



The Analysis of Software Complexity Using Stochastic Metric Selection

Nick J. Pizzi

*Department of Computer Science,
University of Manitoba, Winnipeg, Canada*

pizzi@cs.umanitoba.ca

Aleksander B. Demko

*Department of Computer Science,
University of Manitoba, Winnipeg, Canada*

ademko@cs.umanitoba.ca

Witold Pedrycz

*Department of Electrical & Computer Engineering,
University of Alberta, Edmonton, Canada*

pedrycz@ece.ualberta.ca

Abstract

The automated prediction of qualitative attributes such as software complexity is a desirable software engineering goal. A potential technique is to use software metrics as quantitative predictors for these kinds of attributes. We describe a pattern classification method where a large collection of classifiers is presented with randomly selected subsets of software metrics describing modules from a sophisticated biomedical data analysis system. The method identifies the software metric subset that has the highest discriminatory power vis-à-vis software complexity. That is, we identify the metric subset that is most effective at predicting this qualitative attribute. This classification method is empirically evaluated and carefully validated against three benchmark approaches. We demonstrate that this method has utility in the automated prediction of software complexity using quantitative measures.

Keywords: Software metrics, Pattern classification, Feature selection

1. Introduction

Software systems are used to solve or model increasingly sophisticated and complex problems in a variety of application domains. However, this complexity should not manifest itself as complexity within individual software module (functions, objects, templates, and so on). That is, problem complexity should only produce software complexity as an emergent property and not as an intrinsic property of individual modules. Intrinsic module complexity is an undesirable characteristic, which is often subjectively identified by software developers and managers for some form of remedial action. However, this subjective process is slow, tedious, and error-prone. Automating (to some extent) this process would be desirable especially if coupled with a more rigorous quantitative approach. One possible strategy is the prediction of module complexity using quantitative software measures [1][2][3][4][5].

If an appropriate external reference test can be identified to label the complexity of software modules, then the analysis of module complexity may be cast as a problem of classification: determine a mapping that predicts the complexity of a module, given only its metrics. More formally, $\mathbf{X} = \{(\mathbf{x}_k, \omega_k), k = 1, 2, \dots, N\}$ is a set of N patterns (software modules as described by their metrics), $\mathbf{x}_k \in \mathfrak{R}^n$, with respective class labels (complexity),

$\omega_k \in \Omega = \{1, 2, \dots, c\}$. While some features (metrics) may have discriminatory power, that is, predictive utility vis-à-vis complexity, other features may have none. Moreover, it is possible for some features to have a potential deleterious effect on the classification process [6][7][8]; these confounding features distort class boundaries in the metric space (for instance, a class boundary delineating high complexity and low complexity modules), thereby reducing classification performance. Therefore, it is important to not only identify discriminatory features but to also cull confounding ones.

To this end, a classification strategy, *stochastic feature selection* [9][10], is investigated as a system to determine the subset of software metrics that best predict software module complexity. Such a classification system could serve as a “complexity filter” for software modules. For example, a project manager or software architect could use such a filter to predict module complexity to identify highly complex modules for review and possible revision. The strategy, which stochastically examines subsets of metrics for predictive power, is tested using a database of metrics from a large medical image processing system and its classification performance is benchmarked against two standard classification approaches. We begin with a general discussion of software metrics and module complexity in section 2 and the specific metrics from the software system dataset (section 3). Section 4 describes the stochastic feature selection approach to pattern classification. We present the experiment design in section 5 followed by a discussion of the experiment results in section 6 with concluding remarks in section 7.

2. Measuring Software Attributes

2.1 Software Metrics

Commonly regarded as important factors reflecting the nature of software systems, software metrics [11][12][13][14] are mappings from software modules to sets of numerical values that quantify qualitative attributes of the modules. The software engineering literature is replete with thorough discussions of the scores of different software metric variants and their relative advantages and disadvantages [15][16]. The use of metrics requires a fundamental understanding of what is being measured and the proper interpretation of how the measurements are being acquired both of which are necessary prerequisites to software refactoring, that is, changing a software module to improve its conceptual structure without affecting its behaviour [17][18]. Software metrics may be broken down into three main types [19]: historical measures related to the conventional procedural programming paradigm; Software Science metrics introduced in the seminal paper by Halstead [20]; measures related to the object-oriented programming paradigm, many of which were introduced in another seminal paper by Chidamber and Kemerer [21].

The first type of metric includes procedural programming measures such as the: number of lines of code, tokens, or decisions, as well as their average per module (function or object); maximum decision depth; highest number of parameters defined for a single operation; ratio of the total number of comment lines to lines of code (suggestive of module quality); syntactic length of function names (a crude indicator of meaningfulness). McCabe’s cyclomatic complexity of a module is the number of possible paths in its decision flow graph [22] and is a measure of overall system complexity [23][24], informally, the number of binary decisions in a module plus one. [Assuming that a module is represented as a directed acyclic graph, a more formal definition is $e - g + 2d$ where e is the number of edges in the graph, g is the number of nodes, and d is the number of disconnected parts in the graph.] Another complexity measure is the Belady’s bandwidth metric, $g^{-1} \sum_i (i \times g_i)$, where g_i is the number

of nodes at level i in a nested control flow graph of g nodes. Large values for the last two metrics suggest high module complexity.

