TTCN-3 Quality Engineering: Using Learning Techniques to Evaluate Metric Sets

Edith Werner, Jens Grabowski, Helmut Neukirchen,
Nils Röttger, Stephan Waack, and Benjamin Zeiss

Institute for Informatics, University of Göttingen
Lotzestr. 16–18, 37083 Göttingen, Germany
{ewerner,grabowski,neukirchen,nroettger,waack,zeiss}@cs.uni-goettingen.de

Abstract. Software metrics are an essential means to assess software quality. For the assessment of software quality, typically sets of complementing metrics are used since individual metrics cover only isolated quality aspects rather than a quality characteristic as a whole. The choice of the metrics within such metric sets, however, is non-trivial. Metrics may intuitively appear to be complementing, but they often are in fact non-orthogonal, i.e. the information they provide may overlap to some extent. In the past, such redundant metrics have been identified, for example, by statistical correlation methods. This paper presents, based on machine learning, a novel approach to minimise sets of metrics by identifying and removing metrics which have little effect on the overall quality assessment. To demonstrate the application of this approach, results from an experiment are provided. In this experiment, a set of metrics that is used to assess the analysability of test suites that are specified using the *Testing and Test Control Notation* (TTCN-3) is investigated.

1 Introduction

Quantitative methods like software metrics are a powerful means to assess and control software development [1]. In software development process maturity models, like *Capability Maturity Model Integration* (CMMI) [2] or *Software Process Improvement and Capability dEtermination'* (SPICE) [3], the usage of metrics is considered as an indicator of a high process maturity. For the quality assessment of test suites that are specified using the *Testing and Test Control Notation* (TTCN-3) [4, 5], we have thus proposed a set of several TTCN-3 metrics [6]. When we presented and discussed these metrics at the Fifth International Workshop on System Analysis and Modelling (SAM'06), it was pointed out that the assessment of TTCN-3 specifications depends on the quality characteristic to be evaluated and that several metrics are needed to measure all aspects of a characteristic. Therefore, we developed subsequently a comprehensive test specification quality model [7] that takes various different quality characteristics into account to asses the quality of a test specification. Following the ISO/IEC 9126 standard [8], each quality characteristic is divided into further sub-characteristics

and each sub-characteristic is quantified using several metrics. By taking the measurements of the different metrics into account, a classification of the overall quality of a test specification can be made. But, as also pointed out at the discussion at SAM'06, this may lead to a cluttered set of metrics that is hard to interpret.

Hence, we developed a machine learning approach that can be used to optimise a set of metrics and that helps to judge whether a new metric should become part of the metrics set (i.e. measures a new quality aspect) or whether it is already subsumed by other metrics (i.e. the new metric leads to the same conclusion as other metrics already do). In this paper, we present our machine learning approach and show its practicability by applying it to a set of TTCN-3 metrics.

This paper is structured as follows: after this introduction, we provide foundations on software metrics and machine learning in Sect. 2. As our main contribution, we present our approach of using learning techniques to evaluate metric sets in Sect. 3. Then, in Sect. 4, we demonstrate the usage of this approach by applying it to a suite of TTCN-3 metrics. Finally, we conclude with a summary and outlook.

2 Foundations

In this section, foundations on software metrics and on pattern analysis using *Probably Approximately Correct* (PAC) learning are presented. In our case, the patterns to be learned will be the varying values of a metric set that contribute to a corresponding overall classification of the quality of a test specification.

2.1 Software Metrics

According to Fenton et al. [1], the term *software metrics* embraces all activities which involve software measurement. Software metrics are mostly used for management purposes and quality assurance in software development. They can be classified into measures for attributes of *processes*, *resources*, and *products*.

For each class, internal and external attributes can be distinguished. *External attributes* refer to how a process, resource, or product relates to its environment; *internal attributes* are properties of a process, resource, or product on its own, separate from any interactions with its environment. Internal product attributes are typically obtained by static analysis of the code to be assessed. External product attributes on the other hand are normally gained by accumulating quantitative data of interest during program execution.

Internal product metrics can be structured into *size* and *structural* metrics [1]. Size metrics measure properties of the number of usage of programming or specification language constructs, e.g. the metrics proposed by Halstead [9]. Structural metrics analyse the structure of a program or specification. The most popular examples are complexity metrics based on control flow or call graphs and coupling metrics.

Concerning metrics for measuring complexity of control structures, the most prominent complexity metric is the *cyclomatic complexity* from McCabe [10, 11]. It is a *descriptive* metric, i.e. its value can be objectively derived from source code. By additionally using threshold values, this metric becomes also *prescriptive* [12], i.e. it helps to control software quality. For example, when threshold violations of the metric values are analysed, it can help to identify complex modules which shall be split into several simpler ones [11].

