A fuzzy classifier approach to estimating software quality

Nick J. Pizzi *

Department of Computer Science, University of Manitoba, Winnipeg MB, Canada R3T 2N2

A R T I C L E  I N F O

Article history:
Received 2 June 2010
Received in revised form 5 May 2011
Accepted 19 April 2013
Available online 25 April 2013

Keywords:
Computational intelligence
Fuzzy logic
Pattern classification
Software metric
Software engineering

A B S T R A C T

With the increasing sophistication of today's software systems, it is often difficult to estimate the overall quality of underlying software components with respect to attributes such as complexity, utility, and extensibility. Many metrics exist in the software engineering literature that attempt to quantify, with varying levels of accuracy, a large swath of qualitative attributes. However, the overall quality of a software object may manifest itself in ways that the simple interpretation of metrics fails to identify. A better strategy is to determine the best, possibly non-linear, subset of many software metrics for accurately estimating software quality. This strategy may be couched in terms of a problem of classification, that is, determine a mapping from a set of software metrics to a set of class labels representing software quality.

We implement this strategy using a fuzzy classification approach. The software metrics are automatically computed and presented as features (input) to a classifier, while the class labels (output) are assigned via an expert's (software architect) thorough assessment of the quality of individual software objects. A large collection of classifiers is presented with subsets of the software metric features. Subsets are selected stochastically using a fuzzy logic based sampling method. The classifiers then predict the quality, specifically the class label, of each software object. Fuzzy integration is applied to the results from the most accurate individual classifiers. We empirically evaluate this approach using software objects from a sophisticated algorithm development framework used to develop biomedical data analysis systems. We demonstrate that the sampling method attenuates the effects of confounding features, and the aggregated classification results using fuzzy integration are superior to the predictions from the respective best individual classifiers.

© 2013 Elsevier Inc. All rights reserved.

1. Introduction

Effective pattern classification requires the coupling of an efficient classifier with synergistic pre- and post-processing strategies. It is often the case with problems of classification that only a subset of features possesses discriminatory power while the remainder has a tendency to confound the effectiveness of the underlying classifier. In such cases, a sensible pre-processing strategy is to identify and select the discriminatory features and prune the confounding ones. Moreover, an effective post-processing strategy is to aggregate the predictions from a set of classifiers to produce a consensus set of predictions that is anticipated to be superior to any of the individual classifier predictions. To this end, we present a fuzzy classification system, fuzzy feature selection with prediction aggregation (FSPA), that identifies discriminatory feature subsets (using a fuzzy logic based sampling rule) that are presented to a collection of classifiers where the best classifier predictions are aggregated...
(using fuzzy integration) to produce a consensus set of predictions. To validate this system, we study a classification problem from the area of quantitative software engineering.

Software systems are utilized to solve or model increasingly sophisticated and complex problems in a variety of application domains. Given the complexity of many of these contemporary systems, it is often difficult to gauge the complexity, utility, and extensibility of underlying software objects. A possible strategy to evaluate the qualitative attributes of a system’s objects is to use software metrics as quantitative predictors. Examples of software metrics include the coupling between software objects, the number of lines of code, the number of unique operators and operands, data abstraction coupling, and the ratio of lines of comments to lines of code, to name but a few. If an appropriate external reference test can be identified to label the quality of these software objects then this strategy may be viewed as a problem of classification. That is, predict an object’s qualitative attributes (based on the reference test) using a set of software metrics that prediction. While the main advantage of this strategy is that software metrics may be automatically computed, identifying those software metrics that are the best quality predictors is problematic.

We empirically evaluate our fuzzy classification system using software objects from a sophisticated algorithm development framework. We compare the classification results using the feature subsets found by the fuzzy sampling rule against the predictions using the entire feature set. This sampling rule is an extension to a feature selection algorithm described in prior work [41], in which a user-defined threshold was used to assess the classification performance of feature subsets. Here, we replace this static threshold with a fuzzy update rule that takes into account the relative and evolving performance of feature subsets. We also compare the aggregation approach using fuzzy integration against a majority vote rule. We demonstrate that the sampling method attenuates the effects of confounding features, and the aggregated classification results using fuzzy integration are superior to the predictions from the respective best individual classifiers. We begin with a description of the fuzzy classification system and the fuzzy sampling rule followed by a discussion on software metrics, in general, and the metrics used in the experiments. We continue with a brief presentation of the candidate software system from which the experiment’s features are generated and a description of the experiment design. Finally, we discuss the experiment results, compare these results with a benchmark classification approach, and present concluding remarks.