While some controversy exists [25] concerning the second type of software metrics, the Halstead metrics are an important set of measures routinely used in software engineering to measure complexity. Given the respective total number of operators and operands, n_1 and n_2 , and the respective total number of unique operators and operands, u_1 and u_2 , we may define the following metrics: Halstead's program length, $H_n = n_1 + n_2$; program vocabulary, $H_c = u_1 + u_2$; program volume, $H_v = H_n \times \log_2 H_c$; program difficulty, $H_d = (u_1/2)(n_2/u_2)$; program effort, $H_f = H_d H_v$; estimated program length, $H_e = u_1 \log_2 u_1 + u_2 \log_2 u_2$; and Jensen's program length $H_j = (\log_2 u_1)! + (\log_2 u_2)!$.

The third metric type is a more recent development due to the prevalence of object-oriented software design. While the notion of inheritance has been steadily replaced by interface design methods [26], it is still a relevant design concept. Measuring the depth of inheritance is useful as a deep inheritance chain may indicate increased module (object) complexity and poor maintainability. Related measures include the number of children and sibling classes and the number of implemented interfaces. Object interaction is a necessary phenomenon with the object-oriented programming paradigm; however, unwarranted interactions may lead to maintenance and usability problems. The count of the number of classes to which an object is coupled may reflect these potential issues. A related measure to the response for class metric is the number of methods available to a module (the total number of local and remote methods). Other object-oriented measures include the total number of constructors, members, inherited operations that a class overrides, and the number of operations added by a class, as well as the percentage of private, public, and protected members in a class.

2.2 Software Complexity

Maintainability, as defined by the ISO/IEC 9126 standard [27], is the set of attributes that bear on the effort needed to make specified modifications [28]. It includes the notions of stability, analyzability, changeability, and testability and involves the concept of how difficult it is to make small or incremental changes to an existing software module (to improve performance, correct faults, or adapt to a new environment) without introducing errors in logic or design. Maintainability involves the amount and quality of code comments, the overall size of the object, the number of polymorphic methods, and the overall flow of control. Usability is a concomitant qualitative attribute to maintainability and refers to the "set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users" [29]. Learnability, understandability, and operability are subjective usability criteria. Usability is an important qualitative attribute with respect to the software engineering notion of code re-use. Overarching the concepts of maintainability and usability is the notion of complexity [30]. While complexity often impacts maintainability and usability this is not necessarily the case. For instance, a boilerplate piece of graphical user interface code may be complex but its functionality may be well understood. Some software modules are by their nature complex (for example, sophisticated user interface interactions or complicated scientific or engineering algorithms) while others may be needlessly complex (for example, old code that is continuously modified or extended, which should be re-written or re-factored).

Table 1: Statistics for MIS software metrics.

Measure	Mean	Median	Skew	Range
L1	130.9	89.0	2.4	941.0
L2	104.0	68.0	2.3	690.0
C1	2819.7	1939.0	2.5	21207.0
C2	24.0	15.0	2.6	194.0
C3	974.8	679.0	3.4	9946.0
C4	1647.3	1044.5	2.2	10364.0
P1	298.3	179.0	2.3	2080.0
P2	342.7	240.3	1.8	1775.3
P3	260.0	175.9	1.9	1436.4
F1	11.9	8.0	2.3	79.0
L1	130.9	89.0	2.4	941.0

3. Dataset

3.1 Medical Imaging Systems

This investigation uses the Medical Imaging System (MIS) dataset [31] that has been used in several other studies dealing with software quality [14][32]. MIS is a commercial software system comprising approximately 400,000 lines of code written using Pascal, FORTRAN and PL/M assembly code. The dataset consists of $N = 390$ software modules (patterns). These modules were described using $p = 11$ software measures (features). As discussed in section 2.2, there are many factors involved in the determination of software complexity. The question naturally arises as to what objective criterion to use that can serve to delineate module complexity; that is, a complexity class label. Such criteria vary across software measurement datasets [19][33], and in this case the MIS dataset contains one software measurement that can serve as a module’s complexity class, its *change count*, which is the number of changes made to the module by software developers over the life of that module.

3.2 MIS Metrics

We now describe the 11 specific software metrics supplied with the MIS dataset. They include: L1, the number of lines of code; L2, the number of lines of code excluding comments; C1, the number of characters; C2, the number of comments; C3, the number of comment characters; C4, the number of code characters; P1, Halstead’s program length, H_n ; P2, estimated program length, H_e ; P3, Jensen’s program length H_j ; F1, McCabe’s cyclomatic complexity; and, F2, Belady’s bandwidth metric. Table 1 lists some relevant statistics pertaining to the software metrics found in the MIS dataset. It is interesting to note that the distributions for all software metrics are skewed to the right of their respective means suggesting that a small subset of the software modules may be “outliers” that are significantly more complex than the “average” software module.

