Empir Software Eng (2011) 16:812–841
DOI 10.1007/s10664-011-9162-z

Calculation and optimization of thresholds for sets of software metrics

Steffen Herbold · Jens Grabowski · Stephan Waack

Published online: 25 May 2011

Editors: Nachiappan Nagappan

Abstract In this article, we present a novel algorithmic method for the calculation of thresholds for a metric set. To this aim, machine learning and data mining techniques are utilized. We define a data-driven methodology that can be used for efficiency optimization of existing metric sets, for the simplification of complex classification models, and for the calculation of thresholds for a metric set in an environment where no metric set yet exists. The methodology is independent of the metric set and therefore also independent of any language, paradigm or abstraction level. In four case studies performed on large-scale open-source software metric sets for C functions, C++, C# methods and Java classes are optimized and the methodology is validated.

Keywords Software metrics · Thresholds · Machine learning · PAC

1 Introduction

Software has become part of the everyday life. Embedded software in modern cars controls the distance to the car in front of us. News portals on the Internet utilize sophisticated distributed software to report the news events as they occur. Users expect and need software to conform to a certain standard of quality. The *International Organization for Standardization* (ISO) defines quality as the "*degree to which a set of inherent characteristics fulfills requirements*" in the ISO 9000 standard

S. Herbold (✉) · J. Grabowski · S. Waack
Institute of Computer Science, University of Göttingen,
Goldschmidtstr. 7, 37077 Göttingen, Germany
e-mail: herbold@cs.uni-goettingen.de

J. Grabowski
e-mail: grabowski@cs.uni-goettingen.de

S. Waack
e-mail: waack@cs.uni-goettingen.de

(ISO/IEC 2005). To uphold the required standard of quality, the assurance that software quality attributes are fulfilled is an important aspect of the execution of software projects. Quality attributes like maintainability and understandability are often assessed using *software metrics*. Software metrics provide means to put numbers on abstract attributes, such as complexity or size. Often, one metric is insufficient to effectively analyze a quality attribute. Instead, we use a set of metrics to determine whether a quality attribute is fulfilled or problematic. To determine if metric values are good or bad, clear indicators are required. Otherwise such metric sets are hard to interpret. For this purpose, we use thresholds for metric values: a quality attribute is said to be problematic, when at least one threshold for a metric is violated. For thresholds to be effective indicators, the quality of the threshold values themselves is of great importance. However, the thresholds often depend on the project environment, e.g., programming languages and tool support. Therefore, the definition of thresholds is often problematic and defined thresholds may not be valid in other environments.

During the last years, machine learning has been successfully applied and has become a standard technique for data analysis in many different fields, such as gene analysis in biology, or data mining techniques companies use to optimize their marketing strategies. It has also been used in computer science, e.g., for defect prediction (Nagappan et al. 2006). In this article, we introduce an algorithmic approach for the optimization of the size software metric sets and threshold values used. To this aim, a machine learning algorithm is used to define an approach for the calculation of thresholds for a metric set. In a previous work (Werner et al. 2007), we used relatively simple brute-force approach for the calculation of threshold values for a metric set for the *Testing and Test Control Notation* (TTCN-3) (ETSI 2007; Grabowski et al. 2003). However, such a brute force approach has scalability problems and is therefore infeasible for larger metric sets. This work presents a more sophisticated approach, which utilizes the learning of axis-aligned d-dimensional rectangles for the threshold calculation. The objective of this work is to reduce the complexity of metric-based classifiers for software quality to improve their understandability and interpretability, which will benefit both researchers and the industry as it allows to pinpoint the source of deficits more effectively. To this end, we provide a versatile, data-driven means for both threshold calculation and the optimization of metric sets integrated into a single algorithm.

The contribution of this article is fourfold.

1. A machine learning based method for the computation of thresholds for metric sets.
2. A high-level methodology for the optimization of already existing metric sets with thresholds.
3. Using the same methodology to effectively replace existing classification methods, and thereby reducing their complexity.
4. An outline how a good metric set with thresholds can be determined in an environment where no thresholds exist yet.

For the first contribution, we show how the problem of rectangle-learning relates to sets of software metrics with thresholds and how rectangle learning can be utilized to compute thresholds. The second contribution defines a generic methodology for metric set efficiency optimization, which is in fact not restricted to software metrics,

but applicable to metric sets in general. For this, we assume that an effective metric set with thresholds is already existing. We show how a smaller and effective set can be determined, which is due to its reduced size also efficient. The third contribution shows how this approach can be used to replace existing classification strategies, that may not even be metric-based, with a threshold based classification. Such a replacement can be used to substitute hard-to-interpret or black-box classifiers with easy-to-interpret threshold classification. Finally, we show how our approach can be used to determine a good metric set in an environment where no means for the automated classification of software entities exists yet. In comparison to the first contribution, this includes not only the calculation of the threshold values, but also the selection of an appropriate subset of metrics from a possibly large set of candidate metrics.

All methodologies defined in course of this article are independent of the metric sets themselves and only depend on actually observed data. The methods are therefore independent of any specific programming language (e.g., C, Java) and level of abstraction (e.g., methods, classes). In four case studies, we validated that the approach works well for product metrics in large-scale open-source software projects. As part of the case studies, metric sets for C function, C++ and C# methods, and Java classes are analyzed.

The structure of this article is as follows. In Section 2, we introduce the concepts of software metrics and how they can be used in combination with thresholds for quality estimation. Afterwards, we briefly introduce machine learning and define the foundations of the learning approach used in this article in Section 3. In Section 4, we define the methodology for the optimization of software metric sets with thresholds and provide a description of how it can be applied to perform different tasks is. We validate the applicability and effectiveness of the approach in two case studies, presented in Section 5. We discuss the results of the case studies in Section 6. Afterwards, the article is put into the context of related work in Section 7. Finally in Section 8, we summarize the results and conclude the article.

2 Software Metrics

According to Fenton and Pfleeger, "*Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules*" (Fenton and Pfleeger 1997). A way to measure software is to use *software metrics*. The IEEE defines software metrics as "*the quantitative measure of the degree to which a system, component, or process possess a given attribute*" (IEEE 1990). This means that a software metric is a clearly defined rule, that assigns values to software entities (e.g., components, classes, or methods) or attributes of development processes.

Fenton and Pfleeger divided software metrics into three categories (Fenton and Pfleeger 1997): *process metrics* measure attributes of a development process itself; *product metrics* measure documents and software artifacts that were produced as part of a process; *resource metrics* measure the resources, that were utilized as part of a process. Furthermore, each metric measures either an *internal* or an *external* attribute. Internal attributes are those that can be measured by observing only the process, product or resource itself, without considering its behavior. External

attributes on the other hand are attributes that are related to the behavior of software systems. In this work, the focus is on internal product metrics that measure source code. Some examples for internal attributes that relate to source code are size, reuse, modularity, algorithmic complexity, coupling, functionality, and control-flow structuredness (Fenton and Pfleeger 1997). Further attributes are staticness, method complexity, or attributes that relate to object-oriented software, such as usage of inheritance.

2.1 Metric Sets Under Study

The methods described in this article are general and may be used independent of a specific metric set. However, as part of this article, metric sets for the evaluation of the maintainability are studied exemplary. This is done with two different metric sets on different levels of abstraction: methods and classes. The maintainability describes non-functional aspects such as testability, understandability, or changeability of software. Because no single metric is able to cover all these aspects, we employ a set of metrics that covers internal attributes like the structure, size, and complexity instead. We selected the metrics based on our experience and with the aim to cover the maintenance related aspects of the source code that can be measured automatically with internal product metrics.

For the analysis of methods and functions, we use four metrics listed in Table 1a. With the control-flow structuredness measured by the *Cyclomatic Number* (VG) and *Nested Block Depth* (NBD), coupling measured by *Number of Function Calls* (NFC), and size measured by *Number of Statements* (NST), these metrics cover most of the maintainability-related attributes of methods, except the algorithmic complexity. Since algorithmic complexity is not really an attribute of the source code and cannot be measured automatically, it has been omitted. While both VG and NBD measure the control-flow structuredness, they measure different aspects of this attribute: NBD measures the maximum nesting of structural blocks, while VG measures the overall branching between blocks.

For the analysis of classes, the seven metrics listed in Table 1b are used. With these metrics, five internal attributes of classes are evaluated. The metric *Weighted Methods per Class* (WMC) measures the method complexity as the sum of the metric VG measured for all methods in a class. The metrics *Coupling Between Objects* (CBO) and *Response For a Class* (RFC) measure the coupling. For the measurement of the size of a class, the metrics *Number of Methods* (NOM) and *Lines of Code* (LOC) are utilized. The use of inheritance is measured by *Number of Overridden Methods* (NORM), the staticness of a class is measured by the metric *Number of Static Methods* (NSM). We included the attributes inheritance and staticness, as they greatly influence the maintainability of classes (Daly et al. 1996). Inheritance is often difficult to test and also decreases the understandability of the source code. Static methods and attributes can pose problems, as they are global for all instances of a class and can therefore introduce unwanted side effects.

One might have noted that with WMC, CBO, and RFC, three of the six popular metrics defined by Chidamber and Kemerer (1994) are used. Initially, all of the six metrics were part of the set, but the metrics *Depth of Inheritance Tree* (DIT), *Number of Children* (NOC) and *Lack of Cohesion in Methods* (LCOM) were excluded due to their poor distributions. LCOM was found to be poorly distributed by Basili et al.

