Calculation and Optimization of Thresholds for Sets of Software Metrics

Technical Report No. IFI-TB-2010-01, ISSN 1611-1044

Steffen Herbold, Jens Grabowski, and Stephan Waack

Institute of Computer Science, University of Goettingen, Goldschmidtstraße 7, 37077 Göttingen, Germany
email: {herbold,grabowski,waack}@cs.uni-goettingen.de

Abstract. Many software quality models use software metrics to determine quality attributes of software products. A common practice is to use sets of software metrics with threshold values for the classification of a quality attribute. In this article, we present a novel approach for the calculation of thresholds for a metric set that can be used for efficiency optimization of existing metric sets, for the simplification of complex classification models and the calculation of thresholds for a metric set in an environment where not metric set yet exists. sets, but also to determine thresholds for new metric sets.

1 Introduction

Software has become part of our everyday life, be it embedded in modern cars to control the distance to the next car or the daily news read on the Internet. As users of the software we expect it to fulfill 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 (ISO/IEC, 2005). To uphold the required standard of quality, the assessment and assurance of software quality characteristics are an import part of the execution of software projects. In many models to assess the quality of software, like the ISO 9126 standard (ISO/IEC, 2001-2004), the assessment of complex high-level quality attributes, like the maintainability or the understandability, is on a lower level of the model performed using *software metrics*. Software metrics provide means to put numbers on abstract attributes, e.g., the complexity or the size. To effectively analyze quality attributes, often not only one metric is used, but a set of metrics. These metric sets are then used to determine whether a quality attribute is fulfilled or if not, problematic. A common and easy to understand way for this kind of classification is to use *thresholds* for metric values. In this case a quality attribute of software is determined to be problematic, if the metrics that are part of the set violate the defined thresholds.

The contribution of this paper is an algorithmic approach for the optimization of software metric sets and thresholds. For this, we utilize machine learning. During the last years, using machine learning techniques for data analysis has become very popular and it has been successfully applied in many different fields, e.g., gene analysis in biology or data mining techniques companies use to optimize their marketing. It has also

been used in computer science, e.g., for defect prediction. We use a machine learning algorithm to define an approach for the calculation of thresholds for a metric sets. In a previous work, we used a relatively simple brute-force approach for the calculation of threshold values for a metric set (Werner et al, 2007) for the *Testing and Test Control Notation* (TTCN-3) (ETSI, 2007; Grabowski et al, 2003). In this work, we use a more sophisticated approach, that utilizes the learning of axis-aligned d-dimensional rectangles for the threshold calculation. We then show how our approach can be used in three ways: the optimization of existing effective metric sets to be more efficient; the reduction of the complexity of the classification method used to a simple threshold-based approach; the calculation of thresholds for a completely new metric set.

The structure of this paper is as follows. In section 2 the concepts of software metrics and how they can be used in combination with thresholds for quality estimation are introduced. Afterwards, we give a brief introduction into machine learning and define the foundations of the learning approach we use in this paper in section 3. In section 4 we define our approach for the optimization of software metric sets with thresholds and describe how it can be applied to perform different tasks. The applicability and effectiveness of our approach is validated in two case studies, we present in section 5. Afterwards, we discuss related work in section 6. Finally, in section 7 we discuss our results and conclude our paper.

2 Foundations of 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 entities, that are part of a software system, e.g., components, classes or methods.

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. On the other hand, external attributes are those, that rather relate the behavior. In this work, we focus 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 method complexity, cohesion or attributes that relate to object-oriented software, such as usage of inheritance.

Table 1. Below the metrics that were measured as part of this work are listed. All of the listed metrics are product metrics, that measure internal attributes.

	Metric name	**Internal attribute**
Metrics	*Cyclomatic Number* (VG)	Control-flow structuredness
for	*Nested Block Depth* (NBD)	Control-flow structuredness
methods	*Number of Function Calls* (NFC)	Coupling
and	*Number of Statements* (NST)	Size
functions	*Weighted Methods per Class* (WMC)	Method Complexity
Metrics	*Coupling Between Objects* (CBO)	Coupling
for classes	*Response For a Class* (RFC)	Coupling
	Number of Overridden Methods (NORM)	Inheritance
	Number of Methods (NOM)	Size
	Lines of Code (LOC)	Size
	Number of Static Methods (NSM)	Staticness

2.1 Metric definitions

In our work, we want to focus on product metrics, that measure source code. We used 4 metrics to analyze methods and functions and 7 metrics for the analysis of classes. In table 1 the metrics that we used in our work are listed and categorized according to the internal attributes that they measure. For the analysis of methods and functions, we selected the metrics *Cyclomatic Number* (VG), *Nested Block Depth* (NBD), *Number of Function Calls* (NFC) and *Number of Statements* (NST). For our class analysis, we planned to use the whole metric suite proposed by (Chidamber and Kemerer, 1994), consisting of the metrics *Weighted Methods per Class* (WMC), *Coupling Between Objects* (CBO), *Response For a Class* (RFC), *Depth of Inheritance Tree* (DIT), *Number of Children* (NOC) and *Lack of Cohession in Methods* (LCOM). However, we excluded the latter three of these metrics, due to various reasons. The metric *Depth of Inheritance Tree* (DIT) was poorly distributed, over 97% of the classes we measured had a value for DIT of 0 or 1. NOC is only a measure for the reuse of a class and is therefore irrevelant for our aim to determine its quality. Finally, we excluded LCOM, because it was found to be poorly distributed (Basili et al, 1996). In addition to the Chidamber and Kemerer metrics, we used the metrics *Number of Overridden Methods* (NORM), *Number of Methods* (NOM), *Lines of Code* (LOC) and *Number of Static Methods* (NSM) for the evaluation of classes.

2.2 Metrics for functions and methods

In this sections, we will introduce the metrics we used that can be measured on functions and methods. Since the four metrics that we describe do not take object-orientation or any of the properties that distinguish methods from functions into account, we will hereafter only speak of methods.

***Nested Block Depth* (NBD)** The metric NBD measures the structural complexity of a method through the maximum nested block depth. A structural block of source codes

is usually defined by condiditional statements, through nested conditional statements nested blocks are created. Figure 1 shows a short example of a Java method with a maximum nesting level of 2. . The block with the depth 0 is the method definition itself. The first nested block is defined by the `while` loop, in this block two blocks are nested at a level of 2, defined by the `if` and the `else` statement. In both the `if` and the `else` block, no further block is nested, therefore the maximum nesting level of the method is 2.