3.3 Change Count as Class Labels

In this investigation, the change count (CC) attribute forms the basis of the pattern’s class labels. The 42 CC values found are listed in Table 2 along with the aggregate software module totals for each class label. As with the software metrics themselves, CC values are heavily skewed to the right of the mean (mean: 7.3, median: 4.0, skewness: 3.3, range: 98.0). It is unrealistic to expect reliable classification results with 42 classes and only 390 patterns (software modules). So, the change count class labels are aggregated into two qualitative classes (low complexity versus high complexity) using two different points of delineation, $CC \leq 3$ (DS3) and $CC \leq 4$ (DS4). The delineations were arbitrarily chosen, before

Table 2: Total number of occurrences for each change count.

CC	Total	CC	Total	CC	Total	CC	Total	CC	Total	CC	Total
0	51	7	17	14	5	22	2	32	1	40	2
1	63	8	15	15	5	23	2	33	1	41	1
2	44	9	9	16	7	25	1	34	1	42	2
3	34	10	7	17	2	27	2	35	1	44	1
4	28	11	12	18	1	28	1	37	1	46	1
5	21	12	7	19	4	29	2	38	2	47	2
6	19	13	5	20	2	30	4	39	1	98	1

the classification experiments were conducted, so that the two classes were well distributed (roughly an equal number of software modules in each class). In the delineation case when $CC \leq 3$, the number of software modules in the low complexity class was $N_{11} = 192$ and the number of software modules in the high complexity class was $N_{12} = 198$, respectively. In the delineation case when $CC \leq 4$, the number of software modules in the low complexity class was $N_{21} = 220$ and the number of modules in the high complexity class was $N_{22} = 170$.

4. Stochastic Feature Selection

Stochastic feature selection (SFS) is a feature selection pre-processing method used for pattern classification. SFS may be used with any homogeneous or heterogeneous set of classifiers (linear discriminant analysis, artificial neural networks, support vector machines, and so on). Essentially, SFS iteratively presents, in a highly parallelized fashion, many feature regions (contiguous subsets of pattern features) to the set of classifiers retaining the best set of classifier/region pairs. The SFS algorithm begins with parameter initialization including the minimum and maximum number of feature regions, and the minimum, a , and maximum, b , cardinality for a feature region. For a pattern (set of software measures), $\mathbf{x} = [x_1, \dots, x_n]$, a feature region is defined to be a contiguous subset of its features, $\mathbf{x}^{\alpha\beta} = [x_\alpha, \dots, x_\beta] (1 \leq \alpha \leq \beta \leq b \leq n)$. Other parameters include the fitness function to be used, P_f (see section 5.3) and the stopping criteria (accuracy threshold, P_ϵ , and the maximum number of iterations, η). The general SFS procedure is: (i) randomly select a number of feature regions and, for each region, select a random size (satisfying the above constraints); (ii) prune the features not selected in (i) from the design and validation subsets; (iii) use the design set and classifier to produce classification coefficients. These steps are repeated until the accuracy, P_c , for the current classification task exceeds P_ϵ or the number of iterations exceeds η . Finally, the best classification coefficients found during this design phase are subsequently assessed using the validation subset. It is important to note that the validation patterns are only used in this final step.

4.1 Feature Frequency Histogram

The stochastic nature of this method is normally controlled by the feature frequency histogram. During an SFS run, the performance of the current classification task (classifier paired with its feature regions) is assessed using P_f and compared against a histogram fitness threshold, $0 < P_h < P_\epsilon$. If $P_c \geq P_h$, then the feature frequency histogram is updated. For instance, if the current classifier is paired with the feature region $\mathbf{x}^{\alpha\beta}$ and $P_c \geq P_h$ then the histogram is updated at the feature indices α through β . The histogram is used to generate an *ad hoc* cumulative distribution function (cdf), and subsequent feature regions are randomly sampled using the current cdf. So, rather than each feature having an equal likelihood of being selected for a new classification task, those features that were used in

previous “successful” ($P_c \geq P_h$) tasks have a greater likelihood of being chosen. A temperature term, $t \in [0, 1]$, provides additional control over this process. If $t = 0$, the cdf is used as described but, as $t \rightarrow 1$, the randomness becomes more uniform (when $t = 1$ a strict uniform distribution is used).

4.2 Quadratic Combination of Features

SFS augments the original features with a quadratic combination of feature regions. The intention here is that if the original feature space possesses non-linear class boundaries, the new (quadratic) parameter space may possess more “linearized” class boundaries. For instance, say we have a set of two-feature two-class points (patterns), $\mathbf{x} = \{x_1, x_2\} \in [0, 1]^2$, bounded by the unit square where one class of points, ω_1 , are those within the unit circle ($x_1^2 + x_2^2 < 1$) and the other class, ω_2 , are those points outside ($x_1^2 + x_2^2 \geq 1$). These patterns are obviously separated by a circular (non-linear) class boundary. A linear classifier using, for instance, linear discriminant analysis, would perform poorly with such a dataset as no linear class boundary (line) can accurately delineate the two classes of points (patterns). However, if we create a new two-coordinate feature space by simply squaring the original features, the class boundary (in this new space) would be a line and a linear classifier will now perfectly separate the two classes of patterns.