2. Fuzzy feature selection

Consider a pattern classification problem in which \( X = \{ (\mathbf{x}_k, w_k), k = 1 \ldots N \} \) is a set of \( N \) labeled patterns. Here, \( \mathbf{x}_k \in \mathbb{R}^m \) (\( m \) features or dimensions) and \( w_k \in \Omega = \{ 1 \ldots c \} \). A classifier may be viewed as a mapping, \( f: X \rightarrow \Omega \). Let \( w_k \) be the class label predicted by a classifier for pattern \( \mathbf{x}_k \). If \( w_k = w_k \), the classifier generated a correct classification result (prediction) for \( \mathbf{x}_k \). Formally, feature selection involves finding a mapping \( f^*: X \rightarrow X' \subseteq \mathbb{R}^n \) in the reduced feature space. Classification involves the subsequent determination of a mapping from the reduced feature space to the space of class labels, \( f^*: X' \rightarrow \Omega \).

The motivation for pre-processing strategies exploiting feature selection [55,29,26] is to simplify the determination and construction of optimal class boundaries in the feature space. Many feature selection preprocessing methods exist in the literature. For instance, [18] describes the identification of relevant input variables through a constrained scaling strategy for a support vector machine [57] used in a facial expression recognition task. A sequential search method is presented in [44], which iteratively examines varying numbers of features, and is shown to be more computationally efficient than a branch and bound method. Using a set of multispectral images in a prostate cancer classification task, [54] empirically demonstrates the effectiveness of feature selection using a hybrid classifier approach combining the K-nearest neighbour algorithm [9] with a search heuristic based on Tabu search [16]. In one final example, [46] empirically evaluates, using data from an industrial fabric textile domain, a feature extraction strategy employing Choquet integration [35] within a fuzzy rule classifier, and demonstrates that by extracting relevant features the total number of fuzzy rules generated is reduced.

FSPA is a dimensionality reduction technique used in pattern classification that may be used with any homogeneous or heterogeneous collection of classifiers. Essentially, FSPA iteratively presents many feature regions (contiguous subsets of pattern features) to a collection of classifiers, ultimately retaining the best \( q \) classifier/region pairs with respect to classification accuracy. Subsets are selected stochastically using a fuzzy logic based sampling rule. Fuzzy integration (see Section 2.2) is applied to the \( q \) results to produce a final aggregated classification outcome. Below is the step breakdown of the FSPA algorithm.

**FSPA algorithm**

1. Select classifier types and performance function.
2. Initialize relevant parameters.
3. Repeat:
   a. Select feature regions by sampling feature performance histogram.
   b. Perform quadratic transformation of feature regions.
   c. Instantiate new classifier instance.
   d. Use performance function to assess classification accuracy.
   e. Adjust feature performance histogram using fuzzy logic update rule.
   f. Update best classifier/regions pairs, if necessary.
4. Until: Performance threshold exceeded.
5. Aggregate best \( q \) classification results using fuzzy integration.
Initially, the user selects: one or more classifiers (setting relevant parameters); the number of classifier/region pairs to retain, \( q \); the minimum and maximum number of feature regions; the minimum, \( a \), and maximum, \( b \), region cardinality; the quadratic combination probabilities (see below); the performance function, \( P \); and the stopping criteria (the maximum number of iterations, \( t \), and the performance threshold, \( P_0 \)). For a pattern, \( x = [x_1, \ldots, x_n] \), a feature region is defined as a contiguous subset of its features, \( x^a = [x_a, x_{a+1}, \ldots, x_b] \) \((1 \leq a \leq \leq b \leq n)\). Steps (3–4) iterate until one of the stopping criteria is satisfied: a subset of feature regions is selected from the original feature space (see below); the feature regions are quadratically transformed; a new classifier instance is instantiated and its performance using the feature regions is assessed; the histogram is updated (see below); and the list of best classifier/region pairs is updated, if necessary. Finally, the best \( q \) results are aggregated using fuzzy integration.

FSPA exploits the quadratic combination of (disjoint or overlapping) feature regions. The intent is that if the original feature space has non-linear boundaries between classes, the new (quadratic) parameter space may have boundaries that are more linear. FSPA has three categories of quadratic combinations: using the original feature region; squaring the feature values for the region; or using all pair-wise feature cross products from two regions.

\[ P \in [0, 1], \text{which is a measure of classification accuracy, is computed using the} \ c \cdot c \ 	ext{confusion matrix,} \ R, \text{of desired versus predicted class labels. In this investigation, we use the conventional} \ P = N^{-1}\sum R(i = 1 \ldots c), \text{the ratio of correctly classified patterns to total number of patterns. However, any suitable measure may be used with FSPA such as the average classification accuracy for each class or a chance-corrected measure of agreement [11]. Clearly, low values for} P \text{indicate low classification performance (accuracy) and high values indicate high performance, with 1 indicating perfect accuracy.}

2.1. Fuzzy histogram update rule

An important component of the FSPA algorithm is the feature performance histogram, represented as the vector \( e = [e_1 \ldots e_n] (e_i \in [0, 1]) \), which is used to generate an ad hoc cumulative distribution function that is subsequently used to stochastically sample new subsets of feature regions. The histogram reflects the current classification "performance" of each individual feature. That is, if a feature, \( x_i \), is regularly present in feature regions that contribute to high (low) values of \( P \), then \( e_i \) will also be high (low). Instead of uniformly sampling from the set of original features when selecting subsets of feature regions, feature sampling is based on past “success” (high \( e \)) values.

