Empirical Software Engineering (2021) 26: 86
https://doi.org/10.1007/s10664-021-09984-2

Weighted software metrics aggregation and its application to defect prediction

Maria Ulan[1] · Welf Löwe[1] · Morgan Ericsson[1] · Anna Wingkvist[1]

Accepted: 26 May 2021 / Published online: 23 June 2021
© The Author(s) 2021

Abstract

It is a well-known practice in software engineering to aggregate software metrics to assess software artifacts for various purposes, such as their maintainability or their proneness to contain bugs. For different purposes, different metrics might be relevant. However, weighting these software metrics according to their contribution to the respective purpose is a challenging task. Manual approaches based on experts do not scale with the number of metrics. Also, experts get confused if the metrics are not independent, which is rarely the case. Automated approaches based on supervised learning require reliable and generalizable training data, a ground truth, which is rarely available. We propose an automated approach to weighted metrics aggregation that is based on unsupervised learning. It sets metrics scores and their weights based on probability theory and aggregates them. To evaluate the effectiveness, we conducted two empirical studies on defect prediction, one on ca. 200 000 code changes, and another ca. 5 000 software classes. The results show that our approach can be used as an agnostic unsupervised predictor in the absence of a ground truth.

Keywords Software assessment · Quantitative methods · Defect prediction · Software metrics · Aggregation · Weighting

Communicated by: Burak Turhan

✉ Maria Ulan
maria.ulan@lnu.se

Welf Löwe
welf.lowe@lnu.se

Morgan Ericsson
morgan.ericsson@lnu.se

Anna Wingkvist
anna.wingkvist@lnu.se

[1] Data-driven Software and Information Quality Group, Centre for Data Intensive Sciences and Applications, Linnaeus University, 351 95 Växjö, Sweden

1 Introduction

Quality assurance is usually done with a limited budget. Hence, its activities must be performed as efficiently as possible. Understanding which software artifacts are likely to be problematic helps to prioritize activities and to allocate resources accordingly. To gain this knowledge, software quality assessment employs weighted software metrics. So-called quality models define the aggregation of (weighted) metrics values for an artifact to a single score value for this artifact. Suitable quality models, their metrics, and their weights differ for different quality assessment goals.

A software metric weight can be interpreted as an indicator of the relative importance of this metric with respect to the quality assessment goal. Unfortunately, there is no formal definition of the notion of the "relative importance" of a metric. It is not obvious, e.g., what to expect for an aggregated score of two metrics if one metric is twice as important as the other. And what does "twice as important" exactly mean then?

Moreover, the relative importance of a single metric depends on its (possibly nonlinear) relation with other metrics. Relations include, but are not limited to, correlations between metrics. Adding a metric to a quality model that contains already a highly correlated metric, reduces their importance relative to a third uncorrelated one. As correlation is not transitive, it is hard to account for the correlation between metrics when defining metrics weights, let away to account for all possible non-linear or even non-functional relations between the metrics of a quality model.

That said, setting and interpreting weights properly is hard even for experts. This results in subjective quality models that are difficult to interpret. Still, weights are sometimes suggested to be determined manually by experts, e.g., based on the Analytic Hierarchy Process (Saaty 2008), the Delphi method (Linstone et al. 1975), or expert surveys.

However, weights can also be determined using (linear) regression or other supervised machine learning approaches (Wang and Stanley 1970). These approaches require training instances, i.e., known metric and defect values (the so-called ground truth) for a set of software artifacts, which is hard to obtain. Moreover, some regression approaches make strong assumptions about the metrics' independence, which is not realistic, as discussed before. Finally, some approaches, such as weighted linear regression, do not make any sense, don't have a meaningful interpretation, if metrics do not have the same scale, which is usually the case.

We suggest defining weights automatically in an unsupervised learning approach considering observed metrics values, but without requiring ground truth knowledge on training instances. Weights are automatically set regarding their usage in aggregation and the interpretation of the aggregated scores. More specifically, we take the metrics' joint distributions and transform metrics values into aggregated scores that can be expressed and interpreted as probabilities. We consider weights to represent the relative importance of metrics in the aggregated score.

While there are different quality assessment goals, the present paper focuses on "error-proneness" as it gives an objective ground truth that is relatively easy to measure, at least retrospectively, by looking at the actual bugs found later. As our focus is on weighted aggregation, we try to avoid discussions on what quality means in other contexts.

In the context of defect prediction, quality models are also referred to as defect prediction models. They are functions mapping metric values of a software artifact to a measure of its likelihood to be defective. These models can also be seen as aggregation models implementing, e.g., weighted sum aggregation operators where weights are defined by regression.

We compare our aggregation models to the related quality models in two different scenarios, i.e., change-level and class-level defect prediction. They complement each other to improve the quality of the upcoming release of the software. The difference between them is the development phase they are employed in. Class-level defect prediction is usually conducted before a release; change-level defect prediction is a continuous activity that is conducted when each change is submitted.

It is impractical to inspect all software artifacts that are predicted as buggy, since in general, resources for quality assurance activities are limited. Hence, especially in practical settings, it is important to obey the effort of applying quality assurance activities guided by defect prediction. For example, the effort needed to inspect the large file having lots of bugs could be smaller than the effort needed to inspect several smaller files, each having a small number of bugs, but statistically a higher bug density. Hence, these smaller files could be prioritized for review. We therefore focus on defect prediction models that also regard the effort to react—referred to as *effort-aware evaluation* (Kamei et al. 2013; Mende and Koschke 2010). We define the performance of approaches as a percentage of all defect-including software artifacts they can identify spending a fixed percentage of the total available time budget.

However, we do not argue in favor or against this or any other prioritization, and we do not even aim to build the best performing defect predictor. In contrast, to illustrate our unsupervised aggregation approach, we study whether or not this quality goal agnostic approach could compete with the performance of existing (often supervised) models tailored for effort-aware defect prediction.

Our key contributions are the following:

1. A formal definition of weights in the context of software metrics aggregation.
2. An automated approach to weighted software metrics aggregation using the concepts of a joint probability distribution, and the concepts of correlation and entropy to determine weights. This main contribution defines a general unsupervised aggregation approach as a suitable alternative to supervised approaches that is applicable when a ground truth is lacking.
3. An application and effort-aware evaluation of this approach in the context of software defect prediction.

As a result, we will conclude that our unsupervised, quality-goal-agnostic aggregation is on par with the compared supervised defect prediction approaches in the benchmark systems. In the future, we will also compare it to supervised predictors in other problem domains of software engineering and beyond.

The remainder of the paper is structured as follows. We summarize the background, related work, and the challenges in Section 2. To address these challenges, we formally define metrics interactions and the interpretations of weights in Section 3. We introduce our approach of weighted quality scoring in Section 4. In Section 5, we apply this approach to defect prediction and evaluate it in two empirical studies on code-change- and software-class-level defect prediction. Finally, Section 6 concludes the research and points out directions of future work.

2 Background and Related Work

The problem of defect prediction is embedded in two more general areas of research: software metrics and quality models and multi-criteria decision making. We could understand

defect prediction models as special quality models used to assess the absence of bugs in software artifacts as a quality based on suitable software metrics. We could also understand defect prediction as an instance of multi-criteria decision making where we decide, which software artifact is prone to contain defects. In both cases, defect prediction inherits problems connected to the dependencies between criteria or metrics and connected to the subjective setting of weights, as detailed in Sections 2.1 and 2.2, respectively. We will also relate our work to specific approaches of defect prediction in Section 2.3. Finally, we set ourselves apart from related approaches of dimensionality reduction in Section 2.4.