SFS has three categories of quadratic combinations with which to augment the original features: (i) using the original feature region, $\mathbf{x}^{\alpha\beta}$; (ii) squaring the values for the selected feature region, $[x_\alpha^2, \dots, x_\beta^2]$; or (iii) using all pair-wise cross-products of features from two regions, $\mathbf{x}^{\alpha_1\beta_1}$ and $\mathbf{x}^{\alpha_2\beta_2}$, $[x_{\alpha_1}x_{\alpha_2}, \dots, x_{\beta_1}x_{\alpha_2}, \dots, x_{\alpha_1}x_{\beta_2}, \dots, x_{\beta_1}x_{\beta_2}]$. The probabilities of selecting one of these quadratic combination categories must sum to 1.0.

4.3 Scopira

SFS was implemented using Scopira [34], an open source C++ framework suitable for biomedical data analysis and visualization. Scopira provides high performance end-to-end application development features, in the form of an extensible C++ library. This library provides general programming utilities, numerical matrices and algorithms, parallelization facilities, and graphical user interface elements. Our motivation behind the design of Scopira was to satisfy the needs of three categories of users within the research community: developers; scientists/technologists; and data analysts. With the design, implementation, and validation of new data analysis software, developers typically need to incorporate legacy systems often written in interpreted languages. When this is coupled with the facts that, in a research environment, user requirements often change (sometimes radically) and that data is becoming ever more complex and voluminous, a development framework must be versatile, extensible, and exploit distributed, generic, and object oriented programming paradigms. For the scientist or technologist, data analysis tools must be intuitive with responsive interfaces that operate both effectively and efficiently. Finally, the data analyst has requirements straddling those for the developer and scientist. With an intermediate level of programming competence, they require a relatively intuitive development environment that can hide some of the low level programming details, while at the same time allowing them to easily set up and conduct numerical experiments that involve parameter tuning and high-level looping/decision constructs.

As a result of this motivation, the emphasis with Scopira has been on high performance, open source development and the ability to easily integrate other C/C++ libraries used in the data analysis. This library provides a large breadth of services that fall into the following four component categories. *Scopira Tools* provide extensive programming utili-

ties and idioms useful to all application types. This category contains a reference counted memory management system, network communication, object serialization and persistence, universally unique ids (UUIDs) and XML parsing and processing. The *Numerical Functions* all build upon the core n -dimensional *narray* concept. C++ generic programming is used to build custom, high-performance arrays of any data type and dimension. A large suite of data analysis and pattern recognition functions is also available. *Parallel Processing* is also provided, allowing algorithms to scale with available processor and cluster resources. A Scopira-based object-oriented framework, *Scopira Agents Library*, is included, which may be embedded into desktop applications allowing them to use computational clusters automatically, when detected. Unlike other parallel programming interfaces such as MPI [35] and PVM [36], Scopira’s facilities provide an object-centric strategy with support for common parallel programming patterns and approaches. Finally, a *Graphical User Interface Library* provides a collection of useful widgets including a scalable numeric matrix editor, plotters, image and viewers as well as a plug-in platform and a 3D canvas.

4.4 Parallelized Classification

SFS takes full advantage of parallel computations using the Scopira Agents Library. Given a high-performance computing cluster environment, classification tasks are distributed to slave nodes for computation. A master node coordinates the distribution of tasks, updates the feature frequency histogram, and records intermediate classification performance results. To minimize inter-process communication and maximize continuous computational loads on the processors, SFS efficiently “bundles” sets of classification tasks.

5. Experiment Design

In this section we present details on the design of the experiments we used to demonstrate the utility of our classification approach to predicting software complexity. We describe the SFS process flow for software metric selection, the two benchmark classification approaches, and how performance (prediction accuracy) will be measured. The dataset, \mathbf{X} , is randomly assigned to a design subset, \mathbf{X}^D , comprising N_D patterns, or a validation subset, \mathbf{X}^V , comprising N_V patterns ($N_D + N_V = N$). Using SFS and the benchmarks, we find a mapping, $f : \mathbf{X}^D \rightarrow \Omega$, but validate its effectiveness using \mathbf{X}^V , $f : \mathbf{X}^V \rightarrow \Omega$. To further ensure the reliability of the results of our experiments, we use b -fold validation. That is, we run each experiment b times using b different random allocations of \mathbf{X} to \mathbf{X}^D or \mathbf{X}^V . We report average results with standard deviations.