Table 1 Metrics used in this article

Metric name	Internal attribute	Description
(a) Metrics for methods and functions		
Cyclomatic Number (VG)	Control-flow structuredness	Calculated based on the control flow graph $G = (V, E)$ and number of a method M as $VG(M) = \|E\| - \|V\| + p$, where p is the number of entries and exits.
Nested Block Depth (NBD)	Control-flow structuredness	Maximum number of nested blocks in a method.
Number of Function Calls (NFC)	Coupling	Number of functions called by a method
Number of Statements (NST)	Size	Number of statements of a method
(b) Metrics for classes		
Weighted Methods per Class (WMC)	Method complexity	Complexity of a class as the sum of the complexity of its methods. Here, VG is used as complexity measure.
Coupling Between Objects (CBO)	Coupling	Number of classes, to which a class is coupled.
Response For a Class (RFC)	Coupling	Size of the *response set* of a class, i.e. all methods that can be invoked directly or indirectly by calling a method of a class.
Number of Overridden Methods (NORM)	Inheritance	Number of methods defined by a parent that are overridden by a class
Number of Methods (NOM)	Size	Number of methods of a class
Lines of Code (LOC)	Size	Lines of code, excluding empty and comment-only lines.
Number of Static Methods (NSM)	Staticness	Number of static methods of a class

(1996). Furthermore, DIT and NOC are poorly distributed in the projects measured for the case studies in this work. We discuss their exclusion in Section 5.3.

2.2 Thresholds for Software Metrics

In general, thresholds discriminate values. In case a threshold defines an upper bound, the values that are greater than a threshold value are considered to be problematic, the values lower are considered to be acceptable. Thus, by defining thresholds a simple analysis of measured values is possible. For the interpretation of software metrics thresholds are required. For example, consider a metric m that measures the size of an entity x. Then a threshold t can be used to determine if x is to large:

$$m(x) > t \Rightarrow x \text{ is too large.}$$

While the above is an example of a threshold used as an upper bound, it might as well be a lower bound. For simplicity, we assume that thresholds are always upper bounds. However, this is no restriction as lower bounds can be transformed into upper bounds. Let m a metric with threshold t that defines a lower bound, i.e., entities x are considered to be problematic if $m(x) < t$, which is equivalent to $1/m(x) > 1/t$ if $m(x)$ and t are non-negative, as metrics and thresholds usually are. By defining a new metric $m'(x) = 1/m(x)$ and a new threshold $t' = 1/t$ a new metric with the opposite order is defined and with t' a threshold is obtained that defines an upper bound. However, by inverting the metric, its scale is changed. Another way to transform a lower bound into an upper bound while keeping it to scale is to subtract the metric from a maximum value. Let $m_{\max}$ the maximum value of metric m. Then

$$m(x) > t \Rightarrow m(x) - m_{\max} > t - m_{\max} \Rightarrow m_{\max} - t > m_{\max} - m(x)$$

and a new metric $m''(x) = m_{\max} - m(x)$ and a new threshold $t'' = m_{\max} - t$ are obtained, where t'' is an upper bound for m''. However, this method has the disadvantage that a maximum value $m_{\max}$ has to be known.

Thresholds are not without problems. The first is the generality of threshold values. In general, a threshold value is good in one setting must not necessarily be good every setting. Depending on the organization, the programming language, the tools used, the qualification of the developers, among other factors that are project dependent, good threshold values may vary. This is a problem, as each organization, and maybe even each project, has to define thresholds that are chosen depending on its environment. This issue directly relates to a second issue, as good thresholds depend on so many factors, the definition of thresholds itself is a problem. Therefore, a methodology to determine environment specific thresholds is required.

Table 2 Threshold values for the metrics to measure the maintainability

Metric name	Language	Threshold	Source
(a) Metrics for methods and functions			
VG	C	24	French (1999)
	C++	10	French (1999)
	C#	10	French (1999)
NBD	C	5	French (1999)
	C++	5	French (1999)
	C#	5	French (1999)
NFC	C	5	–
	C++	5	–
	C#	5	–
NST	C	50	–
	C++	50	–
	C#	50	–
(b) Metrics for classes			
WMC	Java	100	Benlarbi et al. (2000)
CBO	Java	5	Benlarbi et al. (2000)
RFC	Java	100	Benlarbi et al. (2000)
NORM	Java	3	Lorenz and Kidd (1994)
LOC	Java	500	Adapted from Copeland (2005)
NOM	Java	20	Adapted from Copeland (2005)
NSM	Java	4	Lorenz and Kidd (1994)

To allow a more differentiated analysis more than one threshold value for one metric can be defined. In this article, we assume source code to be either problematic or un-problematic. However, further shades of gray exist in between. For example, there may be two thresholds, a low one for *weak infractions* and a higher one for *critical infractions*. In this study, we only consider defining a single threshold for a given metric.

As baseline for our analysis, we use the thresholds listed in Table 2 for the metric sets introduced in the previous section to analyze the maintainability of methods and classes. Most of them were determined as good thresholds for these metrics in previous work (Lorenz and Kidd 1994; French 1999; Benlarbi et al. 2000). The languages and the environments for which these threshold values were determined are sufficiently similar to the setting of this work, which is why we argue that these thresholds are transferable to our application. The thresholds for the metrics LOC and NOM are based on thresholds used by the source code analysis tool PMD (Copeland 2005). PMD defines a threshold value of 1000 for lines of code including empty and comment-only lines for Java classes. In this work, we use different definition for the lines of code metric that excludes both empty and comment-only lines, thus we adapted the value to 500. Furthermore, PMD defines a threshold value of 10 for the number of methods excluding methods that start with "get" or "set". As the metric NOM counts all methods, the threshold value is adapted to 20 to account for the additional methods. For the remaining two metrics, NFC and NST, no reference values are available in the literature. Therefore, based on our experience, reasonable threshold values for both metrics have been defined, 5 for NFC and 50 for NST.

3 Foundations of Machine Learning

In this section, we introduce the concepts of machine learning essential for this work. After a brief description of machine learning in general, we define the learning framework used in this work in Section 3.1. Finally, we discuss an algorithm to learn axis-aligned d-dimensional rectangles in Section 3.2. The approach for the optimization of metric sets is based on this algorithm.

In general, machine learning is a way to analyze data. Learning theory assumes that observed data can be described by an underlying process. The type of the process varies and depends on the type of learning. For example, it could be an automaton, but also a stochastic process. The aim of machine learning is to identify this process. Often, this is not accurately possible. However, in most cases it is still possible to detect patterns within the data. Assuming that the underlying stochastic process does not change, it is possible to predict properties of unseen data using the detected patterns. A more detailed introduction to machine learning in general can be found in the literature (e.g. Devroye et al. 1997; Shawe-Taylor and Cristianini 2004; Schölkopf and Smola 2002).

3.1 Concept Learning in the Presence of Noise

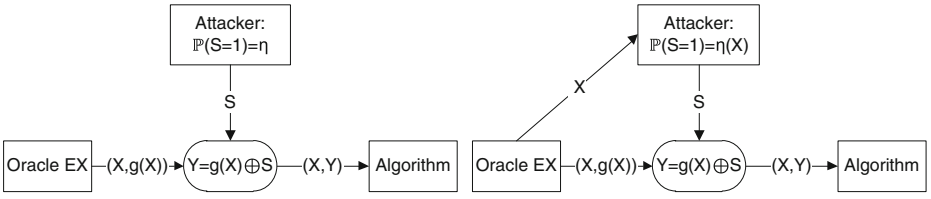
In this work, we use *concept learning*. A concept defines how to divide vectors from the $\mathbb{R}^d$ into positive and negative examples. The task of a learning algorithm is to

infer a *target concept* g out of a *concept class* $\mathscr{C}$. The target concept can also be interpreted as the *bayesian classifier* (Duda and Hart 1973) of the concept. A concept can also be understood as a map $g : \mathfrak{X}^d \to \{0, 1\}$, where $\mathfrak{X}^d \subset \mathbb{R}^d$ denotes the *input space*. A learning sample is of the form $U = (X, Y) \in \mathfrak{X}^d \times \{0, 1\}$, where the *input element* X is randomly distributed according to the *sample distribution* $\mathscr{D}$ defined over the input space $\mathfrak{X}$, Y is the *random label* or *output element* associated with X. In a noise free setting, the value of Y depends only on the random vector X and the target concept g and $Y = g(X)$. To obtain samples U, the concept of an *oracle* is used. On request, an oracle $EX(\mathscr{D}, g)$ randomly draws an input element X according to the distribution $\mathscr{D}$, classifies X using g and returns a sample $U = (X, g(X))$. In practical applications, the oracle can be seen as a training sample that contains classified entities to be used for the learning.

Real-life applications are usually not noise-free, i.e., the property $Y = g(X)$ is not always fulfilled. Most algorithms designed to work on noise-free data often perform poorly or do not work at all in the presence of noise. Therefore, noise modelling and algorithms that use these models for learning in the presence of noise are important. One way to introduce noise into a learning model is the *classification noise* model, which was first formalized by Angluin and Laird (1988). Further details on noise models can be found in Mammen and Tsybakov (1999), Tsybakov (2004). In the classification noise model, the label of the output variable Y is changed with a fixed probability and $Y = g(X) \oplus S$, where $\oplus$ denotes the symmetric difference. The *random noise* $S \in \{0, 1\}$ is 1 with probability η, i.e., $\mathbb{P}(S = 1) = \eta$, where η denotes the *noise rate*. In the classification noise model, S is independent of the input element X. It follows directly that $\mathbb{P}(Y \neq g(X)) = \eta$. In combination with oracles, noise can be seen as an attacker that corrupts the output element of a sample generated by an oracle. Figure 1a visualizes this concept.

In the *Statistical Query Model* (SQM) proposed by Kearns (1998) *query functions* of the form $\chi : \mathfrak{X}^d \times \{0, 1\} \to [a, b]$ are used to infer information about the data. For this purpose, a statistical oracle is introduced that returns the expected result of the queries within a specified degree of precision. The estimation is based on noise models.