2.1 Software Metrics and Quality Models

The notion of *quality* is diverse, complex, and ambiguous. There is no generally accepted definition of quality; it depends on viewpoints and goals (Garvin 1984). The definition of quality might differ considerably for different goals. To this end, software artifacts being free from defects is one possible quality goal, along with others, such as, e.g., performance, maintainability, or security. There are different general approaches to aid quality management, such as the *Factor-Criteria-Metric* and the *Goal-Question-Metric* methods. Both define quality models that use *software metrics* to provide quantitative data (Fenton and Bieman 2014) and, depending on the view and the goal, different metrics become relevant. The software engineering community has proposed and discussed a great variety of such software metrics (Chidamber and Kemerer 1994; Henderson-Sellers 1995; Martin 2002), and many of them are validated (Basili et al. 1996). However, because of different views and goals, companies quite often use their own metrics and customized quality models (Wagner 2013).

We neither aim to define (yet another notion of) quality nor (yet another set of) software metrics. We assume that there is a quality model defined by domain experts connecting a general quality goal to a set of metrics. Also, we expect that the experts know at least for some metrics whether high metrics values contribute positively or negatively to the goal. For instance, high values of Lines of Code (LOC) are known to contribute negatively to a quality goal like maintainability while a high degree of API documentation contributes positively to the same goal. Note, that for the same metric the contribution direction might be different for a different goal. Finally, our approach does not rely on the expert input of metrics directions relative to a quality goal.

We focus on metrics value aggregation, i.e., on how one can combine different metrics into a single score, that preserves properties of original data with a minimum of human supervision. Such a score supports decision making, e.g., ranking different artifacts, which is otherwise difficult based on a set of contradicting metric values.

Many studies discuss hierarchical quality models that rely on several metrics for quality assessment (Wagner 2013). These studies mainly consider the relations between hierarchical levels, but not the relationships between metrics. They either consider metrics to be independent or suggest ways to mitigate the impact of those that are highly correlated. For example, Gil and Lalouche (2017) show a strong pairwise correlation of the metrics in the (Chidamber and Kemerer 1994) metrics suite with code size. Jiarpakdee et al. (2020) show that the interpretation of defect prediction models heavily relies on software metrics that are used to construct them, and propose *AutoSpearman* to mitigate correlated metrics.

Instead of mitigation, we suggest basing prediction models on the joint distribution of metrics values. In addition to pairwise correlation, this approach also handles non-linear and multiple dependencies between metrics.

Aggregation, i.e., combining different metrics values to get an overall score, is an essential part of any quality measurement method. However, the choice of aggregation methods for software quality models is rarely justified. Apart from the widely accepted standards, e.g., ISO/IEC 25010 (2010) and IEEE 610.12 (1990), several other quality models have been proposed (Bansiya and Davis 2002; Letouzey and Coq 2010; Baggen et al. 2012; Wagner et al. 2015). These models use weights to represent the metrics' relative importance contributed to quality. Common ways to determine such weights are experts' opinions or surveys. This makes quality assessment subjective due to the possible lack of consensus among experts, especially, if a notion of quality is multifaceted and vague and might have different meanings for different experts (Garvin 1984).

We suggest an automated aggregation method that adopts a probabilistic weight-learning from the joint distribution of metrics instead of the subjective weight-setting in related studies. In particular, we set two types of weights: one represents the dependency between metrics and the other one their entropy. In contrast to relying on experts' opinions to define the weights, our approach is objective, since based solely on data, and it can be run automatically. Hence, it allows us to update the models easily when more data becomes available.

2.2 Multi-Criteria Decision Making

Multi-Criteria Decision Making (MCDM) chooses the best feasible solution from a set of solutions by explicitly evaluating multiple, potentially interacting, and even conflicting solution criteria.

Even in MCDM, a frequent assumption is that criteria are independent of each other. However, it was shown in many decision problems that criteria are indeed dependent (Carlsson and Fullér 1995; Saaty 1996). The problem has been addressed in different ways. Carlsson and Fuller use a fuzzy set theory to handle dependency. It is known that fuzzy logic is not always accurate. Hence, the results are interpreted based on assumptions. Saaty proposes the *analytic network process (ANP)* as extension of the *analytic hierarchy process (AHP)* based on the *supermatrix* approach. However, their complexity increases quadratic with the number of criteria that pairwise comparisons, which makes it impractical for human experts when facing even a moderate number of criteria.

Criteria weights reflect the importance of criteria and assume that the meaning of criteria "importance" is transparent and well understood by all decision-makers. As this is not guaranteed, criteria weights are often misunderstood and misused (Schenkerman 1991).

2.3 Defect Prediction

Some software defect prediction models aim to predict the *number* of defects in a software system based on software *product* metrics such as, e.g., Basili et al. (1996) who used the Chidamber and Kemerer (CK) metrics to predict buggy code components. Nikora and Munson (2004) measured the evolution of structural attributes in predicting the number of bugs introduced during software system development. Ohlsson and Alberg (1996) relied design metrics to identify error-prone modules.

Other approaches are based on software *process* metrics. For instance, Nagappan and Ball (2005) proposed a technique for early bug prediction based on relative code churn measures. Hassan (2009) introduced the concept of entropy of changes. Kim et al. (2007) proposed a bug cache algorithm that predicts future faults based on the location of previous faults. Some of the approaches are a mix of the previous two, e.g., Moser et al. (2008)

highlighted the superiority of process metrics in predicting buggy code, and Graves et al. (2000) observed that history-based metrics are more powerful than product metrics.

We utilize both product and process metrics, but we do not study the effectiveness of different software metrics for constructing defect prediction models. In contrast, we focus on metrics aggregation and what it could bring to defect prediction.

Module-order models predict a *ranking* of software artifacts ordered by the expected number of defects instead of directly predicting the number of defects (Khoshgoftaar and Allen 1999). Menzies et al. (2010) proposed a model that prioritized modules in ascending order according to their code size. D'Ambros et al. (2010) performed an extensive comparison of bug prediction approaches relying on process and product metrics. They showed that no technique based on a single metric performs better than a quality model using a combination of metrics. We apply our approach for metrics aggregation to their benchmark and compare it against their best performing approaches based on churn and entropy of source code metrics, and aggregate the same metrics.

Arisholm et al. (2010) suggest that the time spent reviewing a potentially buggy software artifact depends on its size and complexity. Therefore, the prediction model should result in a list of software artifacts ranked by the ratio of effort spent on the number of bugs found. Mende and Koschke (2009) proposed a performance evaluation strategy based on cumulative lift charts, where the software artifacts are ordered according to the prediction model on the x axis, and the cumulative number of actual defects on the y axis. Their effort-aware prediction performance P_{opt} assesses the fraction of defects that can be encountered when reviewing only a fraction of the software artifacts. D'Ambros et al. (2012) used LOC as a proxy for effort and rank the classes according to their defect density, i.e., the number of defects divided by the number of lines of code. In the ground truth chart, all software artifacts are ordered decreasingly by the actual bug density while in the prediction model, all software artifacts are ordered decreasing by the predicted bug density. P_{opt} is then one minus the normalized integral of the differences of the ground truth and the prediction model charts. We too evaluate effort-awareness P_{opt} of our aggregation approach.

The defect prediction models discussed so far focus on identifying defect-prone *code*. In practice, however, it is difficult to decide who should be assigned to review this code, since large files often have multiple authors. To address this challenge, Mockus and Weiss (2000) proposed a prediction model that focuses on identifying defect-prone code *changes*. Kamei et al. (2013) referred to the change-level defect prediction as "Just-in-Time Defect Prediction". Their model predicts the risk value of changes. It is 1 divided by the churn, the sum of code added and deleted in a change, if the code change includes a defect, and 0, otherwise. This way they predict defect-density instead of defect-proneness. Their supervised prediction model is trained on a benchmark with the ground truth of buggy changes known. Yang et al. (2016) used the same benchmark. They proposed 12 simple models, each using one software metric and ranking the changes in descendant order of the reciprocal of the metrics' values. The metrics "lines of code added" and "lines of code deleted" are not included since the sum of these metrics (i.e., churn) is used to define the "ground truth".