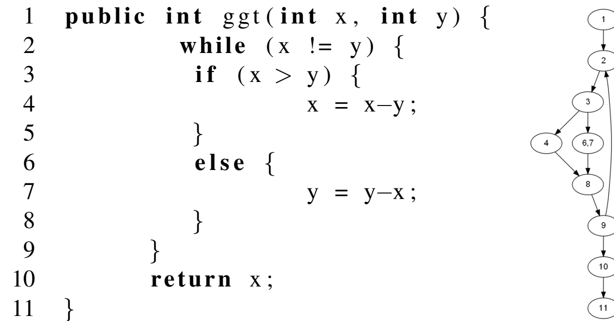

Fig. 1. Above, a simple Java method and its control flow graph are shown, which is used as an example for the metrics NBD, VG and NST

The *Cyclomatic Number* (VG) Similar to the metric NBD, the metric VG measures the structural complexity of a method. The metric was first proposed by (McCabe, 1976) and is hence also commonly known as McCabe complexity. While NBD only measures the maximum level of nested blocks, VG measures the branching of paths in the control-flow. Each branching increases the value of VG. To measure VG, the control flow graph of a method is utilized. Let $G = (V,E)$ the control flow graph of a method M with vertices V and edges E. Let p the number of the points of entry and exit of M, then

$$VG(M) = |E| - |V| + p. \tag{2.1}$$

Figure 1 shows the listing of a method and its associated control flow graph. It has $|E| = 10$ edges, $|V| = 9$ vertices and one point of entry as well as one point of exit, therefore $p = 2$. Thus, the value of VG is 3.

***Number of Function Calls* (NFC)** An aspect different to the control-flow structuredness is the coupling. Coupling is the degree of interdependence between modules (Yourdon and Constantine, 1979). The metric NFC is a measure for the coupling of methods. The value of NFC is the sum of all methods calls of a method. If a method is called more than once, each of these calls is counted. As a method depends on each method that it calls, NFC is a measure for the coupling. Furthermore, it could also be understood as a measure for the complexity of a method, since the complexity increases with

each method call, due to the fact that at least the parameters and return value have to be understood.

***Number of Statements* (NST)** A third attribute of methods is their size. The metric NST measures the size of a method by counting the number of its statements. Therefore, it is independent of how the source code is formated, which makes it more robust than the metric LOC, which is also a commonly used measure for the size of methods. The value of NST for the method shown in figure 1 is 6 and the statements are: the `while(x!=y)` statement in line 2; the `if(x>y)` statement in line 3; the two computational statements `x=x-y;` and `x=y-x` in the lines 4 and 7; the `else` statement in line 6; the `return` statement in line 10.

2.3 Metrics for classes

In this section, we want to introduce the metrics for classes we used. The metrics do not nessecarily measure attributes that are associated with object orientation, but also general attributes like the size. First, we will describe the metrics of the Chidamber an Kemerer metric suite that we use (Chidamber and Kemerer, 1994). Then we will introduce 4 metrics, we use in addition to the Chidamber and Kemerer metrics.

***Weighted Methods per Class* (WMC)** WMC is a measure for the complexity of a class. The complexity of a class is measured indirectly using the sum of the complexity of its methods. For a class C that defines the set of methods M WMC is computed as

$$WMC(C) = \sum_{m \in M} complexity(m) \tag{2.2}$$

where $complexity(\cdot)$ is a measure for the complexity of a method. In our work, we choose VG as the underlying complexity metric.

***Coupling Between Objects* (CBO)** As has been said, coupling describes the interdependence between modules. In case of classes, we say that two classes are coupled if at least one of them uses the other class. The metric CBO measures the coupling of a class by counting the classes, with which it is coupled.

***Response For a Class* (RFC)** This metric measures the size of the *response set* of a class. The response set consists of all methods that can be invoked by calling a method from the class. Chidamber and Kemerer suggest, that only one level of nesting is considered. Therefore, for a class C only the methods that are called by methods of C directly are part of the response set. Obviously, the response set of a class includes all of its public methods and usually also of all its methods with a lower visibility. Otherwise it is an indicator for poor design or dead code.

***Number of Overridden Methods* (NORM)** The metric NORM counts the number of methods, that are overridden by a class., i.e., already defined by one of its parent classes.

Lines of Code **(LOC)** LOC is probably the most intuitive, most used and most controversly discussed metric there is. It measures the size of entities by counting its lines. Of course this is not only restricted to classes, but can be applied to all kinds of documents. In our work, we used LOC as a measure for the size of classes. We counted only lines, that were neither empty nor contained only comments.

Number of Methods **(NOM)** Another measure for the size of a class is defined by the metric NOM. It counts the number of methods defined by a class. The rational behind this metric is, that classes with more methods are larger.

Number of Static Methods **(NSM)** A metric that measures an attribute unrelated to all of the above is NSM. It counts the number of static methods defined by a class and thereby measures its staticness.

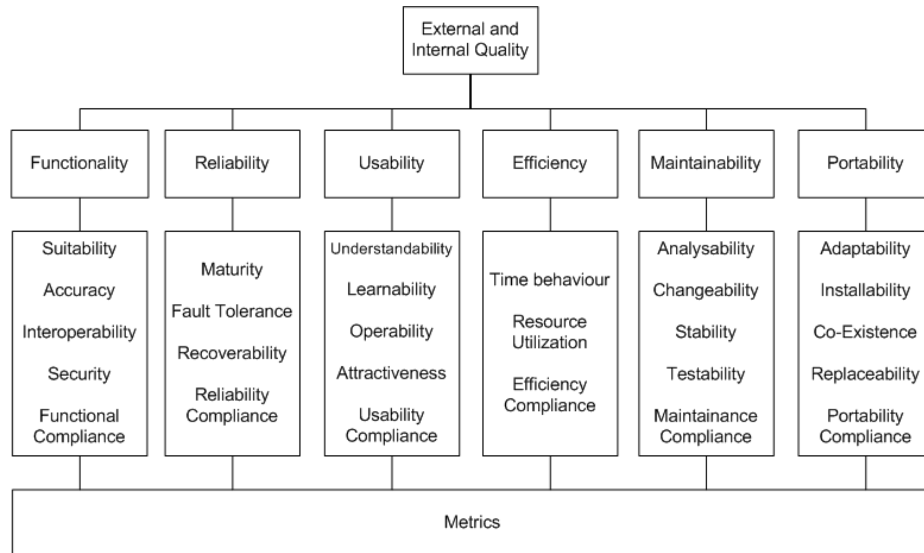

Fig. 2. This figure illustrates the part of the ISO 9126 standard that is responsible for internal and external quality. The quality is described using 6 *characteristics*, which have *sub-characteristics*. The fulfillment of the sub-characteristics is determined using metrics.

2.4 Thresholds for software metrics

In general, thresholds are a simple method to separate values. The values that are greater than a threshold value are considered to be problematic, the values below are okay. Thus, by defining thresholds a simple analysis of measured values is possible. This

mechanism can also be applied to software metrics. For example, by defining a threshold for a metric that measures the size of an entity, all metric values that are above the threshold mark the entity as too large. Thresholds for software metrics are often used in the context of fault-proness. This means that a measured entity is less fault-prone, if it violates a threshold. Even so, thresholds can also be used to define other aspects as problematic, e.g., the maintainability or the understandability. For simplicity, we assume that thresholds are upper bounds. However, this is no restriction. Let m a metric with threshold t that defines a lower bound, i.e., entites x are considered to be problematic if $m(x) < t$. This 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.