We use a generalization of the classification noise model, where the random noise rate is orthogonal to the target concept (Brodag et al. 2010). The restriction that S is independent of the input element X is dropped. Instead, we introduce a *random noise rate* $\eta(X)$ that depends on the input element, as shown in Fig . 1b. Hence, $\eta(x) = \mathbb{P}(S = 1 | X = x)$ and thus the random noise depends on the input. For a given



(a) Classification noise (b) Conditional noise

Fig. 1 Visualization of noise as an attacker

$x \in \mathfrak{X}^d$, the noise rate is $\eta(x) = \mathbb{P}(Y \neq g(X)|X = x)$ and for $y_0 \in \{0, 1\}$ the *conditional expected noise rate* given $g(X) = y_0$ is

$$\eta_{y_0} := \mathbb{E}(\eta(X)|g(X) = y_0). \tag{3.1}$$

Using the conditional expected noise rates η_0, η_1, the *expected noise rate* can be calculated as $\eta := \mathbb{E}\eta(X) = \eta_0\mathbb{P}(g(X) = 0) + \eta_1\mathbb{P}(g(X) = 1)$.

Furthermore, we assume that query functions are *admissible*. A query function χ is admissible, if it is not correlated to the noise rate $\eta(X)$ conditioned on the concept $g(X)$. The geometrical uncorrelation is orthogonality, hence it is said that the noise is orthogonal to the target concept. For the learning, this means that it is not possible to infer the value of χ by simply considering the noise rate $\eta(X)$. This is a reasonable assumption, as usually no information about the result of a query is obtained by simply considering the noise rate.

Based on the introduced concepts and definitions, we can state the central theorem of the learning framework. This theorem describes how the expected value of an admissible query can be calculated if the conditional expected noise rates η_0 and η_1 are known.

Theorem 1 *Let χ be admissible with respect to $g \in \mathscr{C}$, $y_0 \in \{0, 1\}$. Then*

$$\mathbb{E}(\chi(X, y_0)|g(X) = y_0) = \frac{(1 - \eta_{\bar{y}_0})\mathbb{E}[\mathbb{1}_{\{Y=y_0\}}\chi(X, y_0)] - \eta_{\bar{y}_0}\mathbb{E}[\mathbb{1}_{\{Y=\bar{y}_0\}}\chi(X, y_0)]}{\mathbb{P}(Y = y_0) - \eta_{\bar{y}_0}}. \tag{3.2}$$

The proof of Theorem 1, as well as further details concerning learning with orthogonal noise, can be found in Brodag et al. (2010). The function $\mathbb{1}$ is defined as

$$\mathbb{1}_{\{cond\}} = \begin{cases} 1 & \text{if } cond \text{ is } true \\ 0 & \text{if } cond \text{ is } false. \end{cases} \tag{3.3}$$

On a learning sample, the expected values $e_{y_0} := \mathbb{P}(Y = y_0)$, $e_{\chi,y_0} := \mathbb{E}[\mathbb{1}_{\{Y=y_0\}}\chi(X, y_0)]$, and $e_{\bar{\chi},y_0} := \mathbb{E}[\mathbb{1}_{\{Y=\bar{y}_0\}}\chi(X, y_0)]$ can be estimated using standard *maximum-likelihood estimators*. With the estimated values, the conditional expected value of a query $\mathbb{E}(\chi(X, y_0)|g(X) = y_0)$ can be calculated for $y_0 \in \{0, 1\}$ according to Theorem 1 as

$$\mathbb{E}(\chi(X, 0)|g(X) = 0) = \frac{(1 - \eta_1)e_{\chi,0} - \eta_1 e_{\bar{\chi},0}}{e_0 - \eta_1} \tag{3.4}$$

and

$$\mathbb{E}(\chi(X, 1)|g(X) = 1) = \frac{(1 - \eta_0)e_{\chi,1} - \eta_0 e_{\bar{\chi},1}}{e_1 - \eta_0}, \tag{3.5}$$

where $\bar{\chi}$ is an abbreviation for $\bar{\chi}(x, y) = \chi(x, \bar{y})$. The conditional noise rates η_0 and η_1 are usually unknown and estimated by guessing. In practical applications, the noise rates are guessed by sampling. For example, if we estimate that the noise rate is between 0.1 and 0.2, hypotheses for all pairs of noise rates $\eta_0, \eta_1 = 0.1, 0.11, \ldots, 0.19, 0.2$ could be calculated. Afterwards, we select the best of these

hypotheses. This is not an estimation of the noise rate itself, i.e., the noise rate used to calculate the optimal solution is not necessarily the true noise rate. Such noise handling strategies guarantee that if a noise rate close to the true noise rate is part of the sampled noise rates, the result is at least as good as it would be with the true noise rate.

3.2 A Rectangle Learning Algorithm

In this work, we adapted the algorithm for learning axis-aligned d-dimensional rectangles proposed by Kearns (1998) to the noise model described above. The main adaptations are that the conditional expected noise rates η_0 and η_1 have both to be sampled, instead of only the expected noise rate η. Furthermore, the statistical oracle used by the algorithm is changed from the SQM to the random noise model by calculating the expected results of statistical queries based on Theorem 1. The algorithm has two phases. In the first phase, the training data is partitioned according to its distribution. In the second phase, the rectangle is computed based on this partition. Both phases are described in the following.

The aim of the first phase of the algorithm is to find a partition of the d-dimensional real-space, such that

$$\mathbb{P}(X_i \in I_{i,p}) = \mathbb{P}(X_i \in I_{i,q}) = \frac{1}{\lceil 1/\varepsilon \rceil} \approx \varepsilon \tag{3.6}$$

for each dimension $i = 1, \ldots, d$ and $p, q = 1, \ldots, \lceil 1/\varepsilon \rceil$ for $X \in \mathbb{R}^d$ randomly distributed according to $\mathscr{D}$, where X_i donates the i-th component of X and ε an *error bound* that the calculate hypothesis should abide. This means that it is equally likely that the i-th component of the randomly drawn vector X falls into any of the intervals $I_{i,\cdot}$. In the implementation of the algorithm, a sorting algorithm is utilized to obtain these intervals according to the empirical distribution of a discrete training sample. After sorting the values for the i-th dimension, the intervals $I_{i,p}$ can be defined by assigning the first $\frac{p}{\lceil 1/\varepsilon \rceil}$ vectors to $I_{i,1}$, the next $\frac{p}{\lceil 1/\varepsilon \rceil}$ to $I_{i,2}$ and so on. These intervals fulfill the property defined by (3.6). If there are n samples in a training set, the complexity of the first phase is $O(dn \log n)$, as for each dimension the samples have to be sorted and efficient sorting algorithms are $O(n \log n)$.

In the second phase, the boundaries of the target rectangle are calculated. For each dimension separately, the probability $p_{I_{i,p}} = \mathbb{P}(X_i \in I_{i,p} | g(X) = 1)$, i.e., the probability that the target rectangle intersects an interval $I_{i,p}$ is calculated. This probability is calculated using admissible queries and (3.5). If the target rectangle intersects an interval, the probability $p_{I_{i,p}}$ should be significantly larger than 0. Thus, for each dimension i, the probabilities $p_{I_{i,p}}$ are calculated from the left, i.e., $p = 1, 2, \ldots$. The first interval, for which $p_{I_{i,p}}$ is significant defines the left, i.e., lower boundary l_i of the rectangle in the i-th dimension. The same is done from the right, i.e., $p = \lceil 1/\varepsilon \rceil, \lceil 1/\varepsilon \rceil - 1, \ldots$ to determine the right, i.e., upper boundary u_i. Using this procedure for each dimension, boundaries (l_i, u_i) are calculated.

In the second phase, for each dimension, the probability $p_{I_{i,o}}$ is calculated for at most $\lceil 1/\varepsilon \rceil$ intervals from the left and analogously from the right. The estimation of

this probability is $O(n)$. Thus the complexity of the second phase is $O(dn\frac{1}{\varepsilon})$ and the overall complexity of the algorithm is $O(dn \log n + dn\frac{1}{\varepsilon})$.

4 Optimization of Metric Sets and Thresholds

In this section, we introduce our machine learning based approach to optimize metric sets with thresholds for the detection of problematic entities. First, we describe in Section 4.1 how the rectangle learning algorithm is utilized to calculate thresholds. Based on that, we define a threshold optimization algorithm for the calculation of an optimized metric set with thresholds in Section 4.2. Then, in Sections 4.3–4.5, we show three applications for this threshold optimization algorithm: 1) optimization of an existing metric set with thresholds to obtain an effective and efficient subset; 2) reduction of the complexity of the used classification method; 3) determination of environment specific thresholds.

4.1 Calculation of Thresholds Using Rectangle Learning

The analysis approach is based on a given metric set $M = \{m_1, \ldots, m_d\}$ and a set of software entities $\mathbf{X}$ with known classifications $\mathbf{Y}$. The aim is to obtain thresholds $T = \{t_1, \ldots, t_d\}$ for the metrics in M such that the metric set can be used to discriminate software entities in the same way, as it is done by the pair $(\mathbf{X}, \mathbf{Y})$. By measuring the software entities $\mathbf{X}$ with M, we transform the set of software entities $\mathbf{X}$ into a set of vectors in the d-dimensional real space, such that $M(\mathbf{X}) := \{(m_1(x), \ldots, m_d(x)) : x \in \mathbf{X}\} \subset \mathbb{R}^d$. The pair $(M(\mathbf{X}), \mathbf{Y})$ is the input for the axis-aligned rectangle learning algorithm, introduced in Section 3.2. As result, the algorithm yields pairs of upper and lower bounds (l_i, u_i) for each dimension $i = 1, \ldots, d$. As the i-th dimension represents the values the software entities calculated using the metric m_i and under the assumption that high metric values are bad, we interpret the upper bound of the rectangle in the i-th dimension as the threshold for the metric m_i. Therefore, with $t_i = u_i$ a set of thresholds $T = \{t_1, \ldots, t_d\}$ for the metric set M is obtained. For an entity x, the classification of the metric set M and the thresholds T is defined as

$$f_0(x, M, T) = \begin{cases} 1 & \text{if } |\{\, i \in \{1, \ldots, d\} : \; m_i(x) > t_i \,\}| = 0 \\ 0 & \text{if } |\{\, i \in \{1, \ldots, d\} : \; m_i(x) > t_i \,\}| > 0, \end{cases} \tag{4.1}$$

i.e., $f_0(x, M, T)$ is zero when at least one metric m_i exceeds its threshold t_i, and is one when none of the metrics exceeds its threshold.