Metrics are often used in the context of a quality model. The ISO/IEC standard 9126 [8] defines such a quality model for internal quality, external quality, and quality-in-use of software products. It is possible to apply such quality models to test specifications as well [7]. The ISO/IEC quality model describes each distinct quality characteristic of a software product by further subcharacteristics that refine each characteristic. To quantify the quality with respect to each subcharacteristic, according metrics can be used. Based on these metrics and related thresholds, an overall classification of a given software product can be made. The actual scheme used for the overall classification may vary from project to project, e.g. one project may require a scenario in which all the calculated metric values need to be within the corresponding thresholds, whereas in other projects it may be sufficient if only a certain percentage of the involved metrics do not violate their thresholds.

To make sure that reasonable metrics are chosen, Basili et al. suggest the *Goal Question Metric* (GQM) approach [13]: First, the goals which shall be achieved (e.g. improve maintainability) must be defined. Then, for each goal, a set of meaningful questions that characterise a goal is derived. The answers to these questions determine whether a goal has been met or not. Finally, one or more metrics are defined to gather quantitative data which give answers to each question. The GQM approach, however, does not make any statement on the similarity of metrics and whether certain metrics are statistically replaceable by others.

There are numerous publications that try to tackle the orthogonality problem of software metrics, i.e. they try to identify those measures in a set of metrics that do not deliver any meaningful additional information. One early work of Henry et al. [14] demonstrated the high-degree relationship between the *cyclomatic complexity* and Halstead's complexity measures by means of Pearson correlation coefficients. A good overview on further related work is provided by Fenton et al. [1]: they list approaches to investigate the correlation of metrics using Spearmans' rank correlation coefficient and Kendall's robust correlation coefficient. To express the nature of the associations, *regression analysis* has been suggested. Furthermore, *principal component analysis* has been used to reduce the number of necessary metrics by removing those principal components that account for little of the variability. We are not aware of any approaches that use a learning approach as described in the remainder of this paper.

2.2 Extracting pattern from data

The so-called Keplers's third law states that the squares of the periods of planets are proportional to the cubes of their semimajor axes. The law corresponds to regularities present in the planetary data recorded by Tycho Brahe. Johannes Kepler's extraction of these regularities from Brahe's data can be regarded as an early example of pattern analysis.

There are various models to formalise such pattern analysis problems in diverse degrees of generality. A very prominent one is Valiant's learning model [15, 16] that is outlined in the following.

We are given an *input space* $\mathfrak{X} \subset \mathbb{R}^n$. Usually we think of $\mathfrak{X}$ as being a set of encodings of instances or objects in the learner's world. Examples are rectangles in the Euclidean plane $\mathbb{R}^2$, two 2-dimensional arrays of binary pixels of a fixed width and height when it comes to recognising characters, or simply Boolean vectors of length n. The input space is the data source in this model. To this end, a random element $X \in \mathfrak{X}$ is given. It induces an arbitrary distribution P_X on $\mathfrak{X}$.

A *concept* over the input space $\mathfrak{X}$ is a $+1/-1$-valued function on $\mathfrak{X}$ or equivalently a subset of $\mathfrak{X}$. A *concept class* $\mathcal{C}$ is a collection of concepts. Examples are all rectangles in the Euclidean plane $\mathbb{R}^2$, pixel representations of a given alphabet, and all Boolean monomials of a fixed length k over the Boolean variables $x_1, x_2, \ldots, x_n$.

An algorithm A is a PAC learner of the concept class $\mathcal{C}$ by a hypothesis class $\mathcal{H}$, which usually comprises $\mathcal{C}$, if for every accuracy $\epsilon > 0$ and every confidence $\delta > 0$ there is minimal sample size $\mathrm{m}_A(\epsilon, \delta)$ such that for every target concept $g \in \mathcal{C}$, all $m \geq \mathrm{m}_A(\epsilon, \delta)$, and all distributions P_X the following property is satisfied. Let A be given access to a *learning sample*

$$U^{(m)} := ((X_1, g(X_1)), (X_2, g(X_2)), \ldots, (X_m, g(X_m))) \tag{1}$$

of length m, where $(X_1, X_2, \ldots, X_m)$ is a sample drawn independently from $\mathfrak{X}$ according to the distribution P_X. Then A outputs with probability at least $1 - \delta$ a hypothesis $H := A\left(U^{(m)}\right) \in \mathcal{H}$ satisfying $\mathrm{err}(H) \leq \epsilon$. This probability is taken over the random learning samples according to (1) and any internal randomisation, if the learning algorithm is a probabilistic one. The *error* $\mathrm{err}(h)$ of any hypothesis $h \in \mathcal{H}$ is defined by $\mathrm{P}\left(h(X) \neq g(X)\right)$. The preceding condition is sometimes referred to as *consistency* of the learning algorithm A.