5.1 Stochastic Metric Selection

Figure 1 shows the SFS process flow for the experiments described in section 6. The \mathbf{X}^D software metrics are presented to SFS to examine the discriminatory effectiveness of pairs of classifiers and metrics subsets using the algorithm described in section 4. The best design pair is used to produce a prediction mapping validated using the \mathbf{X}^V software metrics.

5.2 Benchmarks

5.2.1 Linear Discriminant Analysis

Linear discriminant analysis (LDA) [6] is a standard classification strategy, which finds linear boundaries between c classes while accounting for between-class and within-class variances. If the error distributions for each class are the same (identical covariance matrices), it can be shown that LDA determines optimal linear class boundaries. In real-world situations, this optimality is rarely achieved since classes typically give rise to different distributions. Nevertheless, LDA is a useful classifier, when appropriate data preprocessing is applied such

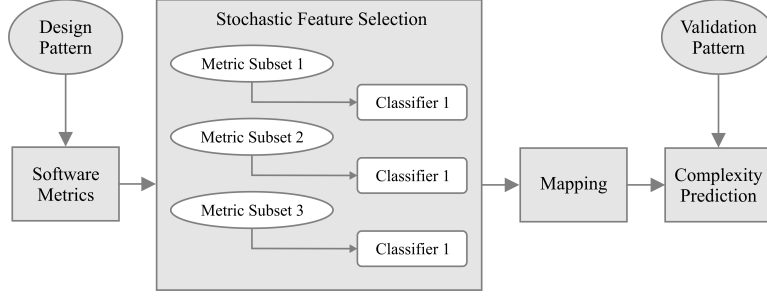


Fig. 1: Schema showing the stochastic feature selection process flow.

as SFS. LDA allocates a pattern, \mathbf{x} , to class k when $q_k p_k(\mathbf{x}) \geq q_j p_j(\mathbf{x}) (\forall j = 1, 2, 3, \dots, c)$, where $p(\mathbf{x})$ is a probability distribution and q is a proportional probability (or a prior probability, if known). The discriminant function is defined as $L_k(\mathbf{x}) = \log q_k + \mathbf{m}_k^T \mathbf{W}^{-1} (\mathbf{x} - \frac{1}{2} \mathbf{m}_k)$, where \mathbf{m}_k is the mean for class k and \mathbf{W} is the covariance matrix for \mathbf{X} . The hyperplane separating class k from class i is defined as $F_{ki}(\mathbf{x}) = L_k(\mathbf{x}) - L_i(\mathbf{x}) = 0$.

5.2.2 Support Vector Machine

The support vector machine (SVM) [38] is a family of supervised learning algorithms that select models that maximize the error margin of a training set. This approach has been successively used in many data classification problems [39]. Given a set of patterns that belong to one of two classes, an SVM finds the hyperplane leaving the largest possible fraction of patterns of the same class on the same side while maximizing the distance of either class from the hyperplane. The approach is usually formulated as a constrained optimization problem and solved using constrained quadratic programming. While the original approach could only be used for linearly separable problems, it may be extended by employing a “kernel trick” [40] that exploits the fact that a non-linear mapping of sufficiently high dimension can project the patterns to a new space in which classes can be separated by a hyperplane. In general, it cannot be determined *a priori* which kernel will contribute to producing the best classification results for a given dataset and one must rely on heuristic (trial and error) experimentation. Common kernel functions, for patterns \mathbf{x} and \mathbf{y} , are [40]: power, $K(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot \mathbf{y})^d$; polynomial, $K(\mathbf{x}, \mathbf{y}) = (a\mathbf{x} \cdot \mathbf{y} + b)^d$; sigmoid, $K(\mathbf{x}, \mathbf{y}) = \tanh(a\mathbf{x} \cdot \mathbf{y} + b)$; and Gaussian, $K(\mathbf{x}, \mathbf{y}) = \exp(-\frac{1}{2} |\mathbf{x} - \mathbf{y}|^2 / \sigma)$.

5.2.3 Typical Feature Selection

One final benchmark will involve a typical feature selection method (FSM). With this method, feature subsets are randomly sampled from the original set of features using neither a feature frequency histogram nor any kind of feature transformations. SVM is used as the underlying classifier for the FSM benchmark.

5.3 Measuring Performance

Selecting a method for measuring the performance (accuracy) of a classification system is often taken for granted, which often leads to overly optimistic results. Given a $c \times c$ confusion matrix of desired (as determined by the external reference test) versus predicted (as determined by the classification system) class labels, performance is typically measured using the observed agreement, $P_o = N^{-1} \sum_i n_{ii} (i = 1, 2, \dots, c)$, which is the ratio of samples that lie in regions associated with their corresponding classes to the total number of samples (n_{ii} are the diagonal elements of the confusion matrix). While often used to measure

Table 3: “Low” (ω_1) and “High” (ω_2) complexity class breakdown into design and validation subsets for the DS3 and DS4 datasets.

	DS3			DS4		
	Design	Validation	Total	Design	Validation	Total
Low: ω_1	120	72	192	120	100	220
High: ω_2	120	78	198	120	50	170
Total	240	150	390	240	150	390

performance, P_o does not take into account the agreement that might be due to chance, $P_r = N^{-2} \sum_i (\sum_j n_{ij} \sum_j n_{ji})(i, j = 1, 2, \dots, c)$ [41].