There are a few problems with thresholds. The first is the generality of threshold values. It cannot be said that a given threshold value is good in every setting. Depending on the organization, the programming language, the tools used, the qualification of the developers and other factors that are project dependent a different threshold value might be better. This is a problem, as each organization – and maybe even each project – has to define thresholds that are choosen 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. In section 4.4 we will show how thresholds can be computed based on previous experience.

Table 2. This table shows the threshold values that were used in this work. If possible, we based our thresholds on previous research. For those that were adapted the source is given in the last column.

	Metric name	Programming language	Threshold	Source
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	-
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	-
	NOM	Java	20	-
	NSM	Java	4	(Lorenz and Kidd, 1994)

In table 2 thresholds are given for the metrics we use in this work. Most of the thresholds are taken from previous work by (Lorenz and Kidd, 1994; French, 1999; Benlarbi et al, 2000). For the metrics NFC and NST, we did not find reference values in the literature, therefore we defined them ourselves with values we think are reasonable. The thresholds we used for the metrics LOC and NOM are based on thresholds that are used by PMD (Copeland, 2005). For the lines of code including empty and comment-only lines PMD uses a threshold value of 1000. We adapted this value to 500 lines for our definition of LOC, in which only lines that contain source code statements are counted. Similarly, PMD defines a threshold of 10 for the number of methods excluding methods, that start with "get" or "set". As our metric NOC has no such restriction, we adapted the value to 20 to account for these methods.

2.5 Metric sets with thresholds for software quality estimation

In most software quality models, e.g., the ISO 9126 standard (ISO/IEC, 2001-2004) or the FCM model (Factors, Criteria, Metrics) by (McCall et al, 1977) the assessment of quality attributes is done using metrics. For this assessment, sets of software metrics with thresholds can be used. The quality of the assessment then depends on the quality of the metric set. Therefore, it is important to have effective and efficient metric sets. As part of our work, we analyzed two concrete metric sets, one for methods and one for classes. The metric set for methods contains the metrics VG, NBD, NFC and NST with thresholds as defined in table 2. These four metrics cover the internal attributes of control-flow structuredness, coupling and size. Together, these attributes can be used to describe the complexity of a method: if it is not well structured, has to much coupling or is too large, it has a complexity problem.

The metric sets we used for the analysis of classes contains the metrics WMC, CBO, RFC, NORM, LOC, NOM and NST with thresholds as defined in table 2. These seven metrics cover the internal attributes of method complexity, coupling, inheritance, size and staticness. Thus, this set analyzes if the sum of the methods are too complex, the coupling of a class is dangerously high, whether inheritance is wrongly used, indicated by too many overwritten methods, the class is too large or if it has too many static methods. If this is the case, it is problematic due to high complexity generated by misuse of object-orientation or simply the overall complexity.

3 Foundations of Machine Learning

In this section, we introduce the concepts of machine learning we use in this work. After we briefly describe machine learning, we define the framework we use in general. Finally, an algorithm to learn axis-aligned d-dimensional rectangles will be introduced. Our approach for the optimization of metric sets is based on this algorithm.

Using computer technology, the analysis of large amounts of data is no longer a problem. One way to analyze data is machine learning. The assumption of learning theory is that the observed data can be described by an underlying process. The type of the process can vary and depends on the type of learning. For example, tt 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 into 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).

In this work, we use *concept learning*. A concept defines a rule 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 *baysian 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 $\mathscr{D}$ and classifies it using g and returns a sample $U = (X,g(X))$.

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 it is important to have models for noise and algorithms that use these models for learning in the presence of noise. 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 to noise models can be found in (e.g. 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 3 visualizes this concept.

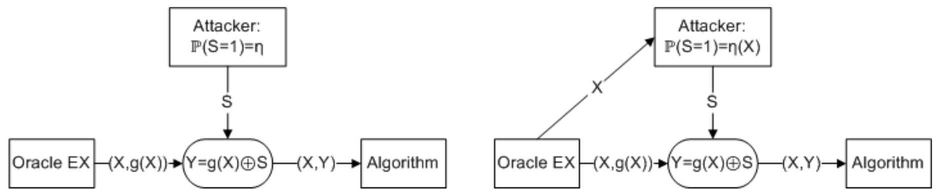

Fig. 3. This figure visualizes how an imaginary attacker manipulates samples provided by an oracle by randomly switching the labels. On the left hand side: Classification noise model, random noise independent of the input element. On the right hand side: Extended noise model, random noise depends on the input element.

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. The learner works through querying the oracle EX and evaluation the query function χ for the (possibly noisy) samples obtained from the oracle.

In this work we use a generalization of the classification noise model, the transparent noise model (Waack, 2009). 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. Now, $\mathbb{P}(S=1|X=x) = \eta(X)$ and thus the random noise depends on the input. For a given $x \in \mathfrak{X}^d$ 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

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

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

We furthermore assume that the query functions are *admissible*. We say that a query function χ is admissible, if it is not correlated to the noise rate $\eta(X)$ conditioned on the concept $g(X)$. 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.

Now that all the required concepts and definitions are introduced, we will state one of the central theorems of our learning approach. 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 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 transparent noise can be found in (Waack, 2009). For this paper, only the application of this theorem is of interest. 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 ML-estimators. Using 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

$$\mathbb{E}(\chi(X,0)|g(X)=0) = \frac{(1-\eta_1)e_{\chi,0} - \eta_1 e_{\bar{\chi},0}}{e_0 - \eta_1} \tag{3.3}$$

and

$$\mathbb{E}(\chi(X,1)|g(X)=1) = \frac{(1-\eta_0)e_{\chi,1} - \eta_0 e_{\bar{\chi},1}}{e_1 - \eta_0}. \tag{3.4}$$

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 the noise rates are estimated to be 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, the best of these hypothesis is selected.

3.1 A rectangle learning algorithm

In our work, we use an adaption of the rectangle learning algorithm for the SQM proposed by (Kearns, 1998) for the learning of axis-alligned d-dimensional rectangles. The algorithm uses the distribution $\mathscr{D}$ of the training data to partition the d-dimensional real-space, such that

$$\mathbb{P}(X_i \in I_{i,p}) = \mathbb{P}(X_i \in I_{i,q}) = \frac{1}{\lceil 1/\varepsilon \rceil} \approx \varepsilon \tag{3.5}$$

for each dimension $i = 1,\dots,d$ and $p,q = 1,\dots,\lceil 1/\varepsilon \rceil$ for $X \in \mathbb{R}^d$ randomly distributed according to $\mathscr{D}$, where X_i donates the i-th component of X. 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 our implementation of the algorithm, we utilize sorting algorithms to obtain these intervals according to the empirical distribution of a discrete training sample. We sort the values for each dimension and obtain sorted values $x'_{i,1},\dots,x'_{i,n}$. The intervals can then be defined as

$$I_{i,p} = [x'_{i,\frac{n}{\lceil 1/\varepsilon \rceil}(p-1)}, x'_{i,\frac{n}{\lceil 1/\varepsilon \rceil}p}]$$

for $p = 1,\dots,\lceil 1/\varepsilon \rceil$. These intervals fulfill the property defined by equation 3.5. 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 $p_{I_{i,p}} = \mathbb{P}(X_i \in I_{i,p} | g(X) = 1)$ is calculated, i.e., the probability that the target rectangle intersects an interval $I_{i,p}$. This probability is calculated using admissible queries and equation 3.4. 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. $p = 1,2,\dots$. 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 done from the right, i.e. $p = \lceil 1/\varepsilon \rceil, \lceil 1/\varepsilon \rceil - 1, \dots$ to determine the right, i.e. upper boundary u_i. Using this procedure for each dimension boundaries (l_i, u_i) are calculated.