In order to devise a PAC learner, it is reasonable to output a hypothesis h that performs faultless on the learning sample (1). To ensure that this will work, especially to avoid what is called *overfitting*, it is, moreover, necessary to bound the *capacity* of the hypothesis class $\mathcal{H}$. Very popular capacity measures are the Vapnik-Cervonenkis dimension [17–19] and the Rademacher complexity [20]. For an overview see [21, 22].

PAC learning can be canonically generalised to what might be called *pattern analysis* or *pattern extraction*. Starting point is the observation that the second

components Y_i of the learning sample

$$U^{(m)} := ((X_1, Y_1), (X_2, Y_2), \ldots, (X_m, Y_m)) \tag{2}$$

need not always be totally depend on the first components X_i. Again we restrict ourselves to *classification problems*, that is to the cases, where the so-called *output variables* Y_i take values in the *output space* $\mathfrak{Y} = \{-1, +1\}$, The product $\mathfrak{U} := \mathfrak{X} \times \mathfrak{Y}$ is denoted as *learning universe*. Using a random variable $U = (X, Y) \in \mathfrak{U}$, it is regarded as source of data. The first component X of U is called *input element*, the second component Y the *output variable*. Analogous to the cases of PAC learning, the distribution P_U with the random element U induced on the learning universe is not determined, but sometimes it has *guaranteed qualities*.

A pattern analysis algorithm A *by* a *pattern class* $\mathcal{P} \subset \mathrm{Map}(\mathfrak{U}, \{-1, +1\})$ takes a learning sample (2) as input. It computes a pattern $A(U^{(m)}) \in \mathcal{P}$ that "approximates" the *risk infimum* $\operatorname{risk} \mathcal{P} := \inf_{\pi \in \mathcal{P}} \operatorname{risk} \pi$ of the class $\mathcal{P}$ in the sense of consistency defined below. The risk of a pattern $\pi \in \mathcal{P}$, which is denoted by $\operatorname{risk} \pi$, in turn is defined to be the expected value $\mathrm{E}\, \pi(U)$ of $\pi(U)$. On overview on this setting is given in [22].

In this paper, we make only use of pattern classes consisting of patterns of the type $\ell_{0/1}(y, h(x))$, where h ranges over a hypothesis class $\mathcal{H} \subset \mathrm{Map}(\mathfrak{X}, \{-1, +1\})$, and $\ell_{0/1}$ is the so-called *0/1-loss function:* $\ell_{0/1}\big(y_1, y_2\big) = 1 - \delta\big(y_1, y_2\big)$, where δ is the Kronecker function.

An example for a guaranteed quality mentioned above is that Y equals $g(X)$, for some target concept g belonging to the concept class $\mathcal{C}$. If the pattern class is formed by means of a hypothesis class $\mathcal{H} \supseteq \mathcal{C}$, then $\operatorname{risk} \pi = \operatorname{err} h$ provided that $\pi(x, y) = \ell_{0/1}(y, h(x))$. That way PAC learning is a special case of pattern extraction. From now on we identify the pattern $\pi(x, y) = \ell_{0/1}(y, h(x))$ with the hypothesis $h(x)$, and consequently the pattern class $\mathcal{P}$ with the hypothesis class $\mathcal{H}$.

A pattern analysis algorithm A by $\mathcal{H}$ is called *consistent*, if for every accuracy $\epsilon > 0$ and every confidence $\delta > 0$ there is a minimal sample size $\mathrm{m}_A(\epsilon, \delta)$ such that for all $m \geq \mathrm{m}_A(\epsilon, \delta)$, and all distributions P_U the following condition is fulfilled. Taking the learning sample (2) as input, A outputs with probability at least $1 - \delta$ a hypothesis $H := A\left(U^{(m)}\right) \in \mathcal{H}$ satisfying

$$\operatorname{risk} H \leq \operatorname{risk} \mathcal{H} + \epsilon. \tag{3}$$

The problem with the risk of a hypothesis is that it cannot be calculated since the distribution P_U is not determined. Consequently, one cannot try to compute a hypothesis of minimal risk. The empirical risk minimisation induction principle *ERM* recommends a pattern analysis algorithm to choose a hypothesis h that minimises the *empirical* risk

$$\operatorname{risk}_{\mathrm{emp}}\left(h \,\middle|\, U^{(m)}\right) := \frac{1}{m} \sum_{i=1}^{m} \ell_{0/1}\left(Y_i, h(X_i)\right) \tag{4}$$

on the learning sample (2). A pattern analysis algorithm A obeying ERM is consistent, if, for example, the Rademacher complexity $\mathrm{rc}_m(\mathcal{H})$ of $\mathcal{H}$ is an $o(1)$. In this cases the empirical risk of the output of A is a consistent estimator of the risk infimum in the sense of mathematical statistics.