These 12 simple models are agnostics with respect to the actual defects, hence, we call them unsupervised models. They outperform the model of Kamei et al. (2013) and other supervised models. In their experiments, Yang et al. (2016) averaged the results across different projects of the benchmark. Fu and Menzies (2017) repeated the said experiment project-by-project. They concluded that supervised predictors sometimes perform better when trained and validated in the same project. Also, they proposed a simple supervised learner that selects the best-performing of the 12 predictors of Yang et al. (2016) for each project. This simple supervised predictor performs better than most unsupervised and

supervised models. They also recommended that *"future defect prediction research should focus more on simple techniques."* In contrast to Yang et al. (2016) and Fu and Menzies (2017) models, Liu et al. (2017) proposed an unsupervised model based on churn, i.e., to rank changes in descendant order according to the reciprocal of churn. Several alternative supervised models were proposed since, e.g., ensemble learning-based models (Yang et al. 2017), multi-objective models (Chen et al. 2018), and logistic regression-based models (Huang et al. 2019).

We evaluate whether or not an aggregation of change metrics, i.e., aggregating all metrics values, is an improvement over the existing (un-)supervised approaches based on a single metric. We use the same benchmark as Kamei et al. (2013) and compare our approach with the unsupervised models of Yang et al. (2016), the supervised model of Fu and Menzies (2017), and the churn-based model of Liu et al. (2017).

2.4 Dimensionality Reduction

Aggregation is somewhat related to but still different from dimensionality reduction (van der Maaten et al. 2009), i.e., the embedding of elements of a high-dimensional vector space in a lower-dimensional space. Approaches to dimensionality reduction include principal component analysis (PCA) (Pearson 1901), stochastic neighbor embedding (SNE) (Hinton and Roweis 2003), and its t-distributed variant t-SNE (van der Maaten and Hinton 2008). One could think that the special case of reducing a multi-dimensional vector space (of metrics) to just one dimension (of a quality score) is a problem equivalent to aggregation. However, dimensionality reduction only aims at preserving the neighborhood of vectors of the original space in the reduced space. In contrast to that, aggregation assumes a latent total order in the measured artifacts related to the orders in each (metric) dimension that is to be aggregated. The aim is to define a totally ordered score order based on the partial orders induced by the (metric) dimensions that matches the latent order of the measured artifacts. Consequently, the accuracy of dimensionality reduction can be evaluated based on the observed data, i.e., the elements of a high-vector space, while the accuracy evaluation of aggregation additionally needs an explicit ground truth order of the measured artifacts.

3 Metrics Dependence – Weights Interpretation

Our approach derives metrics weights automatically and regards metrics dependence. Therefore, we first discuss different types of metrics dependence and different possible interpretations of metrics weights. Then we choose a dependence type and weight interpretation before we introduce our actual approach to derive them in the next section. Note that the discussions apply to metrics (directly measurable in the software artifacts) and criteria (indirect metrics and the result of metrics/criteria aggregation) alike. From here on, we will use the metrics and criteria as synonyms referring to variables whose values are to be aggregated.

3.1 Metrics Dependence

In many real-world situations, metrics have some dependence. Marichal (2000) suggests distinguishing three types of interaction between metrics:

Definition 1 (Types of Metrics Interactions) Metrics are *correlated* if high partial values along one metric imply high (low) partial values along another metric. More general, a

subset of metrics is *preferential dependent* on another, complementary subset if their values are related to each other. Metrics are *substitutable for/complementary to* each other if the satisfaction of only one metric produces almost the same effect as the satisfaction of both.

These types of interactions imply different possible metrics *dependencies*.

Correlation and preferential dependence could be obtained solely from data using statistical tests. Substitutivity/complementary can be interpreted as follows: A set of metrics can be substituted by a subset thereof if bad (good) performance in a subset of metrics produces the same effect as bad (good) performances in all of them. This could be obtained from expert opinions and might then be subjective. Alternatively, it requires a quantitative (ground truth) score of the effect and supervised learning. In our aggregation approach that is data-driven (no experts allowed) and unsupervised (no ground truth available), we focus on correlation and preferential dependence of metrics.

3.2 Interpretation of Metrics Weights

Metrics weights play an important role in measuring the overall performance in many MCDM models. However, there is no canonical interpretation of metrics weights. Choo et al. (1999) identify several possible interpretations and appropriate questions that could be posed to experts for setting appropriate weights:

A: Weights ratios as indifference trade-offs. How much improvement in one metric is acceptable as compensation for one unit of loss in the other, with all values for other metrics held constant?

B: Weights as the relative contribution to overall value. Which one is more important, the average worth of values for one metric or the other, and by how much?

C: Weights as the relative contribution of swing from worse to best. What is the ratio of the contribution to the overall quality of the swing from worst to best in one metric to the contribution of swing from worst to best in the other?

D: Weights as the information content carried in metrics. Of the two metrics being compared, which one is considered to have a higher contrast intensity?

E: Weights as the relative functional importance of metrics. Of the two metrics being compared, which one is more powerful for differentiating the overall desirability of the alternatives?

These interpretations are different possible definitions of the notion of metrics importance. There is no objectively correct way to weight metrics, i.e., any weighting approach is based on some goals and assumptions. We aim for a general, data-driven aggregating approach based on a set of metrics and their values. Therefore, we deliberately exclude interpretations (*A*–*C*), trade-offs, contributions to the overall value, and swinging from worse to best. These are subjective and can only be obtained from domain experts' opinions.

Instead, we focus on the interpretations (*D, E*) where weights can be obtained solely from data. Assuming that no metric is (known to be) more important than the others, metrics that do not distinguish the artifacts are weighed down (*D*). Also, metrics that are highly correlated with each other are down-weighted so that other metrics would have an equal representation in the aggregated quality score (*E*). For example, suppose we have metrics μ_1, μ_2, μ_3, and two of them are highly correlated and the third does not depend on the others, say, $cor(\mu_1, \mu_2) = 0.9$, and $cor(\mu_3, \mu_1) = cor(\mu_3, \mu_2) = 0$, then we expect that μ_3 should get almost half of the total weight while μ_1 and μ_2 should each get about one quarter of the total weight.

Moreover, it is a well-known fact in machine learning that variables with large value ranges dominate the outcome while those with smaller value ranges get marginalized. A metric with larger ranges could influence the aggregated score more due to its larger value, but this does not necessarily mean it is more important than other metrics. We will not use weights to compensate for the range of values for different metrics. Instead, all values are normalized. As for the weights, there is no objectively correct way to normalize metrics. For each metric μ, we choose the normalization to the [0, 1] interval such that normalized metrics values can be interpreted as probabilities. We normalize each metric μ such that higher normalized values either positively or negatively contribute to the quality score, consistently over all metrics, i.e., all normalized metrics will be positively or negatively correlated with the quality score.

In the next section, we discuss in detail how we normalize, weight, and aggregate the metrics.

4 The Weighted Quality Scoring Approach

We interpret software quality as a *latent variable* (Borsboom et al. 2003); it cannot be observed directly, but is inferred from (software) metrics that measure different aspects of quality of a software artifact, e.g., of a software product or its development process. Formally, we use a mathematical model:

$$Model(Quality) := (\mathcal{A}, \mathcal{M}, \mathbb{R}, \succeq)$$

$\mathcal{A}$ is a set of software artifacts and $\mathcal{M}$ is a vector of software metrics. Each metric is a function that maps $\mathcal{A}$ to a subset of $\mathbb{R}$, e.g., a subrange of the natural numbers $\mathbb{N}$ (counts) or the rational numbers $\mathbb{Q}$ (ratios). $\succeq$ is a non-strict, total order relation to compare software artifacts with respect to their quality.

For deriving such a model and applying it to software quality assessment, we follow four steps that are detailed in the subsections below. We refer to this aggregation method as *Weighted Quality Scoring (WQS)*:

1: Calculate metrics scores normalizing the metrics values so that resulting values are in the range [0, 1] and in the same direction (Section 4.1).
2: Calculate weights (*D*) that represent information content and weights (*E*) that represent relative contribution (Section 4.3).
3: Aggregate the metrics scores into a quality score using the joint distribution of their values (unweighted in Section 4.2 and weighted in Section 4.4).
4: Rank software artifacts based on their aggregated quality scores (Section 4.5).