Under the assumption that metric values are positive, this classification describes a rectangle with upper bounds t_i and 0 as the lower bound. Figure 2a visualizes this in a 2-dimensional setting to clarify the relationship between rectangle learning and the usage of metric thresholds for the classification of software entities. Algorithm 1 describes this methodology in a step-wise fashion. The algorithm can be used to

determine thresholds for a metric set given *any* training sample ($\mathbf{X}$, $\mathbf{Y}$) and *any* metric set, regardless of how the training sample or the metric set were determined.

Algorithm 1: Algorithm for the calculation of thresholds

Input : Set of software entities $\mathbf{X}$ with classifications $\mathbf{Y}$, metric set $M = \{m_1, \ldots, m_d\}$
Output: Thresholds $T = \{t_1, \ldots, t_d\}$
$M(\mathbf{X}) \leftarrow \{((m_1(x), \ldots, m_d(x)) : x \in \mathbf{X}\}$;
Apply the rectangle learning algorithm to $(M(\mathbf{X}), \mathbf{Y})$ and obtain (u_i, l_i), $i = 1, \ldots, d)$;
$t_i \leftarrow u_i$ for all $i = 1, \ldots, d$;
$T \leftarrow \{t_1, \ldots, t_d\}$;
return T

The *classification error* is defined as the probability that a randomly drawn sample (X, Y) is classified wrongly

$$\varepsilon = \mathbb{P}(f_0(X, M, T) \neq Y). \tag{4.2}$$

Consequently, the *empirical classification error* on a given training sample ($\mathbf{X}$, $\mathbf{Y}$) is defined as

$$\varepsilon_{\mathbf{X},\mathbf{Y}}(M, T) = \frac{1}{|\mathbf{X}|} \sum_{(x,y)\in(\mathbf{X},\mathbf{Y})} \mathbb{1}_{f(x,M,T)\neq y}. \tag{4.3}$$

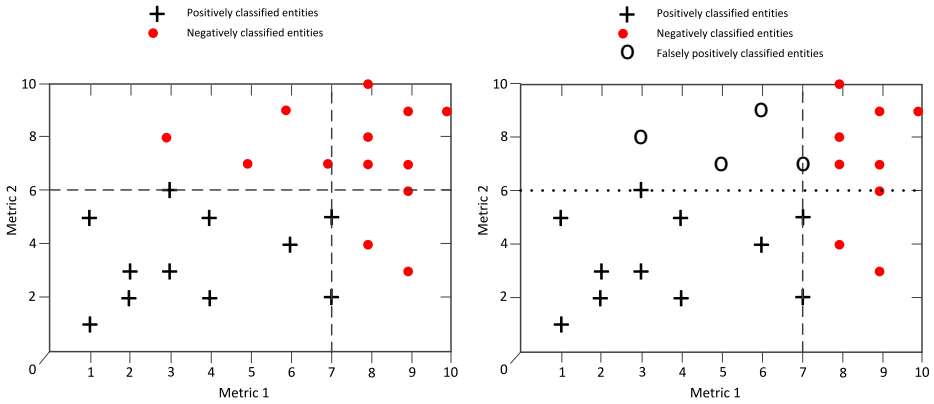
4.2 Threshold and Metric Set Optimization Algorithm

Next, we define a threshold optimization algorithm that computes an optimized metric set based on the calculation of thresholds for a metric set. This means a metric set that is not only effective with respect to the classification it yields, but also efficient in terms of its size. To achieve this, we reduce the dimension of the metric set and recalculate the threshold values for the reduced sets. Recalculating the thresholds allows the algorithm to, e.g., enforce a stronger classification using one metric while dropping another from the set.

The algorithm uses an existing method f for the classification of software entities $\mathbf{X}$. By applying f to the entities $x \in \mathbf{X}$, the classification $\mathbf{Y}$ can be calculated as $\mathbf{Y} = \{f(x) : x \in \mathbf{X}\}$. The resulting pair ($\mathbf{X}$, $\mathbf{Y}$) is the basis for the calculation of thresholds.

Let M be a metric set to be used as basis for the determination of an optimized, i.e., effective and efficient metric set with thresholds. A metric set is called effective if its classification error is close to 0, i.e., less than or equal to a threshold for the error $\delta \in \mathbb{R}$. A metric set is called efficient if it is the smallest set to do so. Therefore, we need to calculate a subset $M' = \{m'_1, \ldots, m'_{d'}\} \subseteq M$ with thresholds $T' = \{t'_1, \ldots, t'_{d'}\}$ that yields classification error smaller than δ. To this aim, we determine thresholds based on the training set ($\mathbf{X}$, $\mathbf{Y}$) for all subsets of M. In other words, all sets that are element of the power set of M: $M' \in \mathscr{P}(M) \setminus \emptyset$. Then, for each subset M' the empirical classification error $\varepsilon_{\mathbf{X},\mathbf{Y}}$ is calculated. The smallest set M' that has a classification error $\varepsilon_{\mathbf{X},\mathbf{Y}} \leq \delta$ is an effective and efficient subset of M. Algorithm 2 describes the whole threshold optimization algorithm in a step-wise fashion. We discuss the run time and scalability of the algorithm in Section 6.1 (research question R5).

The value δ can be used as a steering parameter, depending on the accuracy expected of the optimized set and the available data. The higher the accuracy shall be and the more data is available, the smaller δ should be. It is possible that no M',



(a) Classification with two thresholds (b) Approximation with only one threshold

Fig. 2 Example for the classification of values using thresholds

T' satisfies the condition that its error is below δ. In this case, there are three options to proceed: 1) choose a larger value for δ; 2) use a different metric set M; 3) abort the optimization efforts and conclude that a metric set M' with threshold T' is insufficient to describe the classification. As a practical matter, δ can be sampled, e.g., by starting with $\delta = 0.01$ and increasing it in 0.01 steps till a metric set found.

Algorithm 2: The threshold optimization for metric set optimization

Input : Effective classification method f, set of software entities $\mathbf{X}$, metric set M, error threshold δ
Output: Effective and efficient metric set M^* with thresholds T^*
$\mathbf{Y} \leftarrow \{f(x) : x \in \mathbf{X}\}$;
foreach $M' \in \mathscr{P}(M)$ **do**
  Calculate thresholds T' for $M', \mathbf{X}, \mathbf{Y}$ with Algorithm 1
end
$M^* \leftarrow \min\{M' \in \mathscr{P}(M) : \varepsilon_{\mathbf{X},\mathbf{Y}}(M', T') \leq \delta\}$;
return M^* *and the corresponding* T^* ;

4.3 Optimization of the Efficiency of Metric Sets with Thresholds

Given an existing effective metric set, the threshold optimization algorithm can determine an effective and efficient subset. Let M be a metric set with thresholds T and $\mathbf{X}$ a set of software entities. The function $f_0(x, M, T)$ defines a classification method for $\mathbf{X}$. Then, f_0, $\mathbf{X}$, M, and an appropriate value for δ are the input for the threshold optimization algorithm which will compute an optimized subset M^* with thresholds T^*.[1] As an example, Fig. 2 shows how the classification obtained by two metrics is approximated by only one of the two metrics. The dashed lines visualize the thresholds of the two metrics, used to classify the samples for the training. In Fig, 2a, both metrics are used for the classification; in Fig. 2b, only metric one is used. The squares mark the entities that are misclassified by the approximation.

[1] We use M', T' to denote any subset/threshold combination and M^*, T^* to denote optimal solutions.

4.4 Reduction of the Classification Complexity

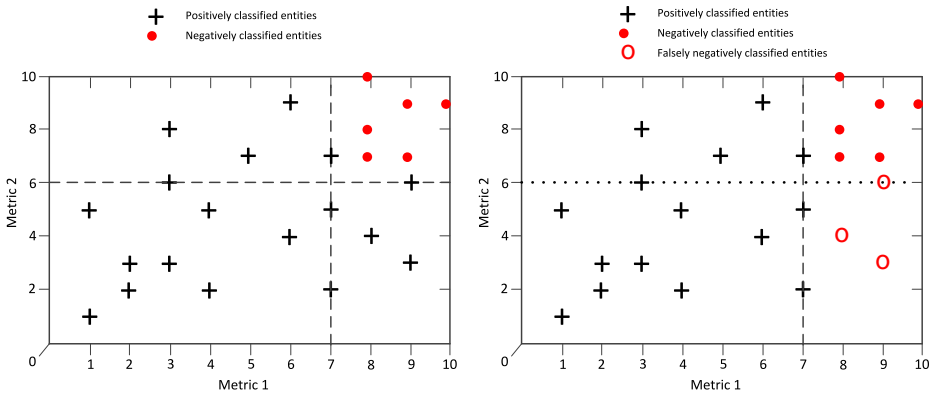
Another way to utilize the threshold optimization algorithm is to reduce the complexity of the classification scheme. With thresholds, a simple kind of classification is described: if a threshold is violated, an entity is problematic. This makes it clear why an entity is problematic and also provides an indicator what the problem might be. A slightly more complex approach is to allow a number of infractions λ, i.e., λ thresholds may be violated. The function

$$f_\lambda(x, M, T) = \begin{cases} 1 & \text{if } |\{\, i \in \{1, \ldots, n\} : \; m_i(x) > t_i \,\}| \leq \lambda \\ 0 & \text{if } |\{\, i \in \{1, \ldots, n\} : \; m_i(x) > t_i \,\}| > \lambda, \end{cases} \tag{4.4}$$

describes the classification defined by a metric set M with thresholds T. With $\lambda = 0$, this f_λ is equal to f_0 (4.1), hence f_λ is a generalization of f_0.