The fuzzy logic update rule for the histogram is as follows. Say, \( e \) is the current feature performance histogram and \( P_C \) (recall \( P \in [0, 1] \)) is the performance (accuracy) value for the current classification iteration. The updated histogram, \( e' \), is computed as

\[
e'_i = \begin{cases} h_1(P_C) \lor e_i & P_C \geq e_i, P_C \geq \varepsilon \\ h_2(P_C) \land e_i & P_C < e_i, P_C > \varepsilon \quad (i = 1 \ldots n) \\ e_i & P \leq \varepsilon \end{cases}
\]

where \( \land \) is the t-norm (min), \( \lor \) is an s-norm (max), \( h_1(x) = x^2 \) is the concentration operator, \( h_2(x) = x^{1/2} \) is the dilation operator, and \( \varepsilon \in [0, 1] \) is a threshold below which no adjustment is made to the corresponding histogram element. There are four intuitive candidates for the threshold: (i) \( \varepsilon = 0.0 \), update \( e_i \) at every classification iteration (step 3.e of the FSPA algorithm) regardless of \( P_C \); (ii) \( \varepsilon = 0.5 \), update \( e_i \) only when \( P_C \) is above chance; (iii) \( \varepsilon = P_A \), update \( e_i \) only when \( P_C \) is above the current average classification accuracy, \( P_A \); (iv) \( \varepsilon = P_B \), update \( e_i \) only when \( P_C \) is above the current best classification accuracy, \( P_B \). The rationale behind (1) is that features within selected regions for which a classifier instance produced high classification accuracy (large \( P \)) should be sampled more often than those features that contribute to classifier instances producing poor classification results (small \( P \)). The threshold, \( \varepsilon \), controls how often the histogram values fluctuate: as \( \varepsilon \rightarrow 0 \), features will be updated often regardless of classification accuracy; as \( \varepsilon \rightarrow 1 \), features will be rarely updated and only when the classification accuracy is high. This fuzzy histogram update rule replaces the static user-defined performance threshold described in [41] and better takes into account the relative and evolving performance of the features.

Before the classification iterations begin, \( e \) must be initialized to a reasonable set of values; we use \( e_i = 0.05 \) (\( i = 1 \ldots n \)). Intuitively, the initial values need to be the same, as there is no prior information about classification performance, so all features should have equal likelihood of being selected. Also, the initial values should be small, but not 0, so that any feature has some likelihood, however small, to be selected during a classification iteration.

2.2. Fuzzy integration

Fuzzy integration [17] is a nonlinear numerical approach to combining multiple sources of information to arrive at a "confidence value" for a decision (hypothesis). The fuzzy integral of the mapping \( h: X \rightarrow [0, 1] \) over a finite ordered universe of discourse, \( X = [x_1, \ldots, x_n] \) with respect to the Sugeno fuzzy measure [51] (a set function used to express the grade of fuzziness) is:

\[
\int h(x) \circ g = \bigvee_i [h(x_i) \land g(X_i)]
\]

where \( X_i = [x_1, \ldots, x_i] \) are descending ordered with respect to \( h(x_i) (h(x_0) = 0) \), \( \land \) is the product t-norm \( (x_i \cdot x_j) \) and \( \lor \) is the probabilistic sum s-norm \( (x_i + x_j - x_i \cdot x_j) \).
Based on prior experience [40], and in the interests of brevity, we restrict our analysis to $h(x) = x^2$ (concentration operator) and the Sugeno integral [36],

$$S(x) = \gamma \[ h(x_i) \land g(X_i) \].$$

(3)

In this investigation, $x$ is the classifier’s predicted class label. Using (3), the final (aggregated) predicted class label from the set of classifiers is the one with the highest integrated value.

While the Sugeno integral is used here, any fuzzy integral, such as Choquet or Shilkret, may be used [7, 34]. Furthermore, although we restrict ourselves to the concentration operator, there are many other options for $h$ (dilation, identity, contrast intensification [60], and so on). Finally, while several interpretations exist for the meaning a fuzzy integral [52, 53], here it is considered to mean the maximum degree of belief (for a predicted class label) obtained by the fusion of several sources of objective evidence (credibility).

Fuzzy aggregation has been used in various quantitative software engineering problems: the evaluation of software engineering tools [3]; improving the quality of decision-making in the software development project under uncertain conditions [1]; identifying software quality requirements [37]; software cost estimation [49]; assessment of maintainability [2]; embedding risk assessment information into software cost estimation [23]; selecting software configuration items [58], and quantifying software complexity [38].