We assume that the view on quality and the goal of quality assessment are given, as well as the relevant metrics, and the metrics values of (software) artifacts.

Given software metrics $\mu_1, \ldots, \mu_k$, we represent the assessment result for software artifacts $a_1, \ldots, a_n$ as an n by k *quality matrix* $[m_{ji}]$ of metrics values. We denote by $m_{ji}, j \in \{1, n\}, i \in \{1, k\}$, the quality of an artifact a_j assessed with metric μ_i. We use $\vec{m}_i = [m_{1i}, \ldots, m_{ni}]^T \in M_i^n$ to denote the i-th column of the quality matrix, which represents metrics values for all software artifacts with respect to metric μ_i, where M_i is the domain of its values.

We denote by σ_i a *metric score function*. It normalizes the metrics values. As the original metrics values m_{ji}, their corresponding normalized values $s_{ji} = \sigma_i(m_{ji})$ still indicate the

degree to which the software artifact $a_j \in \mathcal{A}$ performs in the metric μ_i. The metric and score functions have the following signatures.

$$\mu_i \ : \mathcal{A} \mapsto M_i$$
$$\sigma_i[\vec{m}_i] \ : \ M_i \mapsto [0, 1]$$

Since σ_i is calculated numerically based on metric data, it is parameterized with $\vec{m}_i$, i.e., with a vector of metrics values of metric μ_i observed for software artifacts. If the actual metrics values are not important or obvious, we skip this parameter. We will formally define σ_i when we discuss metrics normalization in Section 4.1.

A *weight vector* w_i represents the relative importance of metrics scores function σ_i compared to the others:

$$\vec{w} = [w_1, \ldots, w_k], \text{ where } \sum_{i=1}^{k} w_i = 1$$

We will formally define the calculation of weights w in Section 4.3.

Based on all metrics score functions $\sigma = \{\sigma_1, \ldots, \sigma_k\}$ and the weight vector $\vec{w}$, our goal is to define an *overall quality score function* such that the relation $\succeq$ on software artifacts and the $\geq$ on the quality scores of these artifacts are isomorphic. Such a function takes a k-tuple of metrics scores and returns a single overall quality score. This aggregation function has the following signature.

$$F[\vec{w}] : [0, 1]^k \mapsto [0, 1]$$

Note that F is parameterized with the vectors of weights corresponding to the metrics. For unweighted aggregation, the weight vector is not important and we skip it. We formally define possible functions F when we discuss unweighted aggregation in Section 4.2 and weighted aggregation in Section 4.4.

Before that, we note that F is an *aggregation operator* (Calvo et al. 2002), i.e., it should satisfy the following

Definition 2 *Aggregation Operator Properties*

Boundary condition. If software artifact a_j is the best (worst) according to all metrics, the overall quality score should be close to 1 (close to 0).

$$F[\vec{w}](s_{j1}, \ldots, s_{jk}) = 1 \text{ as } \forall_{i=1}^{k} s_{ji} = 1$$
$$F[\vec{w}](s_{j1}, \ldots, s_{jk}) = 0 \text{ as } \forall_{i=1}^{k} s_{ji} = 0$$

Averaging property An aggregation output should remain between the minimum and the maximum of the inputs.

$$min_{i=1}^{k}(s_{ji}) \leq F[\vec{w}](s_{j1}, \ldots, s_{jk}) \leq max_{i=1}^{k}(s_{ji})$$

Commutativity The overall quality score function should be commutative. Let π be an arbitrary permutation of k elements.

$$\forall \pi \ \ F[\vec{w}](s_{j1}, \ldots, s_{jk}) = F[w_{\pi(1)}, \ldots, w_{\pi(k)}](s_{\pi(1)}, \ldots, s_{\pi(k)})$$

Monotonicity If all metrics scores of a software artifact a_j are greater than or equal to all the corresponding metrics scores of another software artifact a_l, the same should be true for their overall quality scores.

$$F[\vec{w}](s_{j1}, \ldots, s_{jk}) \geq F[\vec{w}](s_{l1}, \ldots, s_{lk}) \text{ as } \forall_{i=1}^{k} s_{ji} \geq s_{li}$$

4.1 Normalization of Metrics Values to Metrics Scores

Software metrics can have different value ranges, scale types, and *directions*. The latter means that large metric values indicate either poor or good overall quality. If large metric values indicate higher satisfaction, we call the metric's direction *positive*, otherwise *negative*. To normalize these differences, we propose to express the performance of software artifact a_j according to the metric μ_i using probability. For metrics with positive (negative) direction, this normalized metrics score represents a probability of finding another software artifact with metrics value smaller or equal (greater or equal) to the given value. We calculate such a normalized metric score using empirical probabilities as follows.

$$\sigma_i[\vec{m}_i](m_{ji}) = s_{ji} = \frac{1}{n}\sum_{j'=1}^{n} \mathbb{1} \begin{cases} (m_{j'i} \leq m_{ji}), & \text{if direction is positive for } \mu_i \\ (m_{j'i} \geq m_{ji}), & \text{otherwise} \end{cases} \quad (1)$$

where $\mathbb{1}$ is the *indicator function* with: $\mathbb{1}(cond) = 1$ if *cond* and 0, otherwise.

The proposed metrics normalization is the so-called *probability integral transform*, which is widely used in multivariate data analysis to transform random variables in order to study their joint distribution (Nelsen 2007). In other words, we obtain the scores by the *cumulative distribution function* (CDF) when it is assumed that low values are bad, i.e., the metric's direction is positive, and by the *complementary cumulative distribution function* (CCDF), otherwise.

The normalized metric score interpretation is the same for all metrics: the probability of finding another software artifact that performs worse than the given one (0 indicates the worst, 1 the best performance).

Knowing its direction, we can normalize all observed values $\hat{m} \in \hat{M} \subseteq M$ from each metric $\mu : \mathcal{A} \rightarrow M$ empirically by computing the fraction of the values lesser (or greater) than or equal to m as implemented in Algorithm 1, which approximates the metric score function σ that corresponds to μ.

Algorithm 1 Normalize($\hat{M}$, *direction*).

```
for all m ∈ M̂ do
    count = 0
    for all m̂ ∈ M̂ do
        if direction is positive then
            //Normalizing by CDF
            if m̂ ≤ m then
                count ++
            end if
        else
            //Normalizing by CCDF
            if m̂ ≥ m then
                count ++
            end if
        end if
    end for
    σ(m)=count/|M̂|
end for
```

However, it is left to discuss how to derive the directions of the metrics. The knowledge if low values for a specific metric are desirable or not can be obtained by experts' opinions or from the research in the application domain. In software engineering, e.g., it is known that size and complexity (metrics) are negative for maintainability and error proneness, i.e., have a negative direction w.r.t. these overall qualities.

In fact, knowledge of the direction of a single metric μ_1 is enough to define directions for all other metrics: metrics positively correlated with μ_1 have the same direction and we normalize them in the same way as μ_1; negatively correlated metrics have the opposite direction and we normalize them in an opposite way. For example, if it fair to assume that low values of the lines of code (LOC) metric are desirable for error proneness, we set its direction to negative and normalize it by CCDF using Algorithm 1. We also set the direction to negative for all other metrics that are positively correlated with LOC, e.g., the weighted method count (WMC) metric. Therefore, we empirically calculate and test Spearman's coefficient of correlation $\rho(LOC, WMC) \geq 0$. Normalizing all metrics based on the knowledge of the direction of metric μ_1 is implemented empirically in Algorithm 2. Let $\hat{M}_i \subseteq M_i$ be the measured metric values for each metric $\mu_i : \mathcal{A} \rightarrow M_i$ and rho be Spearman's correlation coefficient of samples.