A more conservative performance measure is the κ score [41], a chance-corrected measure of agreement between the desired and predicted class assignments, $\kappa = (P_o - P_r)/(1 - P_r)$. If the agreement is due strictly to chance, $\kappa = 0$. If the agreement is greater than chance $\kappa > 0$; $\kappa = 1$ indicates complete agreement. If the agreement is less than chance then $\kappa < 0$ (the minimum value depends upon the marginal distributions). One may also define corresponding chance-corrected specificity (the ratio of high complexity modules that are assigned the “High” class label), κ_0 , and sensitivity (the ratio of low complexity modules that are assigned the “Low” class label), κ_1 measures using this approach [41]: $\kappa_0 = (n_{11}n_{22} - n_{12}n_{21})/((n_{11} + n_{21})(n_{21} + n_{22}))$ and $\kappa_1 = (n_{11}n_{22} - n_{12}n_{21})/((n_{11} + n_{12})(n_{12} + n_{22}))$

6. Results and Discussion

6.1 Predictive Power

Table 3 lists the breakdown of software module patterns into design and validation subsets for the respective datasets DS3 and DS4 as described in section 3.3. The following SFS parameters were used: LDA as the underlying classifier; $\eta = 10^5$; $P_e = 0.9$; $P_h = 0.4$; $t = 0.5$; 1–11 feature regions; $a = 1$, $b = 11$; 5-fold validation; $P_f = (\kappa + \kappa_0 + \kappa_1)/3$; and the quadratic metric combination breakdown was 30% for squared terms, 30% for pair-wise cross products, and 40% using the original metrics. Tables 4 and 5 list the classification results using DS3 and DS4, respectively. SFS is compared against the three benchmarks, FSM, SVM, and LDA. The tables include the median confusion matrix (desired versus predicted class labels) and the mean classification accuracies, κ , P_f , and P_o .

Table 4 lists the classification performance results for the validation subsets, using the DS3 dataset. SFS produced significantly better classification results than the SVM (using a sigmoidal kernel) and LDA benchmarks, which utilized all the metrics. SFS effected a 27% increase in the κ score (0.66 versus 0.53), a 17% increase in the P_f agreement measure (0.70 versus 0.60), and a 8% increase in the standard P_o agreement measure (0.83 versus 0.77). Note the greater improvement rate with the κ score. The larger improvements in the chance-corrected performance measures demonstrates that the classification performance increase with SFS is more significant than P_o would suggest. This is because SFS had a concomitant improvement in both the sensitivity and specificity results. Interestingly, SFS correctly classified 88% of the high complexity patterns compared to SVM, the best benchmark, with only 81%. This is a good result as one of the motivations for this classification system is to have high predictive power for problematic software modules (that is, those with high complexity). Finally, SFS demonstrated a 12% increase in the κ score compared to the other feature selection method, FSM (0.66 versus 0.59).

Table 5 lists the classification performance results for the validation subsets, using the DS4 dataset. Again, SFS produced significantly better results than the benchmarks: a 16%

Table 4: DS3 median confusion matrices of desired versus predicted (D vs P) for SFS and benchmarks including mean values for κ , P_f and P_o .

D vs P	SFS		SVM		LDA		FSM	
	Low	High	Low	High	Low	High	Low	High
Low: ω_1	57	15	51	21	51	21	53	19
High: ω_2	9	69	15	63	16	62	11	67
κ	0.66 ± 0.02		0.52 ± 0.01		0.50 ± 0.03		0.59 ± 0.02	
P_f	0.70 ± 0.02		0.60 ± 0.01		0.58 ± 0.03		0.65 ± 0.02	
P_o	0.83 ± 0.01		0.77 ± 0.01		0.75 ± 0.02		0.79 ± 0.01	

Table 5: DS4 median confusion matrices of desired versus predicted (D vs P) for SFS and benchmarks including mean values for κ , P_f and P_o .

D vs P	SFS		SVM		LDA		FSM	
	Low	High	Low	High	Low	High	Low	High
Low: ω_1	86	14	80	20	79	21	83	17
High: ω_2	14	36	15	35	18	32	15	35
κ	0.59 ± 0.01		0.51 ± 0.01		0.43 ± 0.02		0.54 ± 0.01	
P_f	0.64 ± 0.01		0.56 ± 0.01		0.52 ± 0.02		0.59 ± 0.01	
P_o	0.82 ± 0.01		0.77 ± 0.01		0.74 ± 0.01		0.78 ± 0.01	

increase in the κ score (0.59 versus 0.51), a 14% increase in the P_f (0.64 versus 0.56), and a 6% increase for P_o (0.82 versus 0.77). Finally, SFS demonstrated a 9% increase in the κ score compared to the other feature selection method, FSM (0.59 versus 0.54).

