optimal metrics used in the classification process. The effectiveness of this approach was examined using the software objects from EvIdent [2], a Java-based biomedical image data analysis system.

2. SOFTWARE METRICS

All 338 software objects in EvIdent were subjectively labeled by an experienced software architect in terms of maintainability. The architect (7 years of programming experience, 6 years with object-oriented systems and 3 years with Java) was asked to rank each Java class as low, medium-low, medium or high. High quality objects should be relatively easy for him to adapt and modify, while low quality objects would require much more effort to determine the purpose of the class, and to determine the potential side effects of modifying the code.

The software metrics that were used to classify the Java software objects are shown in Table 1. A set of 16 metrics were obtained from an evaluation version of Krakatau Metrics Essential Metrics tool [3]. Due to the nature of the application, it was noted that data model objects were relatively simple when compared to graphical user interface (GUI) classes, so an additional metric value was used to aid the classifier. The category of each software object was indicated, a distinction was made between GUI, data model, algorithm, and all other objects.
Memc Description

TYP GUI (=1), Data Model (=2), Algorithm (=3), Other (=4)

LOC The number of lines of code
SLC The number of lines in a file containing source code (SLC = LOC - CLC)
CLC The total number of lines of code in the project that contains comments
WLC The number of lines containing white space
RCC The ratio of comment lines of code to total lines of code including white space and comments (LOC)
RCS The ratio of comment lines of code to source lines of code
SMC Number of semicolons
MET Number of methods
ANL The mean string length of method names
ACN The mean number of control statements (iteration, selection) within a method
AE The mean number of executable statements within a method
ALC The mean number of lines of code within a method
ASC The mean number of semi-colons within a method
ASL The mean number of source lines of code within a method
ACC The mean cyclomatic complexity of a method (the number of possible paths through an algorithm by counting the number of distinct regions on a flow graph)
AEC The mean essential complexity of a method (a measure of the structure of testable paths in a component)

Table 1: Software metrics used with the classifier.

3. GENETIC ALGORITHM

Evolutionary computation refers to the set of methods that simulate evolution, in that a population of solutions are modified using directed random variations and selection in order to optimize the solution to a problem [4]. Genetic algorithms (GAs) are evolutionary methods that may be used to solve search and optimization problems. They are based on the process of Darwinian evolution over many generations, the "fittest" individuals tend to dominate the population.

In GA, the solution is encoded in genes. A collection of genes make up a population, and each gene is composed of a set of chromosomes. A GA evolves a population by mixing (crossover) the chromosomes of the parent genes, and mutating the individual chromosomes of each child gene. Figure 1 illustrates the basic steps in the canonical GA.

![Figure 1: The major steps of a genetic algorithm](image)

In order to determine how well a gene solves the given problem, a fitness value is determined by a problem-dependent objective function, in this case linear discriminant analysis. The genes with high fitness values are more likely be chosen for reproduction and evolve into the next generation, while genes with lower fitness values tend not to evolve and are less likely to survive into the next generation.

The number of genes in a population, the number of elite genes (genes that pass on through to the next generation based on their fitness value), the number of generations and the probability of mutations are parameters that may determines how well the GA converges to a solution. During each generation, the number of genes in the population is constant and non-elite genes are replaced by the offspring genes. The probability of mutation determines how often a chromosome should mutate or change its value.

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 particular problem, each bit
represented a metric to be used. If the bit was on, then that feature was used with the classifier. The parents to mate were chosen based on their fitness value. A gene was picked at random, and a mating probability was calculated. If the gene's fitness value was higher than the mating probability, it was chosen as a parent.

A child gene was created by randomly picking a crossover point and exchanging the corresponding bits from the two genes. The first \( k \) bits from parent one, the second group were the \( N-k \) bits from the second parent, where \( N \) is number of bits in a gene. Mutation was done by flipping the value of each bit of the child gene if the random probability was less than the input parameter. The objective function of the GA is the classification rate of the software metrics encoded in the gene using LDA. A new population is created by merging the children with the elite genes of the previous population. The elite genes encode the software metrics that outperform all the other genes in the population. The investigator, at runtime, specifies the values for the number of genes in a population, the number of generations, the percent of elite genes, and the probability of mutations.

3.1 Linear Discriminant Analysis

Linear discriminant analysis (LDA) is a conventional classifier strategy used to determine linear decision boundaries between groups while taking into account between-group and within-group variances [5]. If the error distributions for each group are the same (each group has identical covariance matrices and sampled from a normal population), it can be shown that linear discriminant analysis constructs the optimal linear decision boundary between groups. In real-world situations, this optimality is seldom achieved since different groups typically give rise to different distributions. Nevertheless, LDA is a useful classifier and will be used here to benchmark each variant.