Algorithm 2 NormalizeAll($\hat{M}_1, \ldots, \hat{M}_k, direction_1$).

```
//For all k − 1 metrics excluding the first one.
for i ∈ 2...k do
    if ρ(M̂_1, M̂_i) ≥ 0 then
        σ_i = Normalize(M̂_i, direction_1)
    else
        σ_i = Normalize(M̂_i, opposite(direction_1))
    end if
end for
```

In the arguably unlikely case, direction knowledge is not even available for a single metric, there is still a way how to normalize the metrics purely based on data. Then we simply guess the direction of one metric μ_1 knowing that we might be wrong. As the result, for all metrics, the normalized scores point in the same direction, which, in turn, is either positively or negatively correlated with the overall quality score. This needs then to be tested empirically by applying the aggregated quality score in the application domain, which is outside the scope of this paper.

Finally, the normalization using (C)CDF deserves some explanations from an information-theoretic point of view. Because of their symmetry, we will focus on CDF and drop CCDF in this discussion. The cumulative distribution function CDF_X of a random variable X is the probability that X will take a value less than or equal to x, i.e., $CDF_X(x) = P(X \leq x)$. It requires that X is measured at least on an ordinal scale, i.e. "less than or equal" ($\leq$) is defined on X. This is guaranteed by our definition of metrics. For instance, LOC (a metric of a class) induces an order whereas the programmer (an attribute of a class, but not a metric) does not.

CDF_X is a monotonously increasing function mapping X to the interval [0, 1]. Without loss of generality, we can assume that each value $x \in X$ occurs with a probability larger than 0, i.e., the probability density $PDF_X(x) > 0$. Hence, CDF_X is even a *strictly* monotonously increasing function. As a consequence, it is invertible for any given probability distribution of X, i.e., CDF_X^{-1} exists and is a function $[0, 1] \mapsto X$. A normalized

metric score $s = CDF_X(x)$ and the distribution of X, uniquely define the value x. In short, $s = CDF_X(x) \Leftrightarrow x = CDF_X^{-1}(s)$. Hence, using the normalized metric score instead of the original metric value does not loose information.

However, the distribution of X is, in general, unknown and we do not want to make any assumptions about possible distribution laws of X. In general, it can only be approximated numerically by observing a (representative, sufficiently large) sample $\hat{X}$ of the population X. Then the empirical (or sample) cumulative distribution function $ECDF_X$ is a good approximation of CDF_X. $ECDF_X(x)$ can be calculated as the relative frequency of observations in the sample $\hat{X}$ that are less or equal x, i.e.,

$$ECDF_X(x) = \frac{|\{\hat{x}|\hat{x} \in \hat{X}, \hat{x} \leq x\}|}{|\hat{X}|},$$

with $|\cdot|$ the size of a set, which is exactly our normalization transformation as implemented in Algorithm 1. If the number of data points is relatively small or biased, one cannot claim to have a representative sample $\hat{X}$ of the population X. Consequently, $ECDF_X$ may not be a good approximation of CDF_X. Conversely, using the normalized metrics scores is as good as using the original values from an information-theoretic point of view if $ECDF_X$ is a good approximation of CDF_X, i.e., if $\hat{X}$ is a representative, sufficiently large sample of X.

4.2 Unweighted Aggregation

Now we are ready to define an *unweighted* aggregation function $F : [0, 1]^k \mapsto [0, 1]$. Let s_{ji} be the metric score of metrics value m_{ji} normalized by (1). This metrics value, in turn, is measured for a software artifact $a_j \in \mathcal{A}$ by a metrics μ_i. Corresponding to the *quality matrix* $[m_{ji}]$ we define a *quality score matrix* $[s_{ji}]$. All scores are in [0, 1], have the same interpretation and the same direction, which makes them comparable. Hence, it is possible to aggregate them. The joint distribution defines an *unweighted aggregated quality score* of an artifact a_j, that is an (empirical approximation of the) joint probability of an artifact performing worse than or equally good as a_j in *all* metrics $\mathcal{M}$.[1]

$$F(s_{j1}, \ldots, s_{jk}) = \frac{1}{n} \sum_{l=1}^{n} \mathbb{1}(s_{l1} \leq s_{j1} \wedge \cdots \wedge s_{lk} \leq s_{jk}) \tag{2}$$

From a theoretical point of view, the proposed quality scoring is nothing but the (empirical approximation of the) joint CDF of the scores, which in turn are (empirical approximations of) either CDFs or CCDFs of the metrics.

We refer to the process of obtaining unweighted aggregated scores as *Quality Scoring (QS).*

4.3 Weighting

Normalization has taken the value range out of the equation. To set objective weights according to the discussion in Section 3), we should consider both: the diversity within a metric (D) and the possible interaction between the different metrics (E). For metrics with a smaller variance, we should assign smaller weights. Also, knowing that a metric is

[1]Recall, in case we could not know the initial direction of a single metric and we guessed it wrongly, the interpretation would be: the joint probability of an artifact performing *better* than or equally good as the current one. We will not repeat this disclaimer from here on.

correlated with many others implies that it might have very little impact on the overall (latent, ground truth) quality because the impact of the fact that this metric assesses is distributed over the correlated metrics that somehow assess the same fact.

We suggest to use an *entropy method* to set objective weights based on the diversity within a metric (*D*). Entropy measures the information content carried by a metric (Zeleny 2012). It assigns a smaller weight to a metric if its scores are similar across the software system. Such a metric does not help in differentiating software artifacts.

The amount of information provided by a score of metric μ_i can be measured using *Shannon's entropy* H. It is commonly used in information theory to determine the average level of information inherent in a random variable's possible outcomes. We interpret entropy weights as a *degree of diversity* of software artifacts from $\mathcal{A}$ with respect to metric scores μ_i; 0 indicates no, 1 a high degree of information to distinguish software artifacts. Metrics with a larger entropy are considered to be more important. Hence, they should have a higher weight.

Formally, we define an entropy for each metric μ_i as:

$$H_i = -\sum_{j=1}^{n} p(s_{ji}) \log p(s_{ji}), \text{ where } n = |\mathcal{A}|$$

We estimate probabilities $p(s_{ji})$ with empirical frequencies. To make the weights for all metrics μ_i, $1 \leq i \leq k$ sum up to 1, we define the respective *entropy weight* as:

$$w_i^{ent} = \frac{H_i}{\sum_{l=1}^{k} H_l}, \text{ where } k = |\mathcal{M}|$$

Weights based on entropy do not consider possible interaction between metrics scores. We, therefore, define another vector of weights that reflects the interaction between the metrics (*E*) assessed by their scores' relative contribution to the joint probability distribution of the metric scores.

Let R_i be the ranking of artifacts $1 \leq j \leq n, n = |\mathcal{A}|$ according to their metrics scores s_{ji} (1) for any metric μ_i, $1 \leq i \leq k, k = |\mathcal{M}|$. Let R be the ranking of artifacts according to the unweighted aggregated quality score (2). We denote by $\rho_i = |\rho(R_i, R)|$ the *degree of dependency* of the metric scores s_{ji} and the (unweighted) aggregation of all scores $F(s_{j1}, \ldots, s_{jk})$. It is defined as the absolute value of *Spearman's* rank order correlation ρ of the rankings above. Metrics μ_i, μ_l inducing a high rank order correlation with the unweighted ranking are not independent and their importance should be reduced by lower weights. Metrics inducing a low rank order correlation show a higher degree of independence; they are more important. Hence, they should have higher weights. To make the weights for all metrics μ_i, $1 \leq i \leq k$ sum up to 1, we define the respective *dependency weight* as:

$$w_i^{dep} = \frac{1 - \rho_i}{k - \sum_{l=1}^{k} \rho_l}$$

To use both entropy and dependency weights, we integrate them as a normalized product:

$$w_i = \frac{w_i^{dep} \times w_i^{ent}}{\sum_{l=1}^{k} \left(w_l^{dep} \times w_l^{ent} \right)}$$

4.4 Weighted Aggregation

We use *Levi-frailty copula* (Mai and Scherer 2009) to model an aggregation as a weighted product function. It links marginals with their joint distribution, and marginal distributions

are modeled separately from the dependence structure. Given metrics scores $[s_{j1}, \dots, s_{jk}]$ and the corresponding weights $\vec{w} = [w_1, \dots, w_k]$, we define the weighted quality score aggregation as follows.

$$F[\vec{w}](s_{j1}, \dots, s_{jk}) = \prod_{i=1}^{k} s_{ji}^{w_i} \tag{3}$$

The result of this aggregation is a *weighted aggregated quality score*. We interpret it as a score of relative overall quality, i.e., how good a software artifact performs in a set of interacting metrics, relative to other artifacts; 0 indicates bad and 1 good quality. The proposed aggregation satisfies the required properties of aggregation operators (boundary condition, averaging, commutativity, and monotonicity). It was formally proved for the well-known weighted product model for multi-criteria decision making (Triantaphyllou 2000), which (3) actually is. Therefore, we omit the proof here. Note that the weighted product model has the advantage compared to the (also well-known) weighted sum model that it implies that poor metrics scores for some metrics cannot be compensated by sufficiently high quality scores for other metrics.