2.3. Prediction aggregation

Integrating the results from multiple classifiers involves using their respective confusion matrices to compute the fuzzy densities for each of the classifiers. To this end, the technique described in [5] will be used, for which we have had success with prior problems of classification [41]. Let $R_k = (n_{kd})$ be the $D \times D$ confusion matrix for classifier $k$ where $n_{kdd}$ is the number of class $d$ software objects that were correctly classified by $k$ and $n_{kd}(d \neq i)$ is the number of class $d$ objects that were incorrectly assigned to class $i$ by $k$. The fuzzy density of $d$ with respect to $k, 0 < g_{kd} < 1$, is

$$g_{kd} = n_{kd} \left( \sum_{d=1}^{D} n_{kd} \right)^{-1} \times W_{kd} \times A_{kd}$$

where $W$ is an adjustment (to the first term) involving the frequency of incorrect classifications within the confusion matrix, $R_k$, and $A$ is an adjustment involving the frequency of incorrect classifications across all classifiers with respect to classifier $k$ (cf. [5] for details).

3. Measuring software attributes

A software metric [27, 6] is a mapping from a software object to a set of numerical values in order to quantify one or more qualitative attributes [12]. Metrics are commonly regarded as important factors reflecting the nature of software systems including, among other characteristics, the qualitative properties of complexity, utility, and extensibility [42, 45, 59]. Using the metrics described in the software engineering literature [13, 30, 39, 43] 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 object to improve its conceptual structure without affecting its behaviour [14, 31]. Software metrics may be used by developers to identify “high risk” software objects: that is, objects that are too complex to understand and test, exhibit poor usability for clients, or are not sufficiently reusable. Metrics can be a valuable tool for the project manager and software engineer in their efforts to produce robust, well-implemented systems. Maintainability, as defined by the ISO/IEC 9126 standard [25], is the set of attributes that bear on the effort needed to make specified modifications. 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 component (to improve its 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. Utility 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” [25]. Overarching the concepts of maintainability and usability is the notion of complexity [4]. Complex components are *inter alia* more difficult to maintain and less usable. 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 components 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).

3.1. Software metrics used in experiments

We now present brief descriptions of the software metrics used in the experiments described in Section 5. We used Borland’s Together® software package [19, 56], to automatically compute the metrics described below for each object of
the candidate software system (see Section 4). We continue to use the notation, $x_n$, to index the software metrics in the order in which they appear in the dataset.

Several metrics are used to represent the conventional procedural programming paradigm. These include: the number of lines of code ($x_{17}$); the comment ratio ($x_8$), which is the ratio of comments to lines of code; the true comment ratio ($x_{45}$), which is the ratio of comments to non-comments; the total number of operands, $x_{35}$, and operators, $x_{36}$; the total number of unique operands, $x_{38}$, and operators, $x_{39}$; the maximum number of parameters defined for a single operation, $x_{25}$; the maximum "size" (see cyclostatic complexity below) of operations in an object, $x_{26}$; the maximum number of levels (nesting depth of decision/looping constructs), $x_{24}$; the number of attributes, $x_{27}$; and fan out ($x_{11}$), which is the number of reference types that are used in attribute declarations. The cyclostatic complexity ($x_6$) of an object is the number of possible paths in its decision flow graph (informally, one more than the number of binary decisions in an object). Assuming that an object is represented as a directed acyclic graph, a more formal definition for cyclostatic complexity is the difference of the number of edges and the number of nodes plus twice the number of disconnected parts in the graph [33]. As this measure increases, the object’s complexity increases.

With the prevalence of object-oriented software design, various object related metrics have been devised, many of which were introduced in [6]. The depth of inheritance hierarchy ($x_{10}$) measures the depth of an object’s inheritance; high values suggest highly specialized objects (while the notion of inheritance has been steadily replaced by interface design methods [8], it is still a relevant design concept). Attribute complexity ($x_1$) is the sum of each attribute’s value in a software object. Coupling between objects ($x_4$) is the count of reference types that are used in attribute declarations, formal parameters, return types, etc., and represents the number of other objects to which an object is coupled. A decrease in the modularity of an object can be expected with high $x_1$. However, some objects have a necessarily high degree of coupling with other objects (for instance, modern user interface frameworks). The change dependency between classes ($x_5$) is the number of methods in an object inheriting from another object, while data abstraction coupling ($x_9$) refers to the number of reference types used in attribute declarations. The response for class ($x_{44}$) is the number of methods that can be invoked by other objects and the method invocation coupling ($x_{22}$) measures the relative number of other objects to which an object sends messages. Violations of Demeter’s Law ($x_{40}$) is the number of “acquaintance” objects for a given object [28]. Large values for this group of metrics indicate potentially excessive coupling, which is detrimental to modular design and prevents software reuse.