One reason to use such a rule is to grant the developers more freedom, e.g., allowing short methods with a high structural complexity or long methods with a low structural complexity. But methods that are both long and structurally complex are forbidden. The classification with λ allowed infractions introduces an additional complexity to understand *why* a problematic entity was classified as such and which counter measures can be taken. Complex approaches that may yield a very good classification may be difficult to impossible to interpret, e.g., Support Vector Machine (SVM) based techniques (Schölkopf and Smola 2002). Other techniques, e.g., classification trees (Quinlan 1986) show directly why an entity was classified as problematic, but it not clear how to fix as the tree may hide other reasons why the entity is problematic. In general, the classification could be performed by an arbitrary complex function f. A metric set that yields the same classification, with no infractions whatsoever allowed is preferable, because as Occams Razor suggests, the simplest solution is preferable (MacKay 2003).

The threshold optimization algorithm calculates a simple threshold-based effective and efficient classification for a metric set M, with f, M and a set of software entities $\mathbf{X}$ as input for Algorithm 2. Figure 3 shows an example of how a classification



(a) Classification with two thresholds and one allowed infraction

(b) Approximation with only onethreshold and no allowed infractions

Fig. 3 Example for the classification with one threshold infraction allowed

obtained with two metrics and one allowed infraction is approximated by only one of them.

4.5 Learning of Environment Specific Thresholds

An important aspect of thresholds for metrics is that they are often dependent on the properties of the project environment such as the requirements, the developer qualification or the programming language. Therefore, the best results are achieved with thresholds tailored to the specific environment. In the previous two sections, we have only shown how the threshold optimization algorithm can optimize already existing classification methods. However, the algorithm is also able to determine thresholds where currently no method of classification exists. For this, an expert has to select a set of software entities $\mathbf{X}$ that are typical for the project environment. Afterwards, the expert manually classifies them into good and bad based on his or her expertise. As basis for this, the expert may use intricate knowledge, but also information about the software, e.g., the fault history to identify which sections are probably problematic (e.g. Rosqvist et al. 2003). This is the traditional approach to determine the quality of a software, without metric sets and thresholds. Using the thus obtained knowledge, we can determine a metric set with environment specific thresholds that mimics the experts knowledge. To conform to our nomenclature, the expert can be seen as a function f that classifies software. Then, given a metric set M, the threshold optimization algorithm is able to determine an effective, efficient, and environment specific metric set M^* with thresholds T^* that emulates the expert's knowledge.

5 Case Studies

For the validation of the approach for the optimization of metric sets, we performed four case studies consisting of a total of nine experiments. After we describe the general methodology used to perform the case studies, we present the results of the case studies. The case studies were designed to answer the following research questions:

R1: Is the method to optimize the efficiency of metric sets effective?
R2: Is the method to reduce classification complexity effective?
R3: Are the methods applicable and effective to different levels of abstraction (e.g., methods, classes, packages) and programming languages?
R4: Is threshold recalculation with the rectangle learning algorithm necessary or is it sufficient to reuse known thresholds?
R5: Is the exponential nature of the approach a threat to its scalability?

We answer these questions with respect to the case study results in Section 6.1.

5.1 Methodology

The case studies are based on metric data mined from archives of large scale open source software projects. By measuring code checked out from source code repositories, we obtained sets of software entities $\mathbf{X}$ with metric values $M(\mathbf{X})$. To

guarantee the validity of the results, the measured data is randomly split into three disjunctive sets: a *training set* ($\mathbf{X}_{\text{train}}, \mathbf{Y}_{\text{train}}$) that contains 50% of the data; a *selection set* ($\mathbf{X}_{\text{sel}}, \mathbf{Y}_{\text{sel}}$) that contains 25% of the data; an *evaluation set* ($\mathbf{X}_{\text{eval}}, \mathbf{Y}_{\text{eval}}$) that contains 25% of the data. Each of the three sets is used at a different stage of our learning approach. The training set is used to calculate a set of hypotheses $h_{p,q}$ for sampled noise rates $\eta_{0,p}, \eta_{1,q}$ using the rectangle learning algorithm. The selection set is used to select the best of these hypotheses, i.e., an optimal hypothesis h^* with respect to the empirical classification error $\varepsilon_{\mathbf{X}_{\text{sel}},\mathbf{Y}_{\text{sel}}}$. The evaluation set is used to calculate the empirical classification error $\varepsilon_{\mathbf{X}_{\text{eval}},\mathbf{Y}_{\text{eval}}}$ of h^* on data that has not been part of the learning process. The error threshold δ for the threshold optimization algorithm is gradually increased in steps of 0.005 until a set is found that abides the threshold.

By splitting the data into three sets, we ensure that no *overfitting* occurs. Overfitting is the effect that a hypothesis is specific to a training set and not generalized. For example, consider the learning of the structure of credit card numbers based on the sample {1111222233334444, 1234567812345678}. A correct and general assumption is that a credit card number consist of 16 digits. This assumption is also correct on any other learning sample. Therefore, it would also yield a low error—in this case no error at all. Thus, with this hypothesis no overfitting occurs. Another possible hypothesis would be that only the number 1111222233334444 and 1234567812345678 are valid credit card numbers. While this is a correct hypothesis on the training data, the hypothesis is not generalized and would indeed be incorrect for every other credit card number. However, if only the error on the training set is considered, both of the above presented hypothesis are equally good. By splitting the available data, this effect is prevented. Once yet unseen credit card numbers are checked for validity, the first hypothesis still yields the correct results and the error remains zero. However, for the second hypothesis, the error increases with every other credit card number seen, thus making it obvious that the hypothesis is tailored specifically to the training data and invalid in a generalized setting.

To further evaluate the case study results, we employ two additional measures for the quality of a hypothesis. The first is the *Matthews correlation coefficient* (MCC),

Fig. 4 Confusion matrix

		Actual classification	
		positive	negative
Hypothesis	positive	true positive (tp)	false positive (fp)
	negative	false negative (fn)	true negative (tn)

a measure for the quality of binary classifications often used in machine learning (Matthews 1975). It is based on the so called confusion matrix. In the confusion matrix, a hypothesis is compared to the actual values separately for positive and negative samples by counting *true positive* (tp), *true negative* (tn), *false positive* (fp), and *false negative* (fn) classified samples. In Fig. 4, the structure of the confusion matrix is visualized. The MCC is defined as

$$MCC = \frac{tp \cdot tn - fp \cdot fn}{\sqrt{(tp + fp)(tp + fn)(tn + tp)(tn + fn)}}. \tag{5.1}$$

Its value is distributed between −1 and 1, whereas 1 represents a perfect prediction, −1 an inverse prediction, and 0 a random prediction. In contrast to the classification error, the MCC provides a balance between fp and fn predictions. Thus, MCC is more sensitive if the hypothesis has a bias towards rather falsely classifying positive samples than negative ones and vice versa.

The second is the *F-score*, another measure for the quality of hypothesis based on the confusion matrix. It is based on the *precision* and *recall* of a hypothesis. The precision measures how many of the positive predicted values of a hypothesis are actually positives. The recall is a measure for how many of the actual positive values are predicted correctly. They are defined as follows:

$$\begin{aligned} precision &= \frac{tp}{tp + fp} \\ recall &= \frac{tp}{tp + fn}. \end{aligned} \tag{5.2}$$

The F-score is then defined as harmonic mean between prediction and recall:

$$F\text{-}score = \frac{precision \cdot recall}{precision + recall}. \tag{5.3}$$

Its value is distributed between 0 and 1, with 1 being a perfect score and zero being the worst.

5.2 Case Study 1: Optimization of Metric Sets for Methods

In the first case study, we analyzed the methodology for the optimization of metric sets for methods and functions. For this purpose, we measured software from various domains implemented in the languages C, C++, and C#. Hereafter, we use the terms method and function interchangeably.

For C, we measured the *Apache HTTP Server*,[2] an open source HTTP server for Unix/Linux and Windows systems developed and maintained by the Apache Foundation.[3] We measured C++ methods for two of the main components of the *K Desktop Environment* (KDE)[4] for Linux, the *kdebase* and the *kdelibs* components. The kdebase component contains most of the core applications of KDE, e.g., the window manager, an X terminal emulator, and the file manager Dolphin. The kdelibs

[2] http://httpd.apache.org/

[3] http://www.apache.org/

[4] http://www.kde.org/

Table 3 Statistical information about the measured projects

Project name	Version	Language	Number of methods	
			Total	Problematic
(a) Projects used for method-level analysis				
Apache Webserver	2.2.10	C	6718	1995
kdebase	12/05/2008	C++	21404	4161
kdelibs	12/05/2008	C++	37444	4921
AspectDNG	1.0.3	C#	2759	232
NetTopologieSuite	1.7.1.RC1	C#	3059	317
SharpDevelop	2.2.1.2648	C#	15700	1950
Project name	Version	Language	Number of classes	
			Total	Problematic
(b) Projects used for class-level analysis				
Eclipse java development tools	3.2	Java	4833	3349
Eclipse platform project	3.2	Java	5399	3517

provide a library of important core functions that are used by KDE, e.g., for networking, printing, and multi-threading. For C#, we measured three projects. The first C# project measured is *AspectDNG*,[5] an aspect weaver that enables *Aspect Oriented Programming* (Kiczales et al. 2002) in C#. The second is the *NetTopologySuite*,[6] a *Geographic Information System* (GIS) solution for the .NET platform. Finally, we measured *SharpDevelop*,[7] an *Integrated Development Environment* (IDE) for C#, VB.NET, and Boo. Table 3a gives further information about the analyzed versions, as well as the size of the projects.