We refer to the process of obtaining weighted aggregated scores as *Weighted Quality Scoring (WQS).*

4.5 Ranking

Once the aggregated quality scores are computed, the software artifacts can be ranked by simply ordering the values. We assign the same rank for artifacts in case their aggregated scores are equal (up to margin of error $\varepsilon = [n]^{-1}, n = |\mathcal{A}|$, and $[n]$ rounds to the nearest power of 10).

A software artifact a_j to be better than or equally good as another artifact a_l, if the aggregated score of a_j is greater than or equal to the aggregated score of a_l. Software artifacts with bad quality should be prioritized and inspected first, therefore we rank software artifacts based on their aggregated scores in acceding order: software artifact with the lowest (highest) possible aggregated score will be ranked as first (last).

For better numerical stability, we calculate the ranking for *WQS*, based on

$$\log(F[\vec{w}](s_{j1}, \dots, s_{jk})) = \sum_{i=1}^{k} (w_i \log(s_{ji})).$$

Putting the loose strings together, let $m_{ji} = \mu_i(a_j), s_{ji} = \sigma_i(m_{ji})$.

$$a_j \succeq a_l \Leftrightarrow \begin{cases} F(s_{j1}, \dots, s_{jk}) \geq F(s_{l1}, \dots, s_{lk}) & \text{for } QS \\ \log(F[\vec{w}](s_{j1}, \dots, s_{jk})) \geq \log(F[w](s_{l1}, \dots, s_{lk})) & \text{for } WQS \end{cases}$$

5 Application to Software Defect Prediction

The objective of a defect prediction model is to determine risky code for further software quality assurance activities.

There are two possible model outcomes: classification, i.e., classify each entity as defective or clean, or ranking based on defect prediction, i.e., provide a list of software artifacts ordered by their risk to be defective. In this research, we predict a ranking of software artifacts by the number and the density of defects they will possibly exhibit, instead of predicting the actual number of defects nor whether a software artifact is buggy or not. In short,

we provide a ranking not a classification. This allows the developers/quality assurance (QA) engineers to focus on the most defect-prone (top-ranked) artifacts, given a limited amount of time for reviewing/QA.

The size of software artifacts is positively coupled to the defects it contains (Koru et al. 2008). However, smaller software artifacts tend to have a higher bug density, i.e., for an equivalent number of bugs to discover, inspecting much smaller software artifact involves less effort than inspecting a few larger artifacts (Arisholm et al. 2010). A corresponding prediction model for software artifacts aims at maximizing defect detection efficiency, i.e., the ratio of the number of bugs potentially detected per effort spent.

Such a prediction model can be utilized in two conceptually slightly different scenarios. They complement each other to improve the quality of an upcoming software release: (i) *short-term quality assurance* predicts the efficiency of detecting defects in software changes and (ii) *long-term quality assurance* predicts the efficiency of detecting defects in a particular software module.

In this paper, we do not aim to build the best-performing predictor. In contrast, to illustrate what our approach can bring to defect prediction, we study whether or not our approach could improve the performance of existing models for defect prediction. We only consider models suggested in the related work as they are and compare them to the approach proposed in Section 4 for ranking software artifacts as they are. We do not try to fine-tune and improve either of the models for the context of defect prediction.

The first study (i), aims at bug prediction in code changes. It ranks code changes based on defect-proneness. We apply the change metrics from the publicly available benchmark of Kamei et al. (2013). The ground truth is the ranking of code changes according to their post-release effort. We compare the results of our approach with other approaches that used the same benchmark (Yang et al. 2016; Liu et al. 2017; Fu and Menzies 2017).

The second study (ii) aims at bug prediction of software classes. It ranks software classes according to their defect proneness. We use software metrics and prediction models from the publicly available benchmark[2] by D'Ambros et al. (2010). The ground truth is the ranking of classes based on their observed post-release defects (D'Ambros et al. 2012).

We implemented all algorithms and statistical analyses in *R*.[3] We provide all *R* scripts that are used to conduct the experiments in a replication package downloadable from https://doi.org/10.5281/zenodo.4603073.

The benchmarks contain several open-source software systems. We do not merge changes/classes over all systems, since otherwise individual differences between the projects might be blurred (Easterbrook et al. 2008). To make the comparison fair, we use the same performance measures and prediction settings as used in related studies (D'Ambros et al. 2010; 2012; Yang et al. 2016; Liu et al. 2017; Fu and Menzies 2017).

5.1 Study (i) — Ranking Code Changes

We apply the *QS* and *WQS* approaches proposed in Section 4 and the approaches from the related work (Yang et al. 2016; Liu et al. 2017; Fu and Menzies 2017) to ranking software changes according to their expected defect detection efficiency, i.e., the number of bugs expected to be found per unit of time spent.

[2] http://bug.inf.usi.ch

[3] The R Project for Statistical Computing, https://www.r-project.org

Table 1 Software systems used in ranking code changes evaluation

System	#changes	#defect-inducing changes
bugzilla	4,620	1,696
columba	4,455	1,361
eclipseJDT	35 386	5,089
eclipse Platform	64,250	9,452
mozilla	98,275	5,149
postgreSQL	20,431	5,119

5.1.1 The Code Changes Benchmark

The benchmark consists of a data from six open source systems: *bugzilla*[4] written in Perl; *columba*[5], *eclipse JDT*[6], and *eclipse Platform*[7] written in Java; *mozilla*[8] written in C++, and *postgreSQL*[9] written in C. Table 1 shows statistics for these systems. The number of defect-inducing changes is the number of changes that induces one or more defects. The CVS periods in the footnotes are the time periods in which commits and bug reports were collected. Table 2 shows the change metrics that were use in the experiments.

5.1.2 Models

All unsupervised and supervised models compared in this study use the same change metrics (see Table 2). Below we briefly explain the models suggested in the related work. They aim at priority ranking the changes such that the expected defect detection efficiency is maximized when revising the changes in this order.

The ***Simple unsupervised model built from change metrics*** (Yang et al. 2016) calculates, for a single metric μ, a risk score as its reciprocal $R = \mu^{-1}$. It prioritizes changes with smaller metric values, hence, it ranks them based on risk score in decedent order. If there is a tie, the change with a larger code churn—the sum of lines of code added LA and deleted LD in a change—will be ranked later. Since code churn should be a tie-breaker, no simple unsupervised model is built for the corresponding metrics LA and LD. Yang et al. do not expect these simple unsupervised models to have a high predictive performance. However, the one built on the LT metric performs quite well in the benchmarks and was even competitive with the supervised models. For this reason, we too include this ***LT*** model in our comparison.

The ***Code churn-based unsupervised model*** (Liu et al. 2017) extends the idea of the simple unsupervised models but uses only the churn metrics to compute a risk score. For each change, the corresponding code churn is again the sum of LA and LD. Changes are ranked in descendant order according to this churn risk score $1/(LA + LD)$. In our comparison, we refer to this model as ***CCUM***.