There are three software metrics that measure the lack of method cohesion; informally, the dissimilarity of methods in an object. The first metric, $x_{18}$, assumes that $I_j$ is the set of instance variables used by method $j$ with the sets $P = \{ I_i \mid I_i \cap I_j = \emptyset \}$ and $Q = \{ I_i \mid I_i \cap I_j = \emptyset \}$. If $|P| > |Q|$, $x_{18} = |P| - |Q|$, otherwise, $x_{18} = 0$. $x_{18}$ is the percentage of methods that do not access a particular attribute averaged over all attributes in an object. $x_{30} = \left( \frac{\sum I_j - r}{s - sr} \right)$, where $r$ is the number of methods, $s$ is the number of attributes, and $r_j$ is the number of methods that access attribute $j$ measures the dissimilarity of methods in an object by attributes [21]. For $x_{18}$ and $x_{19}$ ($x_{20}$), low (high) values indicate high coupling between methods, which suggests high testing effort because many methods can affect the same attributes. This also indicates potentially low reusability.

Other object-oriented software metrics include: the number of added methods by an object, $x_{28}$; the number of objects, $x_{25}$; the number of child objects, $x_{30}$ (large values usually suggest the improper abstraction of an object); the number of constructors, $x_{31}$; the attribute hiding factor, $x_5$, ratio of visible attributes to total attributes; the attribute inheritance factor, $x_3$, ratio of inherited attributes to the number of available attributes; the coupling factor, $x_7$, the ratio of non-inheritance couplings to the maximum possible number of couplings; the method hiding factor, $x_{21}$, the ratio of total objects not visible to each method to the total number of the; the method inheritance factor, $x_{23}$, the ratio of inherited methods to the total number of available methods; the polymorphism factor, $x_{40}$, the ratio of overriding methods to the product of the total number of new methods and the number of descendants; the number of members, $x_{32}$; the number of operations, $x_{33}$; the number of overridden methods, $x_{34}$ (high values may indicate design issues since it is often preferably to extend rather than override parent class functionality; the number of remote methods, $x_{37}$; the percentage of private members, $x_{41}$; the percentage of protected members, $x_{42}$; the percentage of public members, $x_{43}$; and weighting the number of methods (or the number of methods and parameters) by the object’s cyclostatic complexity, $x_{51}$ and $x_{52}$, respectively; and four metrics that quantify the reuse of methods across class-subclass relationships [32,56], total reuse from ancestors percentage ($x_{46}$), total reuse from ancestors unitary ($x_{47}$), total reuse in descendants percentage ($x_{48}$), and total reuse in descendants unitary ($x_{49}$).

The second type comprises the Software Science metrics introduced in the seminal paper by Halstead [20]. While some controversy exists concerning their utility [24], the Halstead software metrics are an important set of measures routinely used in software engineering to measure complexity. The Halstead program length is $x_{14} = x_{35} + x_{36}$ and program vocabulary is $x_{15} = x_{38} \cdot x_{39}$. The program volume is defined as $x_{16} = \text{H}_n \cdot \log_2 x_{15}$, while program difficulty is defined as $x_{12} = \frac{\log_2 (x_{39}/2)}{(x_{35}/x_{38})}$.

4. Candidate software for evaluation

In this section, we present a brief discussion of the candidate software, Scopira, that is used to find values for the software metrics, described in Section 3.1, for each of its software objects. We then discuss the quality class labels for each of these software objects and the assessment process that led to the respective assignment.
4.1. Scopira

Scopira [10,47], an open source framework suitable for biomedical data analysis and visualization, provides high performance end-to-end application development features in the form of an extensible C++ library. This library provides general programming utilities, scientific computing algorithms, parallelization facilities, and graphical user interface elements. The initial driving force for this framework was to develop a comprehensive, object-oriented programming architecture using C++ for the development of applications relating to confirmatory and exploratory biomedical data analysis. Scopira was designed to satisfy the needs of three types of users within the biomedical research community: developers; scientists/technologists; and data analysts. With the design, implementation, and validation of new biomedical 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 biomedical 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 biomedical scientist or technologist, data analysis tools must be intuitive with responsive interfaces that operate both effectively and efficiently. Finally, the biomedical 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 biomedical data analysis field by providing a common OOP API for applications. This library provides a large breadth of services that fall into the following four component categories: extensive programming utilities such as reference counted memory management, flexible/redirectable flow input/output, object serialization and persistence, and XML parsing and processing; Scopira's own generic n-dimensional array concept used to build custom, high-performance arrays of any data type and dimension; multiple APIs for fast object-centred parallel/distributed processing including easy integration with native operating system threads, MPI [50] and PVM [15] libraries; and an extensive graphical user interface library including plotters, viewers, as well as a 3D canvas based on OpenGL" [22].