As for the complexity of the algorithm, the first stage is $O(dn\log n)$, because for each dimension d the samples have to be sorted, which is $O(n\log n)$. In the second stage, for each dimension, the probability $p_{I_{i,o}}$ has to be 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 stage 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 want to introduce our machine learning based approach to optimize metric sets with thresholds for the detection of problematic entities. First we describe how we utilize the noise tolerant algorithm to learn d-dimensional axis-aligned rectangles to calculate thresholds. Then we show how this can be used in three ways. The

first is to reduce the size of a metric set to obtain an effective and efficient subset. The second way is to reduce the complexity of the method that is used to classify entities as problematic. The threshold approach we use defines an easy-to-understand classification method, where the reason why an software entity is classified as problematic is obvious, if only the metric values are considered. Other classification methods may not be as easy to understand, they could even be defined by some black box like algorithm. With our approach it is possible to determine thresholds for a metric set, such that the classification of an arbitrary complex classification method is reduced to a simple threshold approach with an error as low as possible. The third way our approach can be used is to introduce a threshold based and environment specific classification method where no classification has been available yet. Even if no classification method is not available, it is possible to manually classify a sample of software entities based on experience. This sample can then be used to determine thresholds for a metric set.

4.1 Calculation of thresholds using rectangle learning

Our analysis approach is based on a given metric set $M = \{m_1, \dots, m_d\}$ and a set of software entities $\mathbf{X}$ with given classifications $\mathbf{Y}$. The set $\mathbf{X}$ could for example consist of components, classes or methods. The aim is to obtain thresholds 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 $(\mathbf{X}, \mathbf{Y})$. Using the metric set M it is possible to transform the set of software entities into a set of vectors in the n-dimensional real space, such that $M(\mathbf{X}) := \{(m_1(x), \dots, m_d(x)) : x \in \mathbf{X}\} \subset \mathbb{R}^n$. The pair $(M(\mathbf{X}), \mathbf{Y})$ can then be used as input for the axis-aligned rectangle learning algorithm, which we introduced in section 3.1. As result the algorithm will yield pairs of upper and lower bounds (l_i, u_i) for each dimension $i = 1, \dots, d$. 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, the upper bound of the rectangle in the i-th dimension can be interpreted as a threshold for the metric m_i. Therefore, with $t_i = u_i$ a set of thresholds $T = \{t_1, \dots, t_d\}$ 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

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

Under the assumption that metric values are positive, this classification describes a rectangle with upper bounds t_i and the 0 as lower bound. Figure 4 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.

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

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

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

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

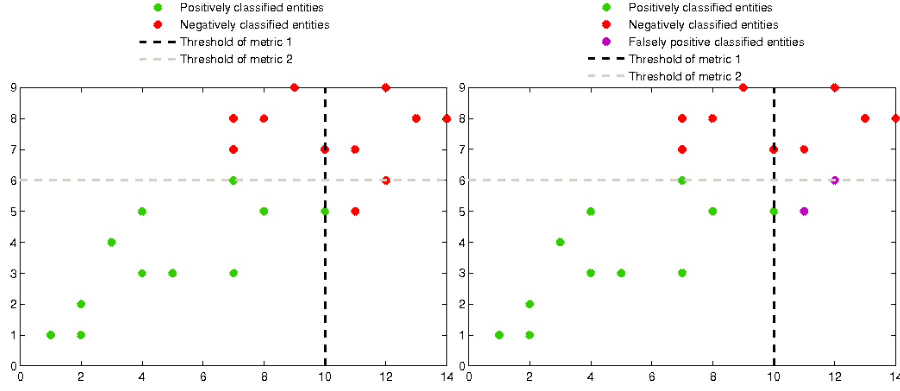

Fig. 4. Right hand side: classification using two metrics with thresholds at 10, respectively 6. On the left hand side: classification using only the second metric.

with

$$\mathbb{1}_{\{cond\}} = \begin{cases} 1 & \text{if } cond \text{ is } true \\ 0 & \text{if } cond \text{ is } false. \end{cases} \tag{4.4}$$

This approach 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.

4.2 Optimization of the efficiency of metric sets with thresholds

The first way to utilize the approach to determine thresholds for a given metric set and a classification as it was introduced in the previous section, is to optimize an already existing effective metric set M with thresholds T. In this case, optimization means that an effective and efficient subset M' of M with thresholds T' is determined. We say that a metric set is effective, if it yields the correct classification and efficient if it is the smallest set to do so. For this, the first step is to create a training sample $(\mathbf{X}, \mathbf{Y})$ that can be used by the method to calculate thresholds we introduced in the previous section. For a given metric set M with thresholds T, the learning sample $(\mathbf{X}, \mathbf{Y})$ can be defined using f_0 as $(\mathbf{X}, \mathbf{Y}) = \{(x, f_0(x, M, T)) : x \in \mathbf{X}\}$. Thus, a training sample can be created using a given set of software entities $\mathbf{X}$.

The aim is to determine a subset of M that is as effective as M, but of smaller size and therefore more efficient. The effectiveness of a metric set with threshold with respect to a given classification can be interpreted as its classification error. Therefore, we need to determine a set $M' = \{m'_1, \ldots, m'_{d'}\}$ with thresholds $T' = \{t'_1, \ldots, t'_{d'}\}$ that yields a low classification error. To achieve this, we apply our learning approach with the training set $(\mathbf{X}, \mathbf{Y})$ to all subsets of M, or in other words all sets that are element of the power set $M' \in \mathscr{P}(M) \setminus \emptyset$. For each of the subsets, the approach will yield thresholds T'. The empirical classification error for M', T' can then be calculated as

$$\varepsilon_{\mathbf{X},\mathbf{Y}} = \frac{1}{|\mathbf{X}|} \sum_{x \in \mathbf{X}} \mathbb{1}_{f_0(x,M,T) \neq f_0(x,M',T')} \tag{4.5}$$

Subsets M' of M are effective if their classification error is close to 0. Therefore, the smallest subset M' that is effective is an effective and efficient subset of M and can be used to replace M and gain an equally effective, but more efficient metric set. In figure 4 an example of how the classification of two metrics can be expressed using only one metric and how entities may be misclassified.