[4]https://www.bugzilla.org, CVS period 08/1998-12/2006
[5]https://sourceforge.net/projects/columba, CVS period 11/2002-07/2006
[6]https://www.eclipse.org/jdt, CVS period 05/2001-12/2007
[7]https://projects.eclipse.org/projects/eclipse.platform, 05/2001-12/2007
[8]https://www.mozilla.org, CVS period 01/2000-12/2006
[9]https://www.postgresql.org, CVS period 07/1996-05/2010

Table 2 Metrics used in ranking code changes evaluation

Metric	Description
NS	Number of modified subsystems
ND	Number of modified directories
NF	Number of modified files
ENT	Distribution of modified code across each file
LA	Lines of code added
LD	Lines of code deleted
LT	Lines of code in file before the change
FIX	Whether or not the change is defect fix
NDEV	The number of developers that changed the modified files
AGE	The avg. time interval between the last and the current change
NUC	The number of unique changes to modified files
EXP	Developer experience
REXP	Recent developer experience
SEXP	Developer experience on a subsystem

The ***Predictor based on simple unsupervised models*** (Fu and Menzies 2017) is a supervised extension of the simple unsupervised models. Fu and Menzies argue that no simple unsupervised model is consistently working best on all software systems and use supervised learning to find the best simple unsupervised model specific for each software system. The simple unsupervised model performing best in four evaluation metrics P_{opt}, *Recall*, *Precision*, and *F1 score*, i.e., the highest mean over these four evaluation metrics, is chosen. The evaluation metrics are detailed in Section 5.2.1 below. Similar to Yang et al., changes are ranked in descendant order according to their predicted risk score computed as the reciprocal of different metrics for different software systems. In our comparison, we refer to this model as ***OneWay***.

We build the unsupervised ***WQS*** model (3) aggregating the change metrics (except LA and LD) and rank the aggregated score in ascending order. To investigate the importance of weights, we also build the unweighted variation ***QS*** (2).

To be consistent with Yang et al., Liu et al., and Fu and Menzies, we normalize the metric LT using CDF, since smaller files have a higher defect density. Then, directions of the other metrics are defined using Algorithm 2. Obviously, we use expert knowledge only for one metric LT while the other approaches use such expert knowledge for all metrics.

We compare the rankings induced by the simple unsupervised model *LT*, the unsupervised model *CCUM*, the supervised model *OneWay*, and the rankings of our unsupervised aggregation approaches *QS* and *WQS* to the ground truth.

Since the unsupervised approaches do not use the ground truth information to build the prediction models, they are not expected to perform better than the supervised models trained on this information. Also, the models *WQS*, *QS*, *LT*, and *OneWay* do not use the churn size metrics LA and LD. These predictor variables are strongly correlated with effort (Kamei et al. 2013), which by definition is strongly (inversely) correlated with efficiency, the response variable. Hence, the *CCUM* model that uses churn size has maybe an (unfair) advantage over the other models.

5.2 Experimental Design

Like Fu and Menzies (2017), we use time-wise cross-validation to avoid testing on data used for training the models. This cross-validation technique even ensures that changes used in testing are always created later than the changes used in training. Therefore, it sorts all changes of each software system based on their commit date and groups all changes submitted in the same month. A sliding window of six months is moved over all n months with commit information for a software project and defines training and test folds. It is moved month by month from month $i = 1$ to $n - 5$ and the months $i, i + 1$ constitute the training fold, the months $i + 4, i + 5$ the test fold.

Because of different values of n and time-wise cross-validation, *bugzilla* has 101 folds, *columba* 44, *eclipse JDT* 80, *eclipse Platform* 80, *mozilla* 84, and *postgreSQL* 167. All unsupervised models only use the test folds to build the models and to evaluate their performance.

We report the 25 percentile, the median, and the 75 percentile values of the distributions of P_{opt}, *Precision*, *Recall*, and *F1 score* for each software system over the corresponding of folds.

To be consistent with Yang et al. (2016), Liu et al. (2017), and Fu and Menzies (2017) in statistically comparing the difference between our and other approaches, we use the Wilcoxon single ranked test to compare the performance scores of the approaches, and the Benjamini-Hochberg adjusted p-value to test whether observations are statistically significant at the 0.05 level. To measure the effect size of the performance scores of the approaches, we compute Cliff's δ, a non-parametric effect size measure. We assess the magnitude of the effect size as trivial if $|\delta| \leq 0.147$.

5.2.1 Evaluation Measures

We compare the rankings of the approaches to the ranking obtained from the ground truth that reviles the changes introducing defects and accounts for the effort required to inspect those changes. To be consistent with the related studies (Yang et al. 2016; Fu and Menzies 2017; Liu et al. 2017), we compare the approaches using the evaluation metrics P_{opt}, *Recall*, *Precision*, and *F1 score* suggested therein and under the same two simplifications of the ground truth.

The first simplification assumes that the code churn of a change is a proxy of the effort required to inspect the change, which is consistent with Kamei et al. (2013). The second simplification assumes that 80% of the defects are contained in 20% of the changed code (Ostrand et al. 2005). Hence, the evaluation metrics are assessed for the rankings up to the 20% effort point, i.e., only consider the top-ranked changes whose churn sums up to 20% of the total churn.

Effort-Aware Prediction Performance, P_{opt} was first suggested by Mende and Koschke (2009). The code churn (LA+LD) is a proxy of the effort to scan a change (x axis) and a binary indicator of whether this code change is buggy or not for defectiveness (y axis: 1 if the code change included a defect and 0, otherwise). We rank all changes descendingly by y and break the ties by x in ascending order. The result is a list of changes starting with the defective ones (smallest to largest) followed by the correct ones (smallest to largest). A model performs better than a random predictor only if $P_{opt} > 0.5$.

Precision, Recall, and F1 score. A model suggests a change as defective iff it is ranked before the 20% effort point. Let tp be the number of defective changes that a model predicted as defective, fp be the number of not defective changes that a model predicted

as defective, and fn be the number of defective changes that are not predicted defective. Define *Precision P and Recall R* as

$$P = \frac{tp}{tp + fp} \text{ and } R = \frac{tp}{tp + fn}, \text{ resp.}$$

and the *F1 score* as the harmonic mean of P and R. A low *Precision* indicates that developers would encounter many false alarms, which may have a negative impact on the developers' confidence in the prediction model. A low *Recall* indicates a defensive prediction that suggests only a few defective code changes and misses many others. A low *F1 score* indicates that a good *Precision* (*Recall*) is achieved at the expense of a low *Recall* (*Precision*).

5.2.2 Summary of Results

The boxplots (with 25 percentile, median, and 75 percentile values) in Fig. 1 visualize the different models and the cross-validation distribution of their performance in the evaluation measures P_{opt}, *Precision*, *Recall*, and *F1 scores*. Each row corresponds to the software system: *bugzilla*, *columba*, *eclipseJDT*, *mozilla*, *eclipse Platform*, and *postgreSQL* (in this order). Each column corresponds to the approaches *LT*, *CCUM*, *OneWay*, *QS*, and *WQS* (in this order).

A dashed line visualizes the median value of *WQS* and shows the differences between our approach and the others. We use colors to indicate a significant difference between the approaches. Blue (red) color indicates that an approach is significantly better (worse) than WQS for a benchmark according to the Wilcoxon signed-rank test and the Benjamini-Hochberg p-values is less than 0.05, and that the magnitude of the difference is not trivial. Black color indicates that an approach is not significantly different from WQS for a benchmark or the magnitude of the difference is trivial.

Figure 1 shows the performance comparisons in terms of *Popt*, *Precision*, *Recall*, and *F1 score*. Tables 3, 4, 5, and 6 show the median values and highlight the respectively best with a dark gray background, and the second best with a light gray background. Our *WQS* model often performs better than or equally good as the models build on a subset of metrics or a single metric in isolation. Only in 16 out of 96 comparisons in total (the blue ones) another approach is significantly better than *WQS*.