In this paper ERM means that one has to minimise the number of misclassifications on the learning sample (2). Practically, one has to ensure that for sufficiently small accuracy and confidence – say $\epsilon = 0.005$ and $\delta = 10^{-6}$ – the learning sample length suffices to fulfil (3).

To get an idea of what the Rademacher complexity $\mathrm{rc}_m(\mathcal{H})$ of a finite $\mathcal{H}$ with respect to samples of length m means, let us mention that $\mathrm{rc}_m(\mathcal{H}) \leq \sqrt{2 \ln |\mathcal{H}|}/\sqrt{m}$ [23].

3 Using Learning Techniques to Evaluate Metric Sets

In this section we describe how to approximate a comprehensive software quality assessment scheme based on a set of n metrics by a restricted scheme supported by the "best" ν-subset ($0 < \nu < n$) of these metrics in terms of learning techniques. To this end, we assign to a family of *parametrised* comprehensive schemes a concept class in the sense of PAC learning. For each ν-subset, the corresponding family of parametrised restricted schemes is, moreover, mirrored by a hypothesis class. We evaluate the performance of the approximation using the risk infimum of the best hypothesis class (see (3)).

In the following, we define the learning setup: The input space $\mathfrak{X}$ equals the Cartesian product of n intervals $[0, c_1]$, $[0, c_2]$, ..., $[0, c_n]$, that are the ranges of the n metrics. Thus each behavioural entity is represented by a vector of length n.

The concept class $\mathcal{C}$, which is equivalent to the parametrised comprehensive assessment schemes, consists of all concepts g such that for all $x \in \mathfrak{X}$

$$g(x) = +1 \quad \Longleftrightarrow \quad x_i > \tau_i \quad \text{for at most } k \text{ of the indices } \{1, 2, \ldots, n\},$$

and $g(x) = -1$ otherwise. Therein the *parameter* $(\tau_1, \tau_2, \ldots, \tau_n)$ is any element of $\mathfrak{X}$, and $k \in \{0, 1, \ldots, n-1\}$ is a constant. The concept g is equivalent to the following comprehensive software quality assessment scheme: A behavioural entity is positively evaluated, if and only if at most k of the n metrics violate their quality threshold given by $(\tau_1, \tau_2, \ldots, \tau_n)$.

We restricted ourselves to a "reasonable" concept $g_0 \in \mathcal{C}$ from the point of view of software quality assessment, rather than to learn the whole concept class in the sense of PAC learning. Our concept g_0 is determined by n threshold values $\gamma_1, \gamma_2, \ldots, \gamma_n$, one for each metric, based on our expertise in software testing. Then the learning samples were assumed to be drawn according to (1), with the target concept g being g_0 as guaranteed quality of the distribution P_U (see Section 2.2).

For each ν-subset $i_1 < i_2 < \ldots < i_\nu$ of $\{1, 2, \ldots, n\}$, we define the elements h of the hypothesis classes $\mathcal{H}(i_1, i_2, \ldots, i_\nu)$ determining a restricted scheme by

$$h(x) = +1 \quad \Longleftrightarrow \quad x_{i_j} > \tau_j \quad \text{for at most } \kappa \text{ elements } j \text{ of the set } \{1, 2, \ldots, \nu\},$$

where $\tau_j \in [0, c_{i_j}]$, for $j = 0, 1, \ldots, \nu - 1$, are the parameters of the hypothesis, and $1 \leq \kappa \leq \nu$ is a constant. Thus each ν-subset determines one restricted *model* of software assessment.

Clearly, learning a hypothesis of the above kind amounts to computing the ν *hypothesis thresholds* $\tau_1, \tau_2, \ldots, \tau_\nu$ from the training data. These thresholds need not be the same as the corresponding ones in the sequence $\gamma_1, \gamma_2, \ldots, \gamma_n$. This is due to the fact that we approximate n metrics by ν ones.

Without proof we notice that for a moderately large number n of metrics the pattern classes defined above are of relatively small capacity such that learning samples of reasonable size suffice to ensure (3) for acceptable accuracy and confidence.

What is a reasonable course of action in our learning setup to approximate a larger scheme by a smaller one and to evaluate the performance of the approximation? The one that follows is rather standard.