4.3 Reduction of the classification complexity

Another way to utilize our approach is to reduce the complexity of the classification that is used. With thresholds, a rather 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 gives an indicator what the problem might be. A slightly more complex approach is to say that a given number of infractions λ is allowed, thus λ thresholds my be violated. The classification defined by a metric set M with thresholds T can then be described by a function

$$f_\lambda(x, M, T) = \begin{cases} 1 & \text{if } |\{\, i \in \{1, \ldots, n\} : \, m_i(x) > t_i \,\}| \leq \lambda \\ 0 & \text{if } |\{\, i \in \{1, \ldots, n\} : \, m_i(x) > t_i \,\}| > \lambda. \end{cases} \tag{4.6}$$

which is a generalization of the definition of f_0.

A reason for such a rule might be, to grant the developers more freedom, e.g., they may write short methods with a high structural complexity or long methods with a low structural complexity. But methods that are both long and have a high structural complexity are not allowed. The classification using λ allowed infractions introduces a certain complexity to understand *why* a problematic entity was classified as such and which counter measures should be taken. With more complex approaches, that are maybe not only based on thresholds, there is a further increase in the complexity. In general, the classification could be performed by an arbitrary complex function f. A metric set that yields the same classification but where no infraction whatsoever are allowed is preferable, because as Occams Razor suggests, the simplest solution is preferable. There are many examples for these more complex models, that are used for defect prediction, e.g. models using neural networks or regression models (see section 6).

Our method to calculate thresholds for a metric set can be used to determine such a metric set. Let $f : \mathscr{X} \to \{0, 1\}$ the function used to classify software entities from an entity space $\mathscr{X}$, e.g., $\mathscr{X}$ could be the space of Java classes. Then a training set $(\mathbf{X}, \mathbf{Y})$ can be determined for a given set of software entities similarly to how it is done in 4.2 as $(\mathbf{X}, \mathbf{Y}) = \{(x, f(x)) : x \in \mathbf{X}\}$. This training set can be used in exactly the same way as it is done in 4.2 to determine a good set of metrics $M' \subset M$ with thresholds.

4.4 Learning of environment specific thresholds

An important aspect of thresholds for metrics is, that they are often dependent on the project environment, such as the requirements, the developer qualification or the programming language. Therefore, the best results can be achieved with thresholds tailored to the own environment. In the previous two sections, we only showed how our approach can be used to optimize already existing classification methods. However, it can

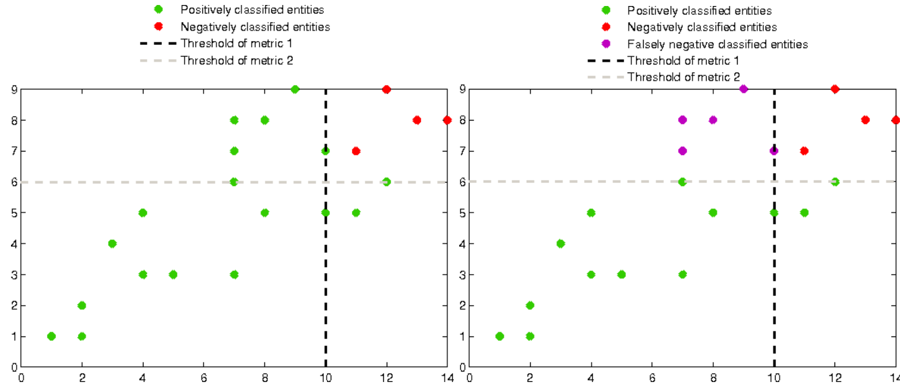

Fig. 5. Right hand side: classification using two metrics with thresholds at 10, respectively 6, where one infraction is allowed. On the left hand side: classification using the second metric with no infractions allowed.

also be used to determine thresholds were currently no method of classification exists. For this, an expert has to manually create a training set $(\mathbf{X}, \mathbf{Y})$. The expert has to choose software entities that are typical for the project environment and manually discriminate them the good and the bad ones. This way a training set is defined, that can then be used the same way as in the previous two sections to determine a good set of metrics $M' \subseteq M$ with thresholds T'. This combination of metrics and thresholds is optimal for the given entities that are part of the training sample. If the entities in the training set are typical for the project environment, they will also be optimal for the whole project environment.

5 Case Studies

For the validation of our approach for the optimization of metric sets, we performed two case studies consisting of 6, respectively 3 experiments. After the general methodology we used to perform the case studies is described, the results of the case studies are presented.

5.1 Methodology

Our case studies are based on metric data measured for large scale open source software projects. That way, we obtained sets of software entities $\mathbf{X}$ with metric values $M(\mathbf{X})$ that can be used for our optimization approach. How the classification $\mathbf{Y}$ is obtained depends on the aim of the threshold calculation (see sections 4.2, 4.3 and 4.4). To guarantee the validity of our results we randomly split the measured data into three disjunctive sets: a *training set* $(\mathbf{X}_{train}, \mathbf{Y}_{train})$ that contains 50% of the data; a *selection set* $(\mathbf{X}_{sel}, \mathbf{Y}_{sel})$ that contains 25% of the data; an *evaluation set* $(\mathbf{X}_{eval}, \mathbf{Y}_{eval})$ that contains 25% of the data. Each of these 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 then used to select the best of these hypotheses, i.e., an optimal hypothesis h^* with respect to the empirical classification error $\varepsilon_{\mathbf{X}_{sel},\mathbf{Y}_{sel}}$. The empirical classification error $\varepsilon_{\mathbf{X}_{eval},\mathbf{Y}_{eval}}$ the error of h^* is calculated to evaluate the performance of h^*.

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 that credit card number consist of 16 digits. This assumption would also be 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 occured. 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, it 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 would be equally good. By splitting the available data this effect is prevented. Once yet unseen credit card numbers are checked for validity, the first hypothesis will still yield the correct results and the error will remain zero. However, for the second hypothesis, the error would increase 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. For algorithms that compute a whole set of hypothesis like the rectangle learning algorithm we use, a split into three sets that contain 50%, 25% and 25% of the data for the training, selection and evaluation is a commonly used method to prevent overfitting.

5.2 Optimization of a metric set for methods

In the first case study we analyzed a metric set for C++ and C# methods and C functions and performed a total of 6 experiments. We did not consider C++ functions. As metric set, we used the set $M = \{VG, NBD, NFC, NST\}$ as it was introduced in section 2.5. As thresholds T, we used the values shown in table 2. The rational of using this set is, that it is effectively able to determine the overall complexity of a method by analyzing its structure, its coupling and its size.