The aim was to optimize the metric set $M = \{VG, NBD, NFC, NST\}$ with the thresholds defined in Table 2a using Algorithm 2 and the methodology defined in Section 5.1. We used the thresholds defined in Table 2a to determine which source code was problematic. We performed three experiments, one for each programming language. The optimization resulted in a metric set $M^* = \{NFC\}$ with a threshold of $t_{NFC} = 5$ for all three languages. While this threshold value is the same as the one used to classify the data, it has actually been calculated by the rectangle algorithm as the optimal threshold values given the classification for this subset. With only the metric NFC instead of the whole set M, we achieve nearly the same classification, with an empirical error of 0.78%, 0.06%, and 0.59% for C, C++, and C#, respectively. Further analysis, using the MCC and F-score revealed no weaknesses of the classification either. The MCC ranges from 0.9555 for C# to 0.9956 for C++, the F-score is at least 0.9942 for all three languages. Thus, a set with only one metric can to replace a set of four metrics with nearly no loss, i.e. the size of the set can be reduced by 75%. In fact, the error of less than 1% can be interpreted as noise. The Tables 4 and 6a depict detailed results and statistical information about the metrics.

[5] http://aspectdng.tigris.org/

[6] http://code.google.com/p/nettopologysuite/

[7] http://www.icsharpcode.net/

Table 4 This table lists some statistical information about the measured C, C++ and C# methods

Metric	Language	Median	Arithmetic mean	Max value	Threshold
VG	C	2	5.74	734	24
	C++	1	3.09	366	10
	C#	1	2.18	134	10
NBD	C	2	2.15	21	5
	C++	2	1.76	13	5
	C#	3	2.71	11	5
NFC	C	2	6.1	410	5
	C++	2	7.81	997	5
	C#	1	2.44	230	5
NST	C	2	15.61	1660	50
	C++	3	8.33	1132	50
	C#	1	4.78	544	50

5.3 Case Study 2: Optimization of Metric Sets for Classes

In the second case study, we analyzed the optimization of metric sets for Java classes. The basis for this case study are two large-scale open source projects, both run by the Eclipse Foundation:[8] the Eclipse Platform[9] and the Eclipse *Java Development Tools* (JDT).[10] The Eclipse Platform Project defines the main components of the Eclipse Platform, like the handling of resources, the workbench, and the editor framework. For the analysis, we excluded the test code and the *Standard Widget Toolkit* (SWT), a framework for user-interface programming. The rational being, that test code is inherently different from product code and thus test classes should not be compared to other classes. For example, test-cases can be highly repetitive as lists of values have to be compared to expected values, leading to a large size of test classes. On the other hand, the structure of test code should be less complex to prevent errors in the test code itself. The thresholds of the related metrics, like LOC and WMC should therefore be different than for normal code. As for the SWT, while it is formally a part of the Eclipse Platform Project, it is mainly independent. The Eclipse JDT implements an IDE for Java development on top of the Eclipse Platform. Again, we excluded the test code from the analysis. Table 3b shows further information about the used versions and the size of both projects.

The metric set under study was $M = \{WMC, CBO, RFC, NORM, LOC, NOM, NSM\}$ with thresholds as defined in Table 2b in the same manner as in case study 1. The metrics DIT and NOC were initially also part of this set, but we had to exclude them beforehand due to their poor distribution. As for DIT, $\sim 98\%$ of the classes had an inheritance depth of 0 or 1. With the metric NORM another inheritance related measure is still part of the metric set, thus DIT can be excluded without reducing the internal attributes measured. The distribution of NORM is not ideal either, with only $\sim 83\%$ of all values greater than or equal to 2. However, this is still

[8] http:///www.eclipse.org/

[9] http://www.eclipse.org/platform/

[10] http://www.eclipse.org/jdt/

Table 5 Statistical information about the measured Java classes

Metric	Median	Arithmetic mean	Max value	Threshold
WMC	12	27.48	2138	100
CBO	8	13.40	212	5
RFC	20	35.21	675	100
NORM	0	0.96	166	3
LOC	24	82.95	6619	500
NOM	6	10.79	418	20
NSM	0	0.81	128	4

better than the distribution of DIT. The same argument is used to exclude NOC, where $\sim 91\%$ are 0 or 1.

The optimization yielded the set $M^* = \{CBO,\ NORM, NSM\}$, with thresholds $t_{CBO} = 5$, $t_{NOM} = 3$ and $t_{NSM} = 4$. Similar to case study 1, the calculated threshold values are the same as the ones used for the classification. The empirical error of this set is 0.27%. The MCC and F-score reveal no weaknesses either, both have values above 0.99. Therefore, by using the set M^* of size $|M^*| = 3$ instead of M of size $|M| = 7$, the size of the metric set is reduced by 57% without loss of generality. The Tables 5 and 6b depict detailed results and statistical information about the metrics.

5.4 Case Study 3: Reduction of the Classification Complexity for Methods

We performed this case study on the same data as case study 1 (see Section 5.2). The case study is designed to test the capability of the threshold optimization algorithm to reduce the classification complexity. To this aim, we calculated the classification Y for the training using the metric set $M = \{VG, NBD, NFC, NST\}$, thresholds T as

Table 6 Case study results

	M^*	T^*	Error ε	MCC	*F-score*
(a) Case study 1					
Language					
C	{NFC}	{5}	0.78%	0.9793	0.9942
C++	{NFC}	{5}	0.06%	0.9956	0.9986
C#	{NFC}	{5}	0.59%	0.9555	0.9949
(b) Case study 2					
M^*					
{CBO, NORM, NSM}		{5, 3, 4}	0.27%	0.9939	0.9959
(c) Case study 3					
Language					
C	{NST}	{50}	0.84%	0.9274	0.9955
C++	{VG}	{10}	0.87%	0.9139	0.9954
C#	{VG}	{9}	1.36%	0.7598	0.9930
(d) Case study 4					
λ					
1	{RFC, NORM, NOM, NSM}	{98, 3, 20, 4}	1.71%	0.9449	0.9894
2	{WMC, RFC}	{99, 110}	2.21%	0.8494	0.9880

defined in Table 2a, and $f_1(\cdot, M, T)$ (see (4.4)) to calculate the classification. Thus, an entity is only considered problematic if the threshold of more than one metric is violated.

In contrast to case study 1, the result is different for the various languages. In case of C, the metric NST with a threshold of $t_{NST,C} = 5$ yields the best result with an empirical error of 0.84%. For C++ and C#, the metric VG with thresholds $t_{VG,C++} = 10$ and $t_{VG,C\#} = 9$ performs best with an empirical error of 0.87%, respectively 1.36%. The calculated threshold value in the C# experiment is different to the one used in the initial classification, while remains the same in the C and C++ experiments. The MCC revealed no weakness for the C and C++ experiments. However, in the C# experiment, the MCC dropped to 0.7598. While this is still a very good value, it indicates a possible weakness of this result. The F-score revealed no such weakness and was above 0.9930 for all three languages. Thus, we were able to use a simpler classification methodology, while also reducing the size of the metric set by 75% for all three languages. Table 6c summarizes the results of this case study.

5.5 Case Study 4: Reduction of the Classification Complexity for Classes

We performed the fourth case study on the same data as case study 2 (see Section 5.3). Like case study 3, it is designed to test the capability to reduce classification complexity. The methodology is similar to the one used in case study 3. Again, we use f_λ instead of f_0 for the classification of software entities. Here, we use $\lambda = 1, 2$, i.e., we perform two experiments: 1) one threshold violation allowed; 2) two threshold violations allowed. Allowing more infractions would render the metric set ineffective, as more than half of the thresholds would have to be violated to even classify a class as problematic.

In both experiments, we determined effective and efficient metric sets. In the first experiment, with one violation allowed, the metric set $M^* = \{RFC, NORM, NOM, NSM\}$ with thresholds $t_{RFC,1} = 98$, $t_{NORM,1} = 3$, $t_{NOM,1} = 20$, and $t_{NSM,1} = 4$ performed best with an empirical error of 1.71%. In the second experiment, the metric set $\{WMC, RFC\}$ with thresholds $t_{WMC,2} = 99$ and $t_{RFC,2} = 97$ was effective and efficient with a classification error of 2.21%. Half of the threshold values calculated in this case study deviated from the ones used for the classification. While the empirical error of the experiment with $\lambda = 1$ was higher than with $\lambda = 2$, the MCC performed the other way around. While the MCC of the experiment with $\lambda = 1$ is unproblematic with 0.9449, it drops slightly for $\lambda = 2$ to 0.8494. This suggest, that the hypothesis in the second experiment has a slight bias towards positive samples, as the F-score revealed no such weakness. It is above 0.98 for both experiments. The results show that a simpler classification can be used in both cases and, furthermore, the metric set sizes can be reduced by 42% and 71%, respecitvely. Table 6d summarizes the results of this case study.

6 Discussion

In this section, we discuss the research questions R1–R5 with respect to the case study results. Afterwards, we discuss other methods for metric set optimization and compare them to our methodology.

6.1 Discussion of Research Questions

R1: Is the method to optimize the efficiency of metric sets effective? The results of the three experiments of case study 1 and the experiment performed in case study 2 show that the methodology is capable of decreasing the size of metric sets between 57% and 75% without a significant loss of classification precision. Based on these four successful experiments, each of them performed in a different environment, the answer to this research question is yes.

R2: Is the method to reduce classification complexity effective? In case studies 3 and 4 we classified the data with a method more complex than the simple threshold classification. A total of five experiments were performed, in all of which simple thresholds were sufficient to reproduce the original classification. Furthermore, the resulting metric sets were also 42% to 85% smaller than the ones used for the classification. Thus, the answer to this research question is yes.

R3: Are the methods applicable and effective to different levels of abstraction (e.g., methods, classes, packages) and programming languages? In the case studies 1 and 3, we analyzed methods and functions, while classes were under consideration in case studies 2 and 4. Thus, the approach does not depend on the level of abstraction. Furthermore, in the case studies, we used projects written in four different programming languages: C, C++, C# and Java. These four languages cover the procedural and the object-oriented paradigm. Moreover, *C* is a low-level and close to the system programming language, whereas Java and C# are relatively high level. Therefore, the results indicate that the programming language has no impact on the capabilities of the methodology and the answer to this question is yes.