A feature vector, \( x \), should be allocated to the group for which the probability distribution, \( p_i(x) \), is greater than any other distribution, while taking into account known prior probabilities. So,

\[
x \in \omega_i \text{ if } q_i p_i(x) \geq q_j p_j(x) (\forall j = 1,\ldots,k)
\]

(1)

where \( q_i \) is the prior probability of observing \( x \) from the group, \( \omega_i \). Here, we substitute the proportional probabilities, \( N_i/N \), for each \( \omega_i \). \( D_i(x) = L_i(x) = 0 \), defines the hyperplane in the feature space that separates \( \omega_i \) from \( \omega_j \), where the discriminant function, \( L_i(x) \), is

\[
L_i(x) = \log q_i + \mu_i^T W^{-1}(x - \frac{1}{2} \mu_i)
\]

(2)

(\( \mu_i \) is the mean for \( \omega_i \) and \( W \) is the dataset's covariance matrix). Figure 2 shows a three-class 2-dimensional classification problem and the decision hyper-planes produced by LDA. The decision rule is straightforward: if \( D_{12}(x) \) \( 0 \) and \( D_{21}(x) \) \( 0 \), then \( x \in \omega_1 \); if \( D_{12}(x) \) \( 0 \) and \( D_{31}(x) \) \( 0 \), then \( x \in \omega_2 \); otherwise, \( x \in \omega_3 \).

Figure 2: LDA decision boundaries for three groups.

4. RESULTS AND OBSERVATIONS

Using all 17 metrics for the 338 software objects to train the LDA classifier and testing it with the leave-one-out validation method gave a classification rate of 62.7%. The classification rate of the top 5 GA metrics subset, compared to using all the metrics, is illustrated in Figure 3. The last column uses the 6 metrics common to all 5 genes.

![Figure 3: Best genes classification rates.](image-url)
After 20 generations, with the following parameters, 50 genes, 25% elite genes, 10% mutation rate 39 of the genes outperformed the entire metrics set. Table 2 lists the metrics that were common to the top 5 genes. As expected, the number of source lines (the mean per method, or in the entire file) and the comments to source code ratio were metrics that seems to match the expert classification. The white space metric is likely an indication of the overall formatting of the code, which impacts how well a programmer is able to understand another programmer's code. The number of executable statements and the cyclomatic complexity also seems to be a common metric for the top genes.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLC</td>
<td>The number of lines in a file containing source code (SLC = LOC - CLC)</td>
</tr>
<tr>
<td>WLC</td>
<td>The number of lines containing white space</td>
</tr>
<tr>
<td>RCC</td>
<td>The ratio of comment lines of code to total lines of code including white space and comments (LOC)</td>
</tr>
<tr>
<td>AE</td>
<td>Average number of executable statements</td>
</tr>
<tr>
<td>ASL</td>
<td>The mean number of source lines of code within a method</td>
</tr>
<tr>
<td>ACC</td>
<td>The mean cyclomatic complexity of a method</td>
</tr>
</tbody>
</table>

Table 2: Software metrics common to top 5 genes.

The best gene resulted in a subset of 9 features and a classification rate of 68.6%. Table 3 shows the additional software metrics that gave the best results after 20 generations.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCS</td>
<td>Ratio of CLC to SLC</td>
</tr>
<tr>
<td>ANL</td>
<td>The mean string length of method names</td>
</tr>
<tr>
<td>ACN</td>
<td>Average number of control statements</td>
</tr>
</tbody>
</table>

Table 3: Additional metrics subset of the best gene.

Interestingly enough, including the mean length of a method name improves the classification rate. Again, this correlates well with a programmer's ability to extract the purpose and meaning of source code when variables and methods have meaningful names that reflect their intended usage and function, which aids in the understanding and eventual maintenance of the class. The average number of control statements, like the cyclomatic complexity of a method, reflect the relative complexity of the method and the potential flow of control paths to other methods or classes.

5. CONCLUSION

This study has shown that by using evolutionary computational strategies such as genetic algorithms for feature selection, it is possible to improve the classification rate for the quality of software objects and focus on the metrics that appear to correlate with the intuitive metrics an expert would use. The GA metrics appear to indicate that code that is easy to read (WLC, ANL), along with comments (RCC, RCS) help developers understand the purpose of the code. The apparent complexity of the method (ASL, CAN, ACC) also affect the quality of a software object in terms of maintainability.

It is important to note that none of the metrics used for this study reflect the properties of an object-oriented system, such as the depth of inheritance, the number of abstract, virtual, overloaded or overridden methods. An expert would consider these aspects of a class when deciding on the quality of the software object in terms of maintenance. The best classification success rate of 68.6% may be a reflection that the metrics used do not take into account OO programming practices and complexity. Metrics specific to Java, such as interfaces implemented and native methods were not calculated. Future research with this dataset should use Java specific object-oriented metrics, and classify other aspects of overall software quality, which would include complexity and reusability.

Acknowledgement

The Natural Sciences and Engineering Council (NSERC) is gratefully acknowledged for its support of this investigation.

References