For this case study, we measured software from various domains implemented in the three language C, C++ and C#. The measurement was performed using the tool SourceMonitor (Campwood Software, 30.4.2009). For C we measured the Apache HTTP Server (The Apache Software Foundation, 30.4.2009a), an open source HTTP server for *nix and windows systems, which is developed and maintained by the Apache Foundation (The Apache Software Foundation, 30.4.2009b). C++ methods were measured for two of the main components of the *K Desktop Environment* (KDE) (The KDE Developers, 30.4.2009) for Linux, the kdebase and the kdelibs component. 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 provides a library of important core functions that is used by KDE, e.g., for networking, printing and multithreading. For C# we measured three projects. The first C# project we measured is

AspectDNG (CollabNet, 30.4.2009), an *aspect weaver* that allows *Aspect Oriented Programming* (Kiczales et al, 2002) in C#. The second is the NetTopologySuite (NetTopologySuite Contributors, 30.4.2009), a *Geographic information system* (GIS) solution for the .NET platform. Additionally, we measured SharpDevelop (SharpDevelop Contributors, 30.4.2009), an *Integrated Development Environment* (IDE) for C#, VB.NET and Boo. Detailed informations about the versions and the sizes of the measured software projects are shown in table 3, statistical information about the measured metric data is shown in table 4.

Our first three experiments were to apply our approach to reduce the size of a metric set to improve its efficiency (see section 4.2) to the metric set M in the languages C, C++ and C#. We used the function f_0 to classify the data according to M and T. The metric with the best performance in this setting is NFC. By using only the metric NFC instead of the whole metric set M, it is possible to achieve nearly the same classification, with an empirical error of 0.78%, 0.06% and 0.59% for C, C++ and C# respectively. The computed threshold equals the original threshold and stays 5 for all three languages. Thus, we were able to reduce the size of the metric set by 75% with nearly no loss. In fact, the loss of below 1% can be interpreted as noise.

In the other three experiments that we performed on the method data, we validated our approach to reduce the complexity of the classification method, as introduced in section 4.3. For this, we classified our data in such a way, that $\lambda = 1$ threshold infraction is allowed. Therefore, the classification can be described by f_λ. While in the previous experiment the metric NFC was the best for all three languages, in this experiment the metric NST has the best performance in case of C with an error of 0.84%. For C++ and C# the metric VG yields the best results with an error of 0.87% and 1.36% respectively. This shows that it is not only possible to describe the one-infraction-allowed model in a zero infraction model, but also that the set size could be reduced by 75% for all three languages.

More information on six experiments is shown in table 5. The complete table of results can be found under (Herbold et al, 2009).

Table 3. Below, informations about the the projects that were analyzed as part of our case studies presented in the sections 5.2 and 5.3 are given.

Project name	**Version**	**Language**	**Number of Methods**
Apache Webserver	2.2.10	C	6718
kdebase	12/05/2008	C++	21404
kdelibs	12/05/2008	C++	37444
AspectDNG	1.0.3	C#	2759
NetTopologieSuite	1.7.1.RC1	C#	3059
SharpDevelop	2.2.1.2648	C#	15700
Project name	**Version**	**Language**	**Number of Classes**
Eclipse Java Development Tools	3.2	Java	5195
Eclipse Platform Project	3.2	Java	7618

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

Metric	Language	Median	Arithmatic 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

Table 5. The table below shows the results of the first case study. It lists the number of methods used in the learning process, the best metric on its own and the best subset of metrics together with their respective empirical errors.

λ	Language	Number of Methods	Best metric	ε_{single}	Best subset	$\varepsilon_{combination}$
0	C	6718	NFC	0.78%	VG, NFC, NST	0.48%
0	C++	21404	NFC	0.06%	VG, NFC, NST	0.02%
0	C#	21503	NFC	0.59%	VG, NFC	0.22%
1	C	6718	NST	0.84%	VG, NST	0.66%
1	C++	21404	VG	0.87%	VG, NST	0.56%
1	C#	21503	VG	1.36%	VG. NBD, NFC	1.14%

5.3 Optimization of a metric set for classes

The second case study was performed on the class level of Java, where we performed three experiements. The metric set we analyzed and optimized is $M = \{WMC, CBO, RFC, NORM, NOM, LOC, NSM\}$ as defined in section 2.5. The thresholds T were defined as in table 2. This metric set determines whether the complexity of a class is problematic by analyzing its method complexity, coupling, misuse of inheritance, size and and staticness.

For the measurement of the metrics two tools were used. We used *ckjm* (Diomidis D. Spinellis, 30.4.2009), a tool for the measurement of the Chidamber and Kemerer metric suite for Java to measure the metrics CBO, RFC and NOM. The metric NOM could be measured using this tool, as it assigns a complexity of 1 to each method, therefore its value for WMC_{ckjm} is nothing else but our metric NOM. For the measurement of the other four metric WMC, NORM, LOC and NSM we used the *Eclipse Metrics Plug-in* (Frank Sauer, 30.4.2009). We measured two large-scale and well defined open-source projects, both run by the *Eclipse Foundation* (Eclipse Foundation, 30.4.2009b): the Eclipse Platform (Eclipse Foundation, 30.4.2009a) and the Eclipse *Java Development Tools* (JDT) (Eclipse Foundation, 30.4.2009c). The Eclipse Platform project is responsible for many of the main components for the Eclipse Platform, like the handling of resources, the workbench or the editor framework. However, we did not measure the whole project, but excluded the *Standard Widget Toolkit* (SWT), a framework for user-interface programming. Formally, SWT is part of the Eclipse Platform Project, but it is mainly independent. The Eclipse JDT defines an IDE for the development of Java software on top of the Eclipse Platform. Additional information about the measured versions of both projects is shown in table 3.

The first experiment was to apply our approach for the optimization of the efficiency of a metric set (see section 4.2 to the set M. We used f_0 to classify the measured metric data according to M and T. The metric with the best performance was CBO with an empirical error of 1.22%. This would be a set size reduction of 83%. With the metric set $M' = \{WMC, CBO, NOM, NSM\}$ an empirical error of 0.31% is achieved by M' and the set size is reduced by 33%. Thus, depending on the empirical error that shall be achieved the size of the metric set can be reduced by up to 83%.

In a two further experiments, we validated that the approach to reduce the complexity of the classification as defined in section 4.3 to this metric set as well. Similarly to how we validated the approach in the first case study, we allowed threshold infractions. In the first of these two experiments, we allowed $\lambda = 1$ infraction, in the second $\lambda = 2$ infractions. For the classification of the measured data, we used the function f_λ. In case of $\lambda = 1$ the metric NOM yields the best results on its own with an empirical error of 4.91%. The best results are obtained using the metrics RFC, NORM, NOM and NSM, with an error of 2.64%. Again, this shows that it is possible to describe the one-infraction-allowed model as a zero infraction model, while reducing the size of the metric set by 33%. In case of two allowed infractions, all subsets of our metric set M can be used with a very low error, which is between 1.92% and 6.42%. That the error for all possible metric sets is this low, is due to the fact that only 7% of the data is classified negatively. Hence, even the trivial hypothesis that classifies all data as positive would yield an error of only 7%. However, using only the metric RFC this can be

further improved to an error of 1.99% while reducing the set size by 83%. The metric set RFC and LOC achieves an error of 1.92% while reducing the set size by 66%.

Additional information for all the results is shown in table 7. The complete table of results can be found under (Herbold et al, 2009).