It is interesting to note that the validation set P_o values using SFS for DS3 and DS4 are similar (0.83 and 0.82, respectively) but κ is significantly different (0.65 and 0.59, respectively). This is evidenced by the poorer balance between the sensitivity and specificity in the case of DS4. Again, while P_o suggests little difference in the performance results using the different datasets, κ strongly suggests that DS3 is a better delineated dataset.

6.2 Predictive Software Measures

Figure 2 shows the feature frequency histogram plots for the DS3 and DS4 datasets. SFS experiments showing how often different software measures were used for “successful” classification tasks. In the case of DS3, two metric regions were selected as being highly discriminatory. One region used 8 original software metrics: C1, C2, C3, C4, P1, P2, P3, and F1. The other region was a pair-wise cross product of the features C1, C2, C3, and C4 with C1, C2, and C3 (recall that region overlap was permitted). In the case of DS4, two metric regions were also selected. The first used 10 original metrics excluding only F2. The other region was the squared values of all regions except F2 and F1. Both experiments demonstrate that quadratic combinations of these software metrics effect greater predictive power than any combination of the original metrics.

7. Conclusion

Stochastic feature selection has been shown to be an effective strategy for the prediction of software module complexity, as expressed by module change count, given a set of software metrics. Compared to two conventional benchmarks, this strategy produced significantly better overall classification results, as well as improved sensitivity and specificity, using only a subset of the original metrics. Finally, it was particularly effective in predicting the occurrence of high complexity modules.

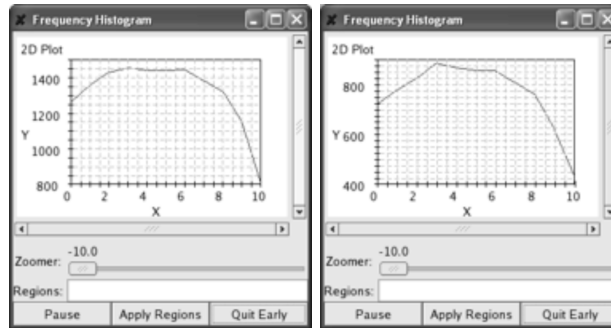


Fig. 2: SFS frequency histogram plots for the DS3 (left) and DS4 (right) experiments.

This study did not involve an *a priori* determination that these metrics were, in some sense, optimal or ideal predictors of software module complexity. It is more than likely that this set is neither wholly necessary nor sufficient for this purpose. These metrics were used for the practical reason that they were available. A thorough review of the utility of other metrics is warranted for future investigation. This could also include an examination of source code trees over time by directly accessing a software project's source control system.

The software engineering community, over the decades, has developed a number of useful and innovative metrics to quantitatively assess software systems. At the same time, software developers have accumulated significant expertise in subjectively evaluating software quality using the intuitive concepts of maintainability, usability, and complexity. No single metric can be an indicator of software quality for all types of systems, however, determining the “optimal” combination of multiple metrics is certainly a non-trivial exercise. Incorporating the expertise of the developer into a “quality filter” analysis system may be an ideal complement to finding a mapping from metrics to quality. Couching the mapping problem as one of classification is a potentially viable option, which is supported by this case study.

Acknowledgments

The authors thank M. R. Lyu for making the MIS dataset available for this investigation.

References

- [1] N. E. Fenton and M. Neil, “A critique of software defect prediction models,” *IEEE Trans. Software Engineering*, vol. 25, no. 5, pp. 675–689, 1999.
- [2] M. Reformat, W. Pedrycz, N. Pizzi, “Building a software experience factory using granular-based models,” *Fuzzy Sets and Systems*, vol. 145, no. 1, pp. 111–139, 2004.
- [3] G. Büyüközkan, C. Kahraman, and D. Ruan, “A fuzzy multi-criteria decision approach for software development strategy selection,” *Intl. J. General Systems*, vol. 33, pp. 259–280, 2004.
- [4] M. A. Sicilia, J. J. Cuadrado, J. Crespo, and E. García-Barriocanal, “Software cost estimation with fuzzy inputs,” *Kybernetika*, vol. 41, no. 2–3, pp. 249–264, 2005.
- [5] S.J. Huang, C. Y. Lin, and N. H. Chiu, “Fuzzy decision tree approach for embedding risk assessment information into software cost estimation model,” *J. Information Sci. and Eng.*, vol. 22, pp. 297–313, 2006.
- [6] G.A.F. Seber, *Multivariate Observations*, Hoboken: Wiley, 1984.
- [7] R. Gnanadesikan, *Methods for Statistical Data Analysis of Multivariate Observations* New York: Wiley, 1997.
- [8] N. J. Pizzi and W. Pedrycz, “Effective classification using feature selection and fuzzy integration,” *Fuzzy Sets and Systems*, vol. 159, pp. 2859–2872, 2008.