1. Represent all available behavioural entities as a vector of length n using the n metrics.
2. Classify them by the concept $g_0 \in \mathcal{C}$.
3. Randomly divide these data set into three parts: a training set (50%), a validation set (25%), and a test set (25%). This is because there are in fact two goals that we have in mind:
   Model selection: estimating the performance of the ν-subsets of our n-set of metrics to choose the best one.
   Model assessment: having chosen a final ν-subset of metrics, estimating the infimum risk.
4. In general, the training set is used to fit the models. In our case this means to compute for each ν-subset $\{i_1, i_2, \ldots, i_\nu\}$ of the index set $\{1, 2, \ldots, n\}$ a hypothesis $h\,(i_1, i_2, \ldots, i_\nu) \in \mathcal{H}\,(i_1, i_2, \ldots, i_\nu)$ in terms of its hypothesis thresholds that minimises the empirical risk (see (4)) on the training data.
5. Choose a best hypothesis $h\left(i_1^{(0)}, i_2^{(0)}, \ldots, i_\nu^{(0)}\right)$ on the validation set. This is done by computing the empirical risks of all hypotheses $h\,(i_1, i_2, \ldots, i_\nu)$ found in Step 4 on the validation data, the so-called *validation errors*.
6. Calculate the empirical risk of $h\left(i_1^{(0)}, i_2^{(0)}, \ldots, i_\nu^{(0)}\right)$ on the test set, the so-called *test error*, thus estimating the risk infimum risk $\mathcal{H}\left(i_1^{(0)}, i_2^{(0)}, \ldots, i_\nu^{(0)}\right)$ that in turn measures how well n metrics can be approximated by ν ones.

4 Application

To evaluate the practicability of our approach, we performed an experiment. In this experiment, we applied our approach to investigate whether it is possible to approximate a set of four metrics by a minimised set of one or two metrics only.

4.1 Metrics

In the following, the four metrics used in the experiment are introduced in more detail. The metrics are selected to capture different perceptions of complexity of behaviour within a TTCN-3 test suite with regards to the maintainability characteristic and its analysability subcharacteristic of the refined quality model for test specifications [7]. In TTCN-3, test behaviour is specified by test case, function, and altstep constructs. We start by describing a measure called *Number of Statements*.

Metric 1 (Number Of Statements NOS). The number of statement count NOS is mostly self explaining. Unlike the common lines of code (LOC) measure, counting the number of statements ignores information regarding the code itself such as the code's formatting or comments while retaining an intuitive measure of the code length.

Even though NOS delivers a measure for the code length per behavioural entity, it does not deliver any statement about code complexity. It is missing a sense of behavioural complexity. McCabe's *cyclomatic complexity* [10] attempts to deliver this, essentially by counting the number of branches of the control flow graph and thus penalising conditional behaviour.

Metric 2 (Cyclomatic Complexity $v(G)$). The cyclomatic complexity $v(G)$ of a control flow graph G can be defined[1] as:

$$v(G) = e - n + 2$$

In this formula, e denotes the number of edges and n is the number of nodes in G.

While the cyclomatic complexity $v(G)$ penalises conditional behaviour, it is missing another factor that comprises code complexity: deeply nested branches are not penalised any different than flat branches. For example, a conditional nested within another conditional is penalised the same as two subsequent conditionals even though nested conditionals obviously complicate things. Thus, we chose to add a simple nesting level metric to our set of metrics.

Metric 3 (Maximum Nesting Level MNL). The maximum nesting level MNL is obtained by inspecting all conditionals within a test behaviour and counting their nesting levels. For example, an if-statement within an if-statement would yield the nesting level 2. The maximum nesting level denotes the highest nesting level measured per behavioural entity.

[1] Several ways of defining $v(G)$ can be found in literature. The above definition assumes that G has a single entry and a single exit point. In the presence of several exit points, this assumption can be maintained by adding edges from all exit points to a single exit point.

Since structured test behaviour may invoke other callable behaviour (e.g. by calling other functions), the complexity of each code fragment also depends on the complexity resulting by calls to such other behavioural entities. For developers, deeply nested call structures can be bothersome as they have to look up and understand each called behaviour within the code piece in front of them when working on this code. The *Maximum Call Depth* provides such a measure.

Metric 4 (Maximum Call Depth MCD). The maximum call depth MCD is obtained by analysis of the call graph[2]. For each behaviour A, the corresponding graph of behaviours called by A is calculated recursively to include indirect calls (i.e. the call relation is transitive); in this graph, the length of each path starting from A is measured and the resulting MCD value is the length of the longest distinct path. If the path contains a cycle due to a recursive call, the MCD value is ∞.

We calculate these four metrics for each behavioural entity (i.e. test case, function, altstep) of a TTCN-3 test suite and use the vector of calculated metric values to obtain an overall classification of the quality of a test behaviour with respect to the quality sub-characteristic *analysability*. To obtain such an overall classification, for each behaviour we compare the calculated metric values against corresponding thresholds. Each element of the vector may be classified as positive, i.e. does not violate its corresponding threshold value, or negative, i.e. does violate the corresponding threshold value. The overall classification of a behavioural entity is again a positive or negative verdict that depends on how many of the elements of the corresponding vector are classified as positive (indicating a good quality) or negative (indicating a bad quality) respectively.

4.2 Experimental Settings

To obtain a reasonable amount of data for applying and assessing our learning approach, we performed an experiment with several huge test suites that have been standardised by the *European Telecommunications Standards Institute* (ETSI). The first considered test suite is Version 3.2.1 of the test suite for the *Session Initiation Protocol* (SIP) [24], the second is a preliminary version of a test suite for the *Internet Protocol Version 6* (IPv6) [25]. Together, both test suites comprise 2276 behavioural entities and 88560 LOC.