Table 6. This table lists some statistical information about the measured Java classes.

Metric	Median	Arithmatic Mean	Max value	Threshold
WMC	12	23.25	814	100
DIT	0	0.49	5	-
NOC	0	0.31	131	-
CBO	8	12.08	197	5
RFC	20	33.54	620	100
NORM	0	0.68	35	3
LOC	26	70.98	2755	500
NOM	6	10.54	205	20
NSM	0	0.65	58	4

Table 7. The table below shows the results of the second case study. It lists the number of classes used in the learning process, the best metric on its own and the best subset of metrics together with their respective empirical errors.

λ	Number of Classes	Best metric	ε_{single}	Best subset	$\varepsilon_{combination}$
0	5399	CBO	1.22%	WMC, CBO, NOM, NSM	0.31%
1	5399	NOM	4.91%	RFC, NORM, NOM, NSM	2.64%
2	5399	RFC	1.99%	RFC, LOC	1.92%

5.4 Discussion of the case study results

As we showed in our case studies, our approach can be used to determine effective and efficient metric sets as well as reduce the classification complexity. We were able to significantly reduce the size of two metric sets, while keeping their generality. Furthermore, we demonstrated that it is possible to use a model with zero allowed infractions instead of a model where some infractions are allowed. In our experiments, the exponential nature of the approach was no threat to its applicability. The execution of all 9 experiments performed as part of both case studies combined took 2 minutes and 19 seconds on a normal desktop workstation running on an Intel Core2 Duo E8400 processor.

In the experiments for the optimization of metric sets the metrics NFC and CBO were the ones with the best performance. This coincides with the user intuition that in most cases coupling is responsible for most of the complexity and therefore mostly the reason while entities are considered to be problematic. The reason for this is that non-local communication is often complex. If you consider methods, in modern languages

not nessecarily the length or its structural complexity is what defines its complexity and understandability. Of course, short methods with a low structural complexity are usually simple. But even on-line methods can be difficult to understand, if in this one line another method is called. This is due to the fact, that at least the parameters and the return values of a method have to be understood. Furthermore, method calls also have an effect on the internal state of on object, which has to be considered. Therefore, the understandability of a method call can be anywhere between "very easy" and "very difficult".

For example, consider a method `double calculateAverage(double[] array)`. What exactly this method calculates is unclear, only that it is some kind of average value. However, the term average is ambiguous. It could mean the arithmetic mean value, the geometric mean value, the median or even some other definition of an average value. So what the method actually calculates, needs to be inferred from another source. If the documentation is available and sufficient, this is no problem. But it is also possible, that the the documentation is unavailable or insufficient, e.g., "Calculates the average value of the array". In this case, there are only two means left to infer the meaning of "average" in this context: if available, analyze the source code of the method directly, otherwise it has to be determined using reverse engineering. As this example shows, the seemingly simple method call could in fact be very difficult to understand. Of course, this example exaggerates the possible problems. But even so, when using external libraries the reality is often very similar.

As for the coupling of classes measured using CBO, the metric NFC is obviously related to CBO. On of the criteria defining the coupling of a class is the number of classes, from which methods are called. Statistically speaking, this increases with the number of method calls with a high probability. Thus, high values of NFC lead to high values of CBO in probability. Vice versa, if methods from many different classes are called, it is highly probable that many methods are called. Furthermore, (Binkley and Schach, 1998) validated that coupling is a good predictor for the complexity of software. In conclusion, it is therefore not surprising that these two metrics were the ones with the best performance, as it validates the user intuition, they are related and previous studied showed, that coupling metrics are indeed good for this purpose.

Another interesting result is the behavior of the metric NSM in our results. While it is useless on its own, due to the fact that most classes do not have any static methods, it is part of the best subset of metrics and also most other good results. This shows, that this aspect of a class cannot be sufficiently expressed using the other six metrics.

The result of the experiment with two allowed infractions in the second case study shows, that two infractions are in fact two much in this setting. Only 7% of the entities are classified as problematic. This distribution of positive and negatively classified entities is a problem for effective learning, as most learning algorithm require a better balance of positive and negative entities. This is due to the fact, that otherwise the trivial hypothesis already yields a good enough result and it becomes difficult to detect noise.

6 Related Work

The metric suite proposed by (Chidamber and Kemerer, 1994) has been the subject of many studies. A literature overview about the analysis of the Chidamber and Kemerer metric suite was performed by (Subramanyam and Krishnan, 2003). Since their overview, the metric suite has been used in further studies. The capability to predict faults was analyzed by (Gyimothy et al, 2005), using various statistical and machine learning methods. A neural network based approach for fault prediction using object-oriented measures, including the Chidamber and Kemerer metrics, was introduced by (Kanmani et al, 2007). In (Olague et al, 2007) the authors analyzed and compared the Chidamber and Kemerer metrics and two other metric suites, the MOOD metrics (Abreu and Carapuça, 1994) and the QMOOD metrics (Bansiya and Davis, 2002) with respect to their capabilities to predict faults.

Independent of any specific metric sets, like MOOD and the Chidamber and Kemerer metrics, many researchers are concerned with defining predictor models based on historical data, e.g. for defect prediction. Most of these models are – at least to some degree – based on software metrics in general (e.g. Denaro et al, 2002; Ostrand et al, 2004; Nagappan et al, 2006; Zimmermann et al, 2007).

Research on how to obtain an environment specific metric set can be obtained was performed by (Basili and Selby, 1985). In contrast to our 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 (Mulaik, 1972). In an earlier work, we used a simpler machine learning based approach for the optimization of a metric set for TTCN-3 (Werner et al, 2007). This work is different, as we use a more complex algorithmic approach and apply it in different settings, i.e., other programming languages and also other task, e.g., the simplification of the classification method.

There is also work, that directly uses or defines thresholds. In (Lorenz and Kidd, 1994) the authors define thresholds for many metrics. Further work on thresholds was performed by (French, 1999). French introduced a statistical method to obtain thresholds and used this method to define thresholds for object-oriented and procedural software. In another study (Benlarbi et al, 2000) give an overview over the thresholds that have been defined for the Chidamber and Kemerer metrics and perform an analysis on threshold effects. Some statical analysis tools, like PMD use software metric thresholds as part of their software analysis Copeland (2005).

7 Conclusion

We present a novel high-level approach for the calculation of thresholds for a metric set. These sets can than be used to classify software entities as problematic with respect to quality attributes, e.g., fault-proneness. Our approach is able to yield a predefined classification with a high accuracy of over 95%, while reducing the size of the metric sets used, thus making it simpler and more effective. In two case studies, we proved that our approach can be used to optimize metric sets or simplify classification models. Another possible application of our approach is to calculate thresholds where no formal classification model exists yet, but experts determine the classification manually.