R4: Is threshold recalculation with the rectangle learning algorithm necessary or is it sufficient to reuse known thresholds? On one hand, the results of case studies 1 and 2 suggest that recalculation of threshold values is not required when optimizing a metric set. In all experiments conducted, the calculated threshold values were the same as the original ones. On the other hand, the results of case study 3 and 4 suggest that when the classification method is changed, recalculation of threshold values is beneficial even if the formerly used method is based on thresholds. In addition to the problems analyzed in the case studies, there are possible applications where no thresholds are available, e.g., if a non-threshold based classification method is to be optimized. In such cases threshold calculation is integral and may not be omitted. In conclusion, whether the recalculation of threshold adds value to the proposed method depends on the application of the method.

R5: Is the exponential nature of the approach a threat to its scalability? The execution of all nine experiments performed as part of the four case studies took 139 seconds in total on a normal desktop workstation running on an Intel Core2 Duo E8400 processor. For these experiments, the rectangle learning algorithm was executed a total of 480 times, therefore, a single execution took about 0.29 seconds in average. As there are 2^{20} different subsets of a metric set of size 20, the execution would take $2^{20} \cdot 0.29 \approx 304.000$ seconds, thus, approximately 3.5 days. While this is a pretty long time, it has to be taken into account that such an optimization must only be performed once and does not need to run regularly. Furthermore, run time can be

reduced by using multiple parallel threads of execution. Of course, with even greater metric sets, this does not resolve the problem. In conclusion, it can be said that the approach is able to handle metric sets with a size of about 20 in an acceptable amount of time. For larger metric sets, a heuristic for the selection of subsets to be analyzed needs to be employed.

6.2 Comparison to Other Methods

One of the main features of the presented methods is the reduction of the number of metrics required for the classification and, therefore, the dimension of the space spanned by the metric set. In the following, we discuss the advantages of our method compared to two other techniques: 1) correlation based methods; 2) the brute-force risk minimization approach presented by Werner et al. (2007).

Correlation based techniques analyze the input variables, i.e., the metrics and determine whether their values are correlated. If so, one of the variables may be removed without effect or a new variable can be defined based on the correlated variables. Examples for correlation based reduction techniques are *Principle Component Analysis* (PCA) and *Factor Analysis* (FA). These techniques are similar to each other. Therefore, we discuss only *Principle Component Analysis* (PCA) here. The results of the discussion are transferable to FA.

The general idea of PCA is to linearly transform the input space, i.e., the space of metric values. The transformed space is such that only few dimensions contain most of the data's variance. This is done by determining *components* c as linear combination of the metrics, i.e., $c = \lambda_1 m_1 + \lambda_2 m_2 + \ldots + \lambda_d m_d$. The first component contains the maximum of the variance that can be achieved using a linear transformation. The second component contains the maximum of the remaining variance, and so on. Thus, the first components contain most of the variance. By using only these components, the dimension of the input space is reduced. In terms of metrics, the components can be thought of as indirect metrics based on the original ones, e.g., $c = 0.2 \cdot WMC + 0.3 \cdot RFC + 0.5 \cdot LOC$. In comparison to single metrics, the components are difficult to interpret as they are influenced by several metrics at once and the nature of their relationship is unclear.

A major disadvantage of such techniques is that the usage of only few components does not guarantee that the number of metrics can actually be reduced. In an extreme case, a single component can rely on *all* input variables. This one component can be sufficient, however, the number of metrics remains the same. Another drawback of using PCA is that the variance is not necessarily a good criterion for the selection of features. For example, the metric LOC for classes has a high variance due to its nature. Its values are distributed on a rather large scale and classes tend to be rather variable in their size. However, this large variance does not mean that LOC is suited for quality prediction, as in the end only threshold violations matter. Therefore, variance is a misleading criterion.

The third drawback is a rather general one. By first determining metrics using PCA and then thresholds in a second step, two locally optimal results are calculated. The PCA determines a reduced metric set, e.g., M_{PCA}, which is optimal in terms of the criteria PCA uses. This metric set is then used to determine thresholds T_{PCA}. The thresholds are optimal for M_{PCA} but this is not necessary the globally optimal result. There can be another metric set M^* with thresholds T^* that yield better results, but

Table 7 Number of threshold combinations using Werner et al. (2007)s method

Max. no. of metrics	No. of threshold combinations	Calc. time assuming 0.1 ms per hypothesis
1	1,415	141.5 ms
2	629,076	~ 63 s
3	149,235,857	~ 248 min
4	18,565,376,659	~ 21.5 days
5	1,201,532,717,441	~ 3.8 years
6	37,125,301,717,441	~ 117 years
7	438,665,979,997,440	~ 1391 years

which is not discovered. In contrast, the approach defined in this article combines the metric set selection with the threshold optimization and finds a globally optimal value.

The method to optimize metric sets presented by Werner et al. (2007) is in principle the same as the one presented in this article. They considered all combinations of metrics up to a desired maximum size. However, the threshold calculation method is a simple brute force approach. The algorithm evaluates *all* possible threshold combinations. This means that they consider every value of metric m that occurs in the training data as a possible threshold value t_m and all combinations of these values are tested. Afterwards, the optimal one is selected. This exhaustive search of the hypothesis space guarantees the best possible result. However, this method of calculation is extremely calculation time intensive, as the number of possible hypotheses grows exponentially. This renders the method infeasible for larger metric sets, as demonstrated by calculating the number of possible hypotheses for the metric set used in case study 2. The seven metrics used in case study 2 had each between 28 and 525 different possible threshold values. In Table 7, the number of threshold combinations as well as estimated calculation times are listed. As can be seen, this method is infeasible for larger sets. The same experiment that was performed in case study 2 would take over 1000 years, even assuming a small calculation time per hypothesis.

6.3 Limitations

We only analyzed open source software in the case studies, non-open-source software has not been analyzed. However, the work by Werner et al. (2007) showed that a similar approach worked with TTCN-3 test suites, i.e., software written in a *Domain Specific Language* (DSL) in a non-open-source environment.

The metric sets we analyzed only consist of internal product metrics on the method and class level. Metric sets on higher levels of abstraction, as well as metric sets including process or resource metrics have not been analyzed. Furthermore, the chosen threshold values may have been inadequate to begin with, leading to misclassified training data.

The proposed methodology produces a binary classification and can therefore only differentiate between “good” and “bad”, further shades of grey are not possible.

7 Related Work

Research on how environment specific metric sets can be obtained was performed by Basili and Selby (1985). In contrast to this work, the authors use a *Goal/Question/Metric* (GQM) (Basili and Weiss 1984; Basili and Rombach 1988) approach to determine a metric set and condense it using factor analysis. A statistical method to obtain threshold values was introduced by French (1999) who used it to derive thresholds for object-oriented and procedural software.

An approach to determine classification trees to identify quality critical modules was proposed by Porter and Selby (1990), Selby et al. (1991). The tree makes its decisions based on intervals of metric values, which is similar to using thresholds.

A methodology to determine metric sets to predict quality critical modules using Boolean Discriminant Functions (BDFs) has been introduced by Schneidewind (1997), Schneidewind (2000). The BDFs consist of boolean disjunctions of threshold violations to identify critical modules, which is just another formalization of the classification model used in this work. They determine the thresholds using Kolmogorov–Smirnov tests (Lilliefors 1967). This model is extended to Generalized BDFs by introducing conjunctions into the boolean functions (Khoshgoftaar 2002). This is similar to the more complex classification used in the case studies 3 and 4, where one threshold *and* another need to be violated.

Lanza et al. (2005) use environment specific thresholds to determine whether metric values are low, average, or high, based on the arithmetic mean and the standard deviation of observed metric data. These thresholds are then used in an *overview pyramid* to provide an overview of object-oriented software based. The metrics are divided into three aspects: inheritance; size and complexity; coupling. Using the thresholds, a coloring scheme is defined that visualizes the software properties. In comparison to this work, the authors do not assume thresholds to define metric values as problematic, but rather use them to discriminate metric values into low, average, and high values.

An instantiation of the maintainability characteristic of the ISO 9126 quality model (ISO/IEC 2001–2004) is described by Heitlager et al. (2007). They use both internal and external product metrics to define ratings for the source code properties *volume*, *complexity per unit*, *duplication*, *unit size*, and *unit testing*. Based on the property ratings, the sub-characteristics of maintainability are rated from which maintainability is inferred. The ratings are based on intervals, which are similar to using thresholds. In comparison to our work, they have five rating classes instead of a binary classification. Furthermore, very good ratings for one property allow bad ratings for another, which is different to the strict threshold classification we apply.

A paper similar to this work, but using a less sophisticated approach for the optimization of metric sets for TTCN-3 is presented by Werner et al. (2007). However, the machine learning methodology used in this work is more mature and the case studies analyze it in a wider setting, i.e. various programming languages and levels of abstraction. For a detailed comparison, see Section 6.2.

In Lorenz and Kidd (1994) the authors define thresholds for many object-oriented metrics, however, they do not validate their proposals. An overview of work on thresholds for the object-oriented Chidamber and Kemerer metrics suite is provided by Benlarbi et al. (2000).

8 Conclusion

We defined a novel high-level approach for the calculation of thresholds for software metrics to evaluate quality attributes. The method is purely data driven and utilizes machine learning techniques. Based on this, we defined a methodology to determine optimized metric sets that replicate a given classification of a quality attribute. We outlined how the methodology can be applied to improve the efficiency of existing metric sets with thresholds, reduce the complexity of a used classifier and how a new metric set can be introduced using the methodology. In two case studies, we showed that the methodology is able to greatly improve the efficiency of existing metric sets. In two further case studies, we reproduced complex classifications successfully with simple thresholds.