The data used in this experiment was computed by our TRex TTCN-3 tool [26, 27] using the metric thresholds given in Table 1. Based on our TTCN-3 experience, these basic thresholds were determined along the lines of the GQM approach mentioned in Sect. 2.1. TRex calculated a vector containing the values of the four metrics for every behavioural entity. Concerning the overall classification, we investigated two different scenarios:

[2] In the call graph, a directed edge from node A to node B indicates that behaviour A calls behaviour B.

Scenario 1. In the first scenario, every metric in the vector must be classified as positive to get a positive classification for this behavioural entity, i.e. a concept $g_0(x) = +1 \iff \forall i : x_i \leq \gamma_i$, where x_i are the metric values computed by TRex, γ_i are the corresponding metric thresholds as given in Table 1 and $i \in [1, 4]$. Using these thresholds, this scenario results in nearly 50% negative examples, i.e. behavioural entities that have a negative overall classification.

Scenario 2. In the second scenario, only three of the four metrics must be classified as positive to get an overall positive classification for the behavioural entity, i.e. a concept $g_0(x) = +1 \iff x_i > \gamma_i$, for at most one of the indices $i \in [1, 4]$. Using the same thresholds γ_i as in Scenario 1 (Table 1), this scenario leads to approximately 13% negative examples.

Table 1. Metric thresholds used for generating the data

Metric	γ
NOS	14
$v(G)$	4
MNL	3
MCD	10

Preprocessing
We implemented our learning approach and applied it to the data generated by TRex. For conducting the experiments, we randomly divided the data from the 2276 behavioural entities into three parts: The first part is used for learning and contains 50% of the behavioural entities. The second part contains 25% of the entities, this part is used for validation. Finally, the third part contains the remaining 25% and is used for testing. We have done such a partitioning for every experiment to have independent data sets.

The Different Experiments
We examine the best and the two best metrics that yield the closest approximation of the vector of four for every scenario, i.e. we try to find one metric and a combination of two metrics that predict the overall classification as good as possible.

For using just one metric, this means we have to find an occurring threshold. Therefore, we begin with the smallest possible threshold that divides all values of one metric in positive and negative examples. In each step we have the hypothesis thresholds τ_i and the positive (the metric value $\leq \tau_i$) and the negative (the metric value $> \tau_i$) examples in the hypothesis class $\mathcal{H}$. Then, we compare this with the overall classification, i.e. the concept class $\mathcal{C}$, and count the number of misclassifications. Performing the same steps for all possible thresholds

we get the threshold that results in the smallest error. Afterwards, we proceed with the next metric and do the same. Finally we compare the metric/threshold combinations and choose the best. This means, we are looking for the smallest error in all metrics using a specific threshold.

For using two metrics, we try to find a combination of thresholds of two metrics that leads to the smallest error with respect to the overall classification. We do exactly the same as above, not searching a specific threshold but a threshold pair. So, we are searching in each of the possible metric combinations for the best threshold which leads to the smallest error.

4.3 Experimental Results

By investigating the two scenarios and for each scenario two different approximations (using either one or two metrics respectively), we obtain four different results.

Results for Learning the "Best" Metric in Scenario 1
For Scenario 1 (i.e. all four metrics must be classified as positive to yield a positive overall classification) and the assumption that the overall classification can be approximated by only one of the four metrics, the resulting data is provided in Table 2. If we select the biggest threshold it is clearly evident that all examples are classified as positive. Then the error is equal to the proportion of the negative examples. For a better overview we append this to the result table as $allNeg$. For any metrics, the resulting error is very big if it is the only metric used to predict the overall classification. For the best metric the risk infimum estimated on the testset is 19.61%. As usual we have estimated the risk only for the best metric. Hence, it is not advisable to replace the four metrics by only one metric. However, it is remarkable that the threshold chosen by the algorithms for the MCD metric is exactly the same as used for data generation. The other located thresholds are at least similar to those used for data generation.

Table 2. Learning the "best" metric in Scenario 1

Metric	Empirical risk	τ	Validation error	Test error
MCD	18.93	10	18.09	19.61
NOS	19.19	7	18.97	-
$v(G)$	29.80	2	34.22	-
MNL	33.57	2	37.41	-
$allNeg$	44.35	-	45.57	-

Results for Learning the "Best Two" Metrics in Scenario 1
When trying to approximate all four metrics using just two metrics, the results for Scenario 1 look like provided in Table 3. If the combination of the two metrics

NOS and MCD is chosen, it is possible to reproduce the overall classification with a quite small risk infimum of 1.94%. The thresholds that have lead to this small error are exactly the same as chosen to generate the dataset. For all other combinations of metrics, the error is significantly larger and it is not advisable to use them instead.