Studies on how well our approach works for the simplification of currently existing classification models are an interesting topic for further research. It would be interesting to see how good a simple threshold approach performs in comparison to complex classification approaches, like neural networks or support vector machines. Especially the comparison to algorithms where the classification method is a black-box and it is unclear *why* an entity was classified the way it was is interesting. With thresholds, additional information would be gained as to what the problem is: the attributes that are measured by metrics that violate their thresholds are the problem.

Altogether, our results are very good and as our case studies included various programming languages and projects from different domains, we conclude that our approach can be successfully used in other settings as well.

A Table of Acronyms

NFC *Number of Function Calls*

NBD *Nested Block Depth*

VG *Cyclomatic Number*

WMC *Weighted Methods per Class*

NST *Number of Statements*

DIT *Depth of Inheritance Tree*

CBO *Coupling Between Objects*

RFC *Response For a Class*

LCOM *Lack of Cohession in Methods*

NOC *Number of Children*

NOM *Number of Methods*

NORM *Number of Overridden Methods*

LOC *Lines of Code*

NSM *Number of Static Methods*

KDE *K Desktop Environment*

GIS *Geographic information system*

IDE *Integrated Development Environment*

SQM *Statistical Query Model*

FCM *Factor, Criteria, Metrics*

SWT *Standard Widget Toolkit*

ISO *International Organization for Standardization*

GQM *Goal/Question/Metric*

TTCN-3 *Testing and Test Control Notation*

B 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
$\mathbb{P}$	probability
$\eta(X)$	random noise rate
η_0, η_1	conditional expected noise rate
$\mathbb{E}$	expected value
χ	query function
M	a set of metrics $\{m_1, \ldots, m_n\}$
$(\mathbf{X}, \mathbf{Y})$	a discrete learning sample
$(\mathbf{X}_{train}, \mathbf{Y}_{train})$	a training set used to calculate hypotheses
$(\mathbf{X}_{sel}, \mathbf{Y}_{sel})$	a selection set used to select the best hypothesis
$(\mathbf{X}_{eval}, \mathbf{Y}_{eval})$	an evaluation set, used to evalute the quality of the selected hypothesis
$M(X)$	the transformation of software entities into the n-dimensional 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

Bibliography

Abreu F, Carapuça R (1994) Object-oriented software engineering: Measuring and controlling the development process. In: proceedings of the 4th International Conference on Software Quality

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

Bansiya J, Davis CG (2002) A hierarchical model for object-oriented design quality assessment. IEEE Trans Softw Eng 28(1):4–17, DOI http://dx.doi.org/10.1109/32.979986

Basili V, Rombach H (1988) The TAME project: towards improvement-oriented softwareenvironments. 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 http://dx.doi.org/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

Binkley AB, Schach SR (1998) Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In: ICSE '98: Proceedings of the 20th international conference on Software engineering, IEEE Computer Society, Washington, DC, USA, pp 452–455

Campwood Software (30.4.2009) SourceMonitor. URL `http://www.campwoodsw.com/sourcemonitor.html`

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

CollabNet (30.4.2009) AspectDNG. URL `http://aspectdng.tigris.org/`

Copeland T (2005) PMD applied

Denaro G, Morasca S, Pezzè M (2002) Deriving models of software fault-proneness. In: SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering, ACM, New York, NY, USA, pp 361–368, DOI http://doi.acm.org/10.1145/568760.568824

Devroye L, Györfi L, Lugosi G (1997) A Probabilistic Theory of Pattern Recognition. Springer, New York

Diomidis D Spinellis (30.4.2009) ckjm Chidamber and Kemerer Java Metrics. URL `http://www.spinellis.gr/sw/ckjm/`

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

Eclipse Foundation (30.4.2009a) Eclipse Platform Project. URL http://www.eclipse.org/Platform

Eclipse Foundation (30.4.2009b) Eclipse Project. URL http://www.eclipse.org

Eclipse Foundation (30.4.2009c) Java Developement Tools Eclipse Plug-in. URL http://www.eclipse.org/JDT

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

Frank Sauer (30.4.2009) Eclipse Metrics Plug-in. URL http://metrics.sourceforge.net/

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 http://dx.doi.org/10.1016/S1389-1286(03)00249-4

Gyimothy T, Ferenc R, Siket I (2005) Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans Softw Eng 31(10):897–910, DOI http://dx.doi.org/10.1109/TSE.2005.112

Herbold S, Grabowski J, Waack S (2009) Result tables

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

Kanmani S, Uthariaraj VR, Sankaranarayanan V, Thambidurai P (2007) Object-oriented software fault prediction using neural networks. Inf Softw Technol 49(5):483–492, DOI http://dx.doi.org/10.1016/j.infsof.2006.07.005

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

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

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

Mammen E, Tsybakov AB (1999) Smooth discrimination analysis. The Annals of Statistics 27(6):1808–1829

McCabe TJ (1976) A complexity measure. IEEE Trans Softw Eng 2(4):308–320, DOI http://dx.doi.org/10.1109/TSE.1976.233837

McCall J, Richards P, Walters G, States U, Division ES, Force A, Center RAD, Command S (1977) Factors in software quality. NTIS

Mulaik S (1972) The foundations of factor analysis. McGraw-Hill New York

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 http://doi.acm.org/10.1145/1134285.1134349

NetTopologySuite Contributors (30.4.2009) NetTopologySuite. URL `http://code.google.com/p/nettopologysuite`

Olague HM, Gholston S, Quattlebaum S (2007) Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. IEEE Trans Softw Eng 33(6):402–419, DOI http://dx.doi.org/10.1109/TSE.2007.1015, senior Member-Etzkorn, Letha H.

Ostrand TJ, Weyuker EJ, Bell RM (2004) Where the bugs are. SIGSOFT Softw Eng Notes 29(4):86–96, DOI http://doi.acm.org/10.1145/1013886.1007524

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

SharpDevelop Contributors (30.4.2009) AspectDNG. URL `http://www.icsharpcode.net/OpenSource/SD/`

Shawe-Taylor J, Cristianini N (2004) Kernel Methods for Pattern Analysis. Cambridge University Press

Subramanyam R, Krishnan MS (2003) Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. IEEE Trans Softw Eng 29(4):297–310, DOI http://dx.doi.org/10.1109/TSE.2003.1191795

The Apache Software Foundation (30.4.2009a) Apache Webserver. URL `http://httpd.apache.org/`

The Apache Software Foundation (30.4.2009b) The Apache Software Foundation. URL `http://www.apache.org`

The KDE Developers (30.4.2009) K Desktop Environment. URL `http://www.kde.org`

Tsybakov AB (2004) Optimal aggregation of classifiers in statistical learning. The Annals of Statistics 32(1):135–166

Waack S (2009) Learning with conditional noise

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

Yourdon E, Constantine L (1979) Structured design: fundamentals of a discipline of computer program and systems design. Prentice-Hall, Inc. Upper Saddle River, NJ, USA

Zimmermann T, Premraj R, Zeller A (2007) Predicting defects for eclipse. In: PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering, IEEE Computer Society, Washington, DC, USA, p 9, DOI http://dx.doi.org/10.1109/PROMISE.2007.10