- [9] N. J. Pizzi, "Classification of biomedical spectra using stochastic feature selection," *Neural Network World*, vol. 15, no. 3, pp. 257–268, 2005.
- [10] N. J. Pizzi, "A computational intelligence strategy for software complexity prediction," *IEEE World Congress on Computational Intelligence*, July 16–21, Vancouver, Canada, 9477–83, 2006.
- [11] G. Poels and G. Dedene, "Distance-based software measurement: necessary and sufficient properties for software measures," *Information and Software Tech.*, vol. 42, pp. 35–46, 2000.
- [12] E. J. Weyuker, "Evaluating software complexity measures," *IEEE Trans. Software Eng.*, vol. 14, no. 9, pp. 1357–1365, 1988.
- [13] B. A. Kitchenham, R. T. Hughes, and S.G. Kinkman, "Modeling software measurement data," *IEEE Trans. Software Eng.*, vol. 27, no. 9, pp. 788–804, 2001.
- [14] J. C. Munson and T. M. Khoshgofthaar, "Software metrics for software reliability assessment," in *Handbook of Software Reliability Engineering*, M. R. Lyu, ed., New York: McGraw-Hill, pp. 493–529, 1996.
- [15] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, Boston: PWS Publishing, 1997.
- [16] L. M. Laird and M. C. Brennan, *Software Measurement and Estimation: A Practical Approach*, Hoboken: Wiley, 2006.
- [17] R. Marinescu, "Detecting design flaws via metrics in object-oriented system," *Intl. Conference and Exhibition on Technology of Object-Oriented Languages and Systems, Santa Barbara, USA*, July 29 – August 3, pp. 173–182, 2001.
- [18] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Reading: Addison-Wesley, 1999.
- [19] N. J. Pizzi, "Software quality prediction using fuzzy integration: a case study," *Software Computing Journal*, vol. 12 no. 1, pp. 67–76, 2008.
- [20] M. H. Halstead, *Elements of Software Science*, New York: Elsevier, 1977.
- [21] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object-oriented design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476–493, 1994.
- [22] T. J. McCabe, "A complexity metric," *IEEE Trans. Software Eng.*, vol. 2, pp. 308–320, 1976.
- [23] B. Curtis, S. Sheppard, and P. Milliman, "Third time charm: stronger prediction of programmer performance by software complexity metrics," *Proc. IEEE Intl. Conference on Software Eng.*, Munich, Germany, September 17–19, pp. 356–360, 1979.
- [24] J. Ward, "Software defect prevention using McCabe's complexity metric," *Hewlett-Packard Journal*, vol. 2, pp. 66–69, 1989.
- [25] C. Jones, "Software metrics: good, bad, and missing," *Computer*, vol. 27, pp. 98–100, 1994.
- [26] P. Coad, M. Mayfield, and J. Kern, *Java Design: Building Better Apps & Applets*, Upper Saddle River: Prentice Hall, 1999.
- [27] H.-W. Jung, S.-G. Kim, and C.-S. Chung, "Measuring software product quality: a survey of ISO/IEC 9126," *IEEE Software*, vol. 21, no. 5, pp. 88–92, 2004.
- [28] S. Pressman and R. Pressman, *Software Engineering*, New York: McGraw Hill, 2000.
- [29] N.E. Fenton and A.A. Kaposi, "Metrics and software structure," *Information and Software Tech.*, vol. 29, pp. 301–320, 1987.
- [30] D. Card and R. Glass, *Measuring Software Design Quality*, Englewood Cliffs: Prentice-Hall, 1990.
- [31] M. R. Lyu, <http://www.cse.cuhk.edu.hk/lyu/book/reliability/>, Last accessed 2010.11.24.
- [32] R. Lind and K. Vairavan, "An experimental investigation of software metrics and their relationship to software development effort," *IEEE Tran. Software Eng.*, vol. 15, no. 5, pp. 649–653, 1989.
- [33] M. Reformat, W. Pedrycz, and N. J. Pizzi, "Software quality analysis with the use of computational intelligence," *Information and Software Tech.*, vol. 45, pp. 405–417, 2003.
- [34] A. B. Demko and N. J. Pizzi, "Scopira: An open source C++ framework for biomedical data analysis applications," *Software-Practice and Experience*, vol. 39, no. 6, pp. 641–660, 2009.
- [35] M. Snir and W. Gropp, *MPI: The Complete Reference*, Cambridge: MIT Press, 1998.
- [36] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. S. Sunderam, *PVM: Parallel Virtual Machine*, Cambridge: MIT Press, 1994.
- [37] F. S. Hill and S. M. Kelley, *Computer Graphics Using OpenGL*, Englewood Cliff: Prentice-Hall, 2006.

- [38] V. Vapnik, *Statistical Learning Theory*, New York: Wiley, 1998.
- [39] L. Wang, *Support Vector Machines: Theory and Applications*, Berlin: Springer-Verlag, 2005.
- [40] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, Cambridge: Cambridge University Press, 2007.
- [41] B. S. Everitt, "Moments of the statistics kappa and weighted kappa," *British Journal of Mathematical and Statistical Psychology*, vol. 21, 97–103, 1968.