Table 3. Learning the "best two" metrics in Scenario 1

Combination	Empirical risk	τ_1	τ_2	Validation error	Test error
NOS, MCD	2.11	14	10	1.76	1.94
$v(g)$, MCD	7.98	3	10	7.56	-
MNL, MCD	11.84	3	10	12.30	-
MNL, NOS	19.73	3	7	18.80	-
$v(g)$, NOS	19.82	3	7	18.98	-
$v(g)$, MNL	30.87	2	3	30.93	-
$allNeg$	44.74	-	-	43.94	-

Results for Learning the "Best" Metric in Scenario 2
In comparison with the first scenario, we here obtained a different ordering in the metrics, i.e. where MCD was the best metric in Scenario 1, it is now $v(g)$ (Table 4). The learned threshold for the first metric is exactly the same like the one chosen to generate the dataset. The estimated risk infimum for the best metric $v(G)$ is 5.57%. Similar to Scenario 1, a risk infimum of 5.57% is too high to use just one metric to compute the overall classification.

Table 4. Learning the "Best" Metric in Scenario 2

Metric	Empirical risk	τ	Validation error	Test error
$v(G)$	5.75	4	5.94	5.57
NOS	8.94	22	8.22	-
MNL	9.02	3	9.97	-
MCD	13.01	14	12.94	-
$allNeg$	13.01	-	12.76	-

Results for Learning the "Best Two" Metrics in Scenario 2
As in the previous experiment, the set of the best metrics as shown in Table 5 is a different one to Scenario 1. The estimated risk infimum for the combination $v(G)$, NOS that has the smallest validation error is 6.47%. Although the highest empirical risk in this experiment is 13.12% and therefore smaller than for the corresponding experiment in Scenario 1, the smallest empirical risk is much greater than for Scenario 1 and therefore less significant. Also, for NOS, the

learned threshold $\tau_2 = 27$ differes very much from the threshold $\gamma = 14$ that was used to generate the data.

Table 5. Learning the "Best Two" Metrics in Scenario 2

Combination	Empirical risk	τ_1	τ_2	Validation error	Test error
$v(G)$, MNL	5.02	4	4	5.11	-
$v(G)$, NOS	5.46	4	27	4.93	6.47
$v(G)$, MCD	5.55	4	14	5.46	-
MNL, NOS	7.57	3	20	6.51	-
NOS, MCD	8.63	20	14	8.63	-
MNL, MCD	9.42	3	14	8.27	-
$allNeg$	13.12	-	-	12.85	-

5 Summary and Outlook

In the previous sections, we have presented a machine learning based method for the minimisation of metric sets. In this method, tuples of this metric set are first used to classify each measured entity of the software under investigation as either "good" or "bad". This classification is determined by threshold values for each metric in the set. The presented approach then attempts to approximate the same classification with a smaller set of metrics to reduce the number of necessary metrics in this set and to identify metrics with overlapping information.

We have tried this approach in two different classification scenarios on a set of four metrics substantiating the analysability quality characteristic of TTCN-3 test specifications. In the first scenario, the classification has been obtained by requiring that every metric value in the observed set must be smaller than its corresponding threshold value to be classified as "good". Given this setting, we have tried to approximate the entity classifications by using one single metric and by using a set of two metrics. According to the results, the risk of misjudging the classification of a behavioural entity by using just one single metric is quite high. By using a combination of two metrics, i.e. the NOS and MCD metrics, the approximated classification is very reasonable.

In the second scenario, only three of four metric values in the set have been required to be below their corresponding threshold values to be classified as "good". Again we have tried to approximate the classifications by using one single metric and a set of two metrics respectively. In both cases the risk of an incorrect assessment has been too high. The detailed reasons will be subject of further investigations.

In the experiments presented, we have applied our approach to metrics extracted from TTCN-3 test suites. We are currently working on quality assurance techniques for graphical languages such as the *Specification and Description Language* (SDL) and *Unified Modeling Language* (UML) which also includes metrics

for models. We expect that our presented approach will also deliver reasonable results for metric sets designed to work on models. In addition, we want to evaluate metric sets used on Java implementations.

So far, we have only used a small set of metrics for the evaluation and therefore time and space complexity of our algorithm was not yet an issue. However, to make our technique applicable to larger sets of metrics as well, we plan to investigate the complexity of our method in more detail and optimise it accordingly.