4.2. Quality assessment by architect

A software architect was asked to carefully scrutinize the software objects comprising Scopira. Based on the architect's expert judgment, each object was assigned to one of three classes: low, medium, or high. The expert considered low quality objects inferior, either in design or implementation; these objects needed to be reviewed and possibly rewritten. The expert considered high quality objects as well designed and well implemented; these objects struck an excellent balance between functionality, performance, complexity, and developer ease of use. All other objects were assigned a medium quality class label; these objects were adequately designed and implemented, but not as well as high quality objects, and did not require extensive review or revision, unlike the low quality objects. The expert's subjective assessment was based on a number of software engineering considerations including: clarity of purpose; documentation adequacy; error handling and reporting; overall complexity with respect to distribution of methods and attributes (public, protected, private), base classes, and accessors (this is an overall subjective evaluation and not the objective use of metrics described in Section 3.1); the use or extension of objects by other objects; the presence of stub methods (while useful for extensibility, this may indicate poor use of overriding; use of external libraries; meaningful object, method, and attribute names; number of developers (found in comments) and their level of expertise; use of enumeration, typedef, and numeric constants; use of developer tags such as "todo", "future", and "implement"; platform workarounds (e.g., "#ifndef irix ..."); use of deprecated code; the use of volatile, register, or mutable members; use of system calls or system environment variables; use of embedded objects; presence of code that has been commented out; the use of objects that instantiate common software design patterns; the use of objects in global or anonymous namespaces; the use of the Standard Template Library; and the use of templates. After this thorough assessment, the expert assigned each object to one of the high, medium, or low classes.

5. Experiment design

Any homogeneous or heterogeneous collection of classifier types may be used with FSPA. However, our primary interest in this investigation is the effectiveness of the fuzzy feature selection (pre-processing) and fuzzy prediction aggregation (post-processing), rather than performance analysis of a suite of classifier types. Hence, we constrain the experiments to only one classifier type, namely, linear discriminant analysis (LDA) [48]. LDA determines linear boundaries between \( c \) classes while taking into account between-class and within-class variances. LDA allocates a pattern, \( x \), to class \( i \) for which the probability distribution, \( \gamma(x) \), is greatest; that is, \( x \) is allocated to class \( i \) if \( q_i \gamma_i(x) \geq q_j \gamma_j(x) \) \((j = 1 \ldots c)\), where \( q \) are the proportional probabilities. The discriminant function is

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

where \( \mu_i \) is the class \( i \) mean and \( W \) is the covariance matrix. The feature space hyperplane separating classes \( i \) and \( j \) is defined by \( F_{ij}(x) = L_i(x) - L_j(x) = 0 \).
We also initialize the following parameters. The performance function, $P$, will be restricted to the standard performance function, the ratio of correctly classified patterns to total number of patterns (see Section 2). The stopping criteria will be $i = 1000$ and $P_S = 0.95$. We retain the best $q = 5$ classifier/region pairs and the quadratic combination probabilities will be set to 0.4, 0.3, and 0.3, respectively (see Section 2).

Finally, we compare the experiment results using the proposed method with one of the feature selection classification approaches described in Section 2. Namely, we implemented the hybridized feature selection approach that uses K-nearest neighbor classification with a Tabu based search heuristic [54].

Before proceeding to the results using the software metrics, we present a pedagogical example to demonstrate the utility of the fuzzy feature selection component of FSPA. We stochastically generated a dataset of $N = 2000$ patterns with $n = 100$ features (dimensions). All of the patterns are bounded by a hypercube, $x_i \in [-1,1]^n$, while half of the patterns have the additional property that features, $x_i (i = 21 \ldots 25, 76 \ldots 80)$ are bounded by a 10-dimensional hypersphere, $\sum x_i^2 \leq 1$. The patterns are assigned to one of two classes: those with the additional hypersphere constraint (IN) and those without (OUT). If we ignore FSPA for a moment, Table 1a shows the classification performance using only LDA on the entire set of features. The poor performance, $P = 0.58$, is to be expected as the boundary for the IN class is non-linear (a hypersphere) and, as a result, a hyperplane cannot adequately separate the two classes.

Table 1b shows the classification performance with fuzzy feature selection (using 2–5 regions with a cardinality of 2–10). The significant improvement in classification accuracy is due to two reasons. First, the best classifier/region pair used two feature regions, $x_{20} \ldots x_{25}$ and $x_{74} \ldots x_{79}$; FSPA identified all of the discriminatory features, except $x_{80}$, with only one confounding feature, $x_{20}$. Second, FSPA used the second category of quadratic feature combination, namely, the feature values were squared before presentation to LDA. Interestingly, this new quadratic feature space allowed LDA to find a hyperplane that well separated the two classes of patterns. Finally, Fig. 1 shows the feature performance histogram for FSPA using this synthetic dataset with the discriminatory features highlighted in grey.

### Table 1

Synthetic dataset classification performance using (a) linear discriminant analysis and (b) fuzzy feature selection (CA is the class-wise accuracy).