Future projects may include more case studies, on how well the approach works in other environments, e.g., domain specific languages or how well it handles sparse data. Moreover, it may be investigated how learning of *Disjunctive Normal Form*s (DNFs) of thresholds instead of conjunctions affects the hypothesis quality, the metric set reduction, and the interpretability of the resulting classifiers. Furthermore, a detailed comparison with black-box classification techniques like *Artificial Neural Network*s (ANNs) or SVMs is an interesting topic for the future. Another research direction is to determine metric sets and thresholds that can be used to steer software project decisions. To this aim, the approach needs to be adapted for process data.

Appendix: Glossary

$\mathbb{R}^d$	d-dimensional real-space
g	target concept
$\mathscr{C}$	concept class
$\mathfrak{X}^d$	input space, defined over the d-dimensional real-space $\mathbb{R}^d$
$U = (X, Y)$	learning sample, $U \in \mathfrak{X}^d \times \{0, 1\}$
X	input element
Y	output element, random label
$\mathscr{D}$	sample distribution
$EX(\mathscr{D}, g)$	oracle
S	random noise
η	noise rate
$\eta(X)$	random noise rate
η_0, η_1	conditional expected noise rates
$\mathbb{P}$	probability
$\mathbb{E}$	expected value
χ	query function
M	a set of metrics $\{m_1, \ldots, m_n\}$
$(\mathbf{X}, \mathbf{Y})$	a discrete learning sample

$(\mathbf{X}_{\text{train}}, \mathbf{Y}_{\text{train}})$	a training set used to calculate hypotheses
$(\mathbf{X}_{\text{sel}}, \mathbf{Y}_{\text{sel}})$	a selection set used to select the best hypothesis
$(\mathbf{X}_{\text{eval}}, \mathbf{Y}_{\text{eval}})$	an evaluation set, used to evalute the quality of the selected hypothesis
$M(X)$	the transformation of software entities into the n-dimensional real-space using M $M(\mathbf{X}) := \{(m_1(x), \ldots, m_n(x))^t : x \in \mathbf{X}\} \subset \mathbb{R}^n$
(l_i, u_i)	the pair of lower and upper bound of an axis aligned rectangle in the i-th dimension
t_i	threshold value for the metric m_i
$f_0(x, M, T)$	classification of a software entity x using a metric set M with thresholds T
$f_\lambda(x, M, T)$	classification of a software entity x using a metric set M with thresholds T, with λ allowed infractions.
ε	classification error
$\varepsilon_{\mathbf{X},\mathbf{Y}}$	empirical classification error
$\mathscr{P}(M)$	power set of the set M
$\mathscr{X}$	space of software entities, e.g. methods or classes
h^*	optimal hypothesis
M', T'	a subset $M' \subseteq M$ with thresholds T'
M^*, T^*	a subset $M^* \subseteq M^*$ with thresholds T^* that is optimal, i.e., the smallest subset that is effective.

References

Angluin D, Laird P (1988) Learning from noisy examples. Mach Learn 2(4):343–370. doi:10.1023/A:1022873112823

Basili V, Rombach H (1988) The TAME project: towards improvement-oriented software environments. IEEE Trans Softw Eng 14(6):758–773

Basili V, Weiss D (1984) A methodology for collecting valid software engineering data. IEEE Trans Softw Eng 10(6):728–738

Basili VR, Selby RW Jr (1985) Calculation and use of an environment's characteristic software metric set. In: ICSE '85: proceedings of the 8th international conference on Software engineering. IEEE Computer Society Press, Los Alamitos, CA, USA, pp 386–391

Basili VR, Briand LC, Melo WL (1996) A validation of object-oriented design metrics as quality indicators. IEEE Trans Softw Eng 22(10):751–761. doi:10.1109/32.544352

Benlarbi S, Emam KE, Goel N, Rai S (2000) Thresholds for object-oriented measures. In: ISSRE '00: proceedings of the 11th international symposium on software reliability engineering. IEEE Computer Society, Washington, DC, USA, p 24

Brodag T, Herbold S, Waack S (2010) A generalized model of pac learning and its applicability. Mach Learn (manuscript in revision)

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493. doi:10.1109/32.295895

Copeland T (2005) PMD applied

Daly J, Brooks A, Miller J, Roper M, Wood M (1996) Evaluating inheritance depth on the maintainability of object-oriented software. Empir Softw Eng 1(2):109–132

Devroye L, Györfi L, Lugosi G (1997) A probabilistic theory of pattern recognition. Springer, New York

Duda R, Hart P (1973) Pattern classification and scene analysis.

ETSI (2007) ETSI Standard (ES) 201 873-1 V3.2.1 (2007-02): the testing and test control notation version 3; part 1: TTCN-3 core language. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, also published as ITU-T Recommendation Z.140

Fenton N, Pfleeger S (1997) Software metrics: a rigorous and practical approach. PWS Publishing Co. Boston, MA, USA

French V (1999) Establishing software metric thresholds. In: International workshop on software measurement (IWSM99)

Grabowski J, Hogrefe D, Réthy G, Schieferdecker I, Wiles A, Willcock C (2003) An introduction to the testing and test control notation (ttcn-3). Comput Netw 42(3):375–403. doi:10.1016/S1389-1286(03)00249-4

Heitlager I, Kuipers T, Visser J (2007) A practical model for measuring maintainability. In: 6th international Conference on the Quality of information and communications technology, 2007. QUATIC 2007, pp 30–39. doi:10.1109/QUATIC.2007.8

IEEE (1990) Ieee glossary of software engineering terminology. ieee standard 610.12. Tech. rep., IEEE

ISO/IEC (2001–2004) ISO/IEC standard no. 9126: software engineering—product quality; parts 1–4. International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC), Geneva, Switzerland

ISO/IEC (2005) ISO/IEC Standard No. 9000. International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC), Geneva, Switzerland

Kearns M (1998) Efficient noise-tolerant learning from statistical queries. J ACM 45(6):983–1006. doi:10.1145/293347.293351

Khoshgoftaar TM (2002) Improving usefulness of software quality classification models based on boolean discriminant functions. In: ISSRE '02: proceedings of the 13th international symposium on software reliability engineering. IEEE Computer Society, Washington, DC, USA, p 221

Kiczales G, Lamping J, Lopes C, Hugunin J, Hilsdale E, Boyapati C (2002) Aspect-oriented programming. US Patent 6,467,086

Lanza M, Marinescu R, Ducasse S (2005) Object-oriented metrics in practice. Springer-Verlag New York, Inc., Secaucus, NJ, USA

Lilliefors HW (1967) On the Kolmogorov–Smirnov test for normality with mean and variance unknown. J Am Stat Assoc 62(318):399–402. http://www.jstor.org/stable/2283970

Lorenz M, Kidd J (1994) Object-oriented software metrics: a practical guide. Prentice Hall PTR

MacKay DJ (2003) Information theory, inference, and learning algorithms. Cambridge University Press

Mammen E, Tsybakov AB (1999) Smooth discrimination analysis. Ann Stat 27(6):1808–1829

Matthews BW (1975) Comparison of the predicted and observed secondary structure of t4 phage lysozyme. Biochim Biophys Acta, Protein Struct 405(2):442–451. doi:10.1016/0005-2795(75)90109-9. URL:http://www.sciencedirect.com/science/article/B73GJ-47T22GD-132/2/b5b0dbd824d44e6edeebf7b8d2613775

Nagappan N, Ball T, Zeller A (2006) Mining metrics to predict component failures. In: ICSE '06: proceedings of the 28th international conference on software engineering. ACM, New York, NY, USA, pp 452–461. doi:10.1145/1134285.1134349

Porter AA, Selby RW (1990) Empirically guided software development using metric-based classification trees. IEEE Softw 7(2):46–54. doi:10.1109/52.50773

Quinlan JR (1986) Induction of decision trees. Mach Learn 1:81–106. doi:10.1007/BF00116251

Rosqvist T, Koskela M, Harju H (2003) Software quality evaluation based on expert judgement. Softw Qual J 11:39–55. doi:10.1023/A:1023741528816

Schneidewind NF (1997) Software metrics model for integrating quality control and prediction. In: ISSRE '97: proceedings of the eighth international symposium on software reliability engineering. IEEE Computer Society, Washington, DC, USA, p 402

Schneidewind NF (2000) Software quality control and prediction model for maintenance. Ann Softw Eng 9(1–4):79–101. doi:10.1023/A:1018920623712

Schölkopf B, Smola AJ (2002) Learning with kernels. MIT Press

Selby RW, Porter AA, Schmidt DC, Berney J (1991) Metric-driven analysis and feedback systems for enabling empirically guided software development. In: ICSE '91: proceedings of the 13th international conference on software engineering. IEEE Computer Society Press, Los Alamitos, CA, USA, pp 288–298

Shawe-Taylor J, Cristianini N (2004) Kernel methods for pattern analysis. Cambridge University Press

Tsybakov AB (2004) Optimal aggregation of classifiers in statistical learning. Ann Stat 32(1):135–166

Werner E, Grabowski J, Neukirchen H, Rottger N, Waack S, Zeiss B (2007) TTCN-3 quality engineering: using learning techniques to evaluate metric sets. Lect Notes Comput Sci 4745:54

Steffen Herbold is a doctoral student at the Institute for Computer Science at the Georg-August University of Göttingen. He received his B.Sc. and M.Sc. degree in Applied Computer Science from the Georg-August University of Göttingen. He is interested in algorithmic learning methods and their applications in software quality assurance.

Jens Grabowski is professor at the Institute for Computer Science at the Georg-August University of Göttingen in Germany, where he is head of the Software Engineering for Distributed Systems research group. His research interests include software engineering processes, modeling, testing, and quality engineering.

Stephan Waack is professor and head of the theory group of the Institute for Computer Science at the Georg-August University of Göttingen. In particular, he is interested in applying algorithmic methods and learning techniques to software quality management and software testing as well as to problems from bioinformatics and business informatics.