References

1. Fenton, N.E., Pfleeger, S.L.: Software Metrics. PWS Publishing Company, Boston (1997)
2. CMMI Product Team: CMMI for Development, Version 1.2. Technical Report CMU/SEI-2006-TR-008, Carnegie Mellon University, Software Engineering Institute (2006)
3. ISO/IEC: ISO/IEC Standard No. 15504: Information technology – Process Assessment; Parts 1–5. International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC), Geneva, Switzerland (2003-2006)
4. ETSI: 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 (February 2007)
5. Grabowski, J., Hogrefe, D., Réthy, G., Schieferdecker, I., Wiles, A., Willcock, C.: An introduction to the testing and test control notation (TTCN-3). Computer Networks **42**(3) (June 2003) 375–403
6. Zeiss, B., Neukirchen, H., Grabowski, J., Evans, D., Baker, P.: Refactoring and Metrics for TTCN-3 Test Suites. In Gotzhein, R., Reed, R., eds.: System Analysis and Modeling: Language Profiles. 5th International Workshop, SAM 2006, Kaiserslautern, Germany, May 31–June 2, 2006, Revised Selected Papers. Volume 4320 of Lecture Notes in Computer Science (LNCS)., Berlin, Springer (December 2006) 148–165
7. Zeiss, B., Vega, D., Schieferdecker, I., Neukirchen, H., Grabowski, J.: Applying the ISO 9126 Quality Model to Test Specifications – Exemplified for TTCN-3 Test Specifications. In Bleek, W.G., Raasch, J., Züllighoven, H., eds.: Software Engineering 2007. Volume 105 of Lecture Notes in Informatics (LNI)., Bonn, Gesellschaft für Informatik, Köllen Verlag (March 2007) 231–242
8. ISO/IEC: ISO/IEC Standard No. 9126: Software engineering – Product quality; Parts 1–4. International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC), Geneva, Switzerland (2001-2004)
9. Halstead, M.: Elements of Software Science. Elsevier, New York (1977)
10. McCabe, T.J.: A Complexity Measure. IEEE Transactions on Software Engineering **2**(4) (1976) 308–320
11. Watson, A.H., McCabe, T.J.: Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metricy. NIST Special Publication 500-235, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD, United States of America (1996)

12. Fan, C.F., Yih, S.: Prescriptive metrics for software quality assurance. In: Proceedings of the First Asia-Pacific Software Engineering Conference, Tokyo, Japan, IEEE-CS Press (December 1994) 430–438
13. Basili, V.R., Weiss, D.M.: A Methodology for Collecting Valid Software Engineering Data. IEEE Transactions on Software Engineering **10**(6) (1984) 728–738
14. Henry, S., Kafura, D., Harris, K.: On the Relationships Among Three Software Metrics. In: Proceedings of the 1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality, New York, NY, USA, ACM Press (1981) 81–88
15. Valiant, L.: A theory of learnability. Communications of the ACM **27**(11) (1984) 1134–1142
16. Valiant, L.: Deductive learning. Philosophical Transactions of the Royal Society London A **312** (1984) 441–446
17. Vapnik, V., Chervonenkis, A.Y.: On the uniform convergence of relative frequencies of events and their probabilities. Theory of Probabilty and its Applications **16**(2) (1971) 264–280
18. Blumer, A., Ehrenfeucht, A., Haussler, D., Warmuth, M.K.: Learnability and the Vapnik-Chervonenkis dimension. Journal of the ACM **36**(4) (1989) 929–969
19. Vapnik, V.: The Nature of Statistical Lerning Theory. Springer Verlag, New York (1995)
20. Koltchinskii, V.I., Pachenko, D.: Rademacher processes and bounding the risk of learning function. High Dimensional Probability II (2000) 443–459
21. Kearns, M.J., Vazirani, U.V.: An Introduction to Computational Learning Theory. MIT Press (1994)
22. Shawe-Taylor, J., Cristianini, N.: Kernel Methods for Pattern Analysis. Cambridge University Press (2004)
23. Massart, P.: Some applications of concentration inequalities to statistics. Annales de la Faculté des Sciences de Toulouse (2000) 245–303 volume spécial dédié à Michel Talagrand.
24. ETSI: Technical Specification (TS) 102 027-3 V3.2.1 (2005-07): SIP ATS & PIXIT; Part 3: Abstract Test Suite (ATS) and partial Protocol Implementation eXtra Information for Testing (PIXIT). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (July 2005)
25. ETSI: Technical Specification (TS) 102 516 V1.1 (2006-04): IPv6 Core Protocol; Conformance Abstract Test Suite (ATS) and partial Protocol Implementation eXtra Information for Testing (PIXIT). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (April 2006)
26. Baker, P., Evans, D., Grabowski, J., Neukirchen, H., Zeiss, B.: TRex – The Refactoring and Metrics Tool for TTCN-3 Test Specifications. In: Proceedings of TAIC PART 2006 (Testing: Academic & Industrial Conference – Practice And Research Techniques), Cumberland Lodge, Windsor Great Park, UK, 29th–31st August 2006., IEEE Computer Society (August 2006)
27. TRex Team: TRex Website. `http://www.trex.informatik.uni-goettingen.de` (2007)