<table>
<thead>
<tr>
<th></th>
<th>a. $P = 0.58$</th>
<th></th>
<th>b. $P = 0.95$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IN</td>
<td>OUT</td>
<td>CA</td>
</tr>
<tr>
<td>IN</td>
<td>576</td>
<td>424</td>
<td>0.58</td>
</tr>
<tr>
<td>OUT</td>
<td>419</td>
<td>581</td>
<td>0.58</td>
</tr>
</tbody>
</table>

![Fig. 1. The feature performance histogram for the synthetic dataset.](image)

6. Results and discussion

The software metrics dataset contains $n = 52$ features (software metrics) describing $N = 714$ patterns (software objects). As previously mentioned, the software objects were assigned, by the software architect, to one of three class labels: low (L), 105 objects; medium (M), 248 objects; or high (H), 361 objects. We used the following parameter initialization values: LDA will be the only classifier type; $P$ will be the ratio of correctly classified objects to total number of objects; $i = 1000$ and $P_S = 0.99$ are the stopping criteria; $q = 5$; 0.4, 0.3, and 0.3 are the respective quadratic combination probabilities; 1–4 regions; and a cardinality of 2–12 for each region. We use leave-one-out for validation purposes; that is, we execute FSPA $N$ times using $N – 1$ objects for the design set and the remaining object for the test set. Table 2 shows the confusion matrix, and overall performance ($P = 0.67$) using the LDA standard and all of the software metrics. If we examine the class-wise accuracies (CA), we see that 78% of the high quality objects were correctly classified, while only 52% of the low quality objects were classified correctly. This is not a good result as it is preferable to correctly identify as many low quality objects as possible, as it is these objects that software developers and project managers need to identify accurately in order to review and pos-
(0.75 and 0.74 versus 0.76) but still significantly better than (the performance histogram is updated only when is greater than the best current accuracy).

Classification accuracy using FSPA with four different performance thresholds: (a) \( \sum = P_b \), (b) \( \sum = P_a \), (c) \( \sum = 0.5 \), and (d) \( \sum = 0.0 \) (CA is the class-wise accuracy).

<table>
<thead>
<tr>
<th></th>
<th>L</th>
<th>M</th>
<th>H</th>
<th>CA</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P = 0.67 )</td>
<td>55</td>
<td>26</td>
<td>24</td>
<td>0.52</td>
</tr>
<tr>
<td>( P = 0.71 )</td>
<td>69</td>
<td>19</td>
<td>17</td>
<td>0.66</td>
</tr>
<tr>
<td>( P = 0.74 )</td>
<td>66</td>
<td>27</td>
<td>12</td>
<td>0.63</td>
</tr>
<tr>
<td>( P = 0.75 )</td>
<td>69</td>
<td>19</td>
<td>17</td>
<td>0.66</td>
</tr>
<tr>
<td>( P = 0.77 )</td>
<td>78</td>
<td>25</td>
<td>2</td>
<td>0.74</td>
</tr>
<tr>
<td>( P = 0.79 )</td>
<td>78</td>
<td>25</td>
<td>2</td>
<td>0.74</td>
</tr>
<tr>
<td>( P = 0.80 )</td>
<td>78</td>
<td>25</td>
<td>2</td>
<td>0.74</td>
</tr>
<tr>
<td>( P = 0.81 )</td>
<td>78</td>
<td>25</td>
<td>2</td>
<td>0.74</td>
</tr>
</tbody>
</table>

6.1. Feature selection performance

Table 2 lists the confusion matrices and overall performance results for FSPA using four different performance thresholds: \( \sum = P_b \), \( \sum = P_a \), \( \sum = 0.5 \), and \( \sum = 0.0 \). We will discuss each threshold in turn including the feature regions that were selected and how they were combined.

Table 3 lists the results for \( \sum = P_b \) (the performance histogram is updated only when \( P_c \) is greater than the best current performance). The overall performance was significantly better than the LDA standard (0.76 versus 0.67). An interesting result is that 68% of the low quality objects were correctly classified. Moreover, only 14 L objects (13%) were misclassified as H objects, while only 16 H and M objects (3% of all H and M objects) were misclassified as L objects. These results were achieved using 37 of the original 52 software metrics (71%). Specifically, two feature regions were identified: the cross product of \([x_{19}, x_{27}]\) with \([x_{28}, x_{39}]\); and the cross product of \([x_{33}, x_{43}]\) with \([x_{7}, x_{18}]\). Table 3b and c lists the respective results for \( \sum = P_a \) (the performance histogram is updated only when \( P_c \) is greater than the current average performance) and for \( \sum = 0.5 \) (the performance histogram is updated only when \( P_c \) is greater than chance). The results using these thresholds are similar to each other and very slightly worse than \( \sum = P_b \) (0.75 and 0.74 versus 0.76) but still significantly better than the LDA standard. The sensitivity and specificity for low quality objects (versus H and M objects) are also comparable to \( \sum = P_a \). For \( \sum = P_a \), these results were achieved using 33 of the original 52 software metrics (63%). Two feature regions were identified: the cross product of \([x_{6}, x_{14}]\) with \([x_{28}, x_{40}]\); and the square of \([x_{41}, x_{52}]\). For \( \sum = 0.5 \), these results were achieved using 39 of the original 52 software metrics (75%). Again, two feature regions were identified: the cross product of \([x_{6}, x_{17}]\) with \([x_{28}, x_{38}]\); and the cross product of \([x_{12}, x_{24}]\) with \([x_{44}, x_{52}]\). Table 3d lists the results for \( \sum = 0.0 \). The overall performance was better than the LDA standard (0.71 versus 0.67) and 74% of the L objects were correctly classified. Further, only 2 of the L objects (2%) were misclassified as H objects. However, using this threshold only 40% of the M objects were correctly classified. Compare this extremely poor result with the LDA standard, 57%, and the best result (\( \sum = P_a \), 62%). Moreover, 28 H objects and 49 M objects (11% of all H and M objects) were misclassified as L objects. For \( \sum = 0.0 \), these results were achieved using 28 of the original 52 software metrics (54%). Two feature regions were identified: the cross product of \([x_{3}, x_{11}]\) with \([x_{12}, x_{20}]\); and the cross product of \([x_{9}, x_{50}]\) with \([x_{26}, x_{33}]\). For completeness, Fig. 2 plots the feature performance histogram for each of the four performance thresholds and each of the software metrics within the dataset.

6.2. Prediction aggregation performance

Fig. 3 is a plot of the overall classification performance using each performance threshold, \( \sum = P_b \), \( \sum = P_a \), \( \sum = 0.5 \), and \( \sum = 0.0 \). For each threshold, we show \( P \) using the aggregation of the best \( q = 11 \) classifier/region pairs with the fuzzy integration approach (described in Section 2.3) and compare each result with the best individual classifier result (described in Section 6.1) as well as a majority vote aggregation of the \( q \) classifier results. We see that the fuzzy aggregation results were superior to the best of the comparison results across all thresholds: 0.78 versus 0.76 (best individual); 0.78 versus 0.75 (both comparison results); 0.77 versus 0.76 (majority vote); and 0.74 versus 0.72 (majority vote).
One final experiment was to compare the suite of results described above against an external benchmark. We implemented the feature selection method (TK) described in [54] and used it to classify the dataset. TK achieved an overall classification accuracy of 0.75 (see Fig. 3). This is a better result than any achieved when $P = 0.0$ and is comparable to both the best individual and majority vote results using all thresholds. However, fuzzy aggregation produced superior results across all thresholds save 0.0.

Finally, we examine which software metrics contributed most often to successful classification results, in other words, the most discriminatory metrics. Fig. 4 is a plot of the relative frequency of software metrics usage within the best classifier/region pairs across all performance thresholds, $P = P_g$, $P = P_a$, $P = 0.5$, and $P = 0.0$. We now look at software metric values outside the standard deviation of mean metric usage. The most discriminatory metrics (in descending order) were $x_{42}$ (protected members), $x_{33}$ (operations), $x_{35}$ (operands), $x_{37}$ (remote methods), $x_{19}$ (method cohesion), $x_{31}$ (constructors), $x_{27}$ (parameter count), and $x_{34}$ (method name).

---

**Fig. 2.** Feature performance histogram for the software metrics and the performance thresholds: $\sum = P_g$ (solid black), $\sum = P_a$ (dashed black), $\sum = 0.5$ (solid grey), and $\sum = 0.0$ (dashed grey).

**Fig. 3.** Classification performance, $P$, for each threshold with fuzzy aggregation, the best individual classifier, and majority vote, as well as the external benchmark, TK.

**Fig. 4.** Plot of the relative frequency of software metrics usage within the best classifier/region pairs across all performance thresholds.
We have demonstrated that fuzzy feature selection with prediction aggregation is an effective pattern classification strategy. By sampling feature subsets using a fuzzy performance update rule (pre-processing) for presentation to a collection of classifiers, and aggregating the best independent results using fuzzy integration to produce consensus outcomes (post-processing), we achieve significantly better results than any respective best individual classifier. We empirically validated this strategy by examining a classification problem in quantitative software engineering using automatically generated software metrics to predict the quality of the corresponding software objects. Not only were the classification outcomes consistently superior across all experiments, but these results were achieved using only a fraction of the original software metrics.

The software engineering community has developed innovative metrics to quantitatively assess software systems. At the same time, software architects have accumulated significant expertise in subjectively evaluating software quality using the intuitive concepts of complexity, utility, and extensibility. No single metric can be an indicator of software quality for all types of systems; however, determining the “optimal” combination of multiple software metrics is certainly a non-trivial exercise. Incorporating the expertise of software architects 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 where the qualitative assessments are initially determined by the expert is a potentially viable option as demonstrated in this investigation.

This investigation did not involve an a priori determination that the software metrics used here were, in some sense, optimal or ideal predictors of software object quality. It is more than likely that this set is neither wholly necessary nor sufficient for this purpose. These metrics were selected for the purely practical reason that metric generation software was available to compile this specific set of metrics. A thorough review of the utility of other software metrics is warranted for future investigation.

Acknowledgments

The authors thank Conrad Wiebe for his assistance in implementing key aspects of the fuzzy feature selection with prediction aggregation algorithm, Alexander Demko for the implementation of the relevant Scopira framework objects, and Mark Alexiuk for his thorough assessment of the software objects used in the experiments.

We also gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC) for its support of this investigation.

References