Effort-Aware Prediction Performance, P_{opt}: Figure 1a shows that all approaches but *CCUM* perform statistically worse than or equally good as*WQS*. It is no surprise that *CCUM* performs statistically better since it is based on churn that is used in the predictor and the response. Table 3 shows the median values of effort-aware evaluation results for each software system. We observe that *QS* never performs best or second best. The *WQS* model performs second-best in 5 out of 6 systems, and once even significantly better than all others. *WQS* performs better than *LT* and *OneWay* in 4 out of 6 systems, and equally good in the remaining systems. We conclude that considering weights is important for quality scoring in the effort-aware evaluation and that aggregating all metrics often improves the results even of single metrics models, even when the metrics are carefully chosen or trained with supervised learning.

Precision: Figure 1b shows that all approaches perform statistically worse or sometimes equally good compare to *WQS*, except for two cases: *OneWay* performs better for *mozilla* and *eclipseJDT*. It is not a surprise that *OneWay* could perform statistically better since it is a supervised model. Table 4 shows the median *Precision* results for each software system. *QS* performs second best and *WQS* best in half of the systems. We conclude that weights are

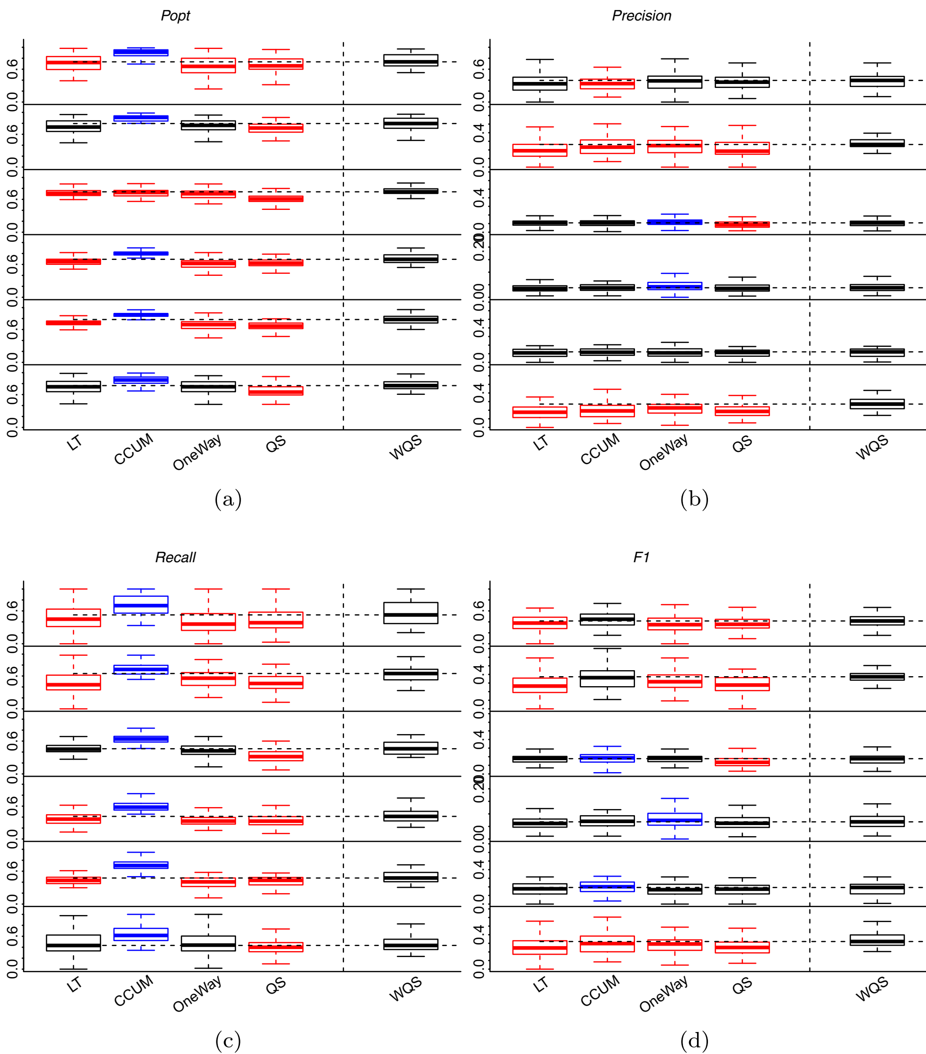

Fig. 1 Ranking Code Changes. Performance comparisons in terms of P_{opt}, *Precision*, *Recall*, and *F1 score*, between the proposed WQS and models from a benchmark over 6 software systems (from top to bottom are bugzilla, columba, eclipseJDT, mozilla, eclipse Platform, postgreSQL)

important for *Precision* and that aggregating all metrics often improves the *Precision* over single metric models.

Generally the *Precision* is low between 0.395 down to 0.038. However, on average *WQS* performs better than all other predictors in the contest and much better than a random predictor (randomly predicting defect-inducing changes). On average, *WQS* improves this random predictor by 80%. For details of the random predictor, we refer to Kamei et al. (2013).

Recall: Figure 1c shows that all but *CCUM* perform statistically worse or sometimes equally good as *WQS*. It is again no surprise that *CCUM* performs statistically better since it is based on churn used in prediction and evaluation. Table 5 shows the median *Recall*

Table 3 Ranking code changes. Effort-aware evaluation by P_{opt}

	LT	CCUM	OneWay	WQS	QS
bugzilla	0.721	0.909	0.652	0.734	0.661
columba	0.732	0.907	0.763	0.796	0.714
eclipseJDT	0.709	0.725	0.702	0.737	0.608
mozilla	0.651	0.802	0.617	0.693	0.625
eclipse Platform	0.717	0.868	0.689	0.763	0.663
postgreSQL	0.742	0.866	0.742	0.707	0.645

results. We observe that *QS* performs worst and *WQS* second best in all systems. Both *LT* and *OneWay* perform worse than *WQS* but in *jdt* and *mozilla* where they perform equally well. We conclude that considering weights is important for *Recall* and that aggregating all metrics can improve the results over single metric models.

F1 score: Figure 1d shows that *WQS* is almost as good as *CCUM*. Only in 3 out of 24 comparisons it is not the best-performing model. *CCUM* performs better in *eclipseJDT* and *eclipsePlatform*, *OneWay* in *mozilla*. As discussed before, no surprise that *CCUM* and *OneWay* might perform statistically better. Table 6 shows the median of *F1 score* results for each software system. *QS* performs second best in 2 of 6 and *WQS* best in 3 of 6 software systems, and second best in the remaining 3 systems. We conclude that weights are important for the *F1 score* and that aggregating all metrics often improves the results over single metric models.

5.3 Study (ii) — Ranking Software Classes

We apply the *QS* and *WQS* approaches proposed in Section 4 and approaches from the related work to rank software classes according to their expected number of bugs. Similar to the previous study (i), we also evaluate the expected defect detection efficiency, i.e., the number of bugs expected to be found per unit of time spent.

5.3.1 The Bug Prediction Benchmark

D'Ambros et al. (2010) published a set of metrics and defect information for several software systems. They also studied the performance of supervised prediction models, i.e., generalized linear regression build on different sets of metrics. The benchmark consists of

Table 4 Ranking code changes. Evaluation by *Precision*

	LT	CCUM	OneWay	WQS	QS
bugzilla	0.333	0.238	0.387	0.395	0.364
columba	0.19	0.179	0.25	0.263	0.185
eclipse JDT	0.112	0.077	0.117	0.108	0.087
mozilla	0.035	0.038	0.042	0.038	0.035
eclipse Platform	0.11	0.074	0.11	0.122	0.114
postgreSQL	0.176	0.111	0.227	0.271	0.186

Table 5 Ranking Code Changes. Evaluation by *Recall*

	LT	CCUM	OneWay	WQS	QS
bugzilla	0.449	0.697	0.359	0.527	0.384
columba	0.44	0.72	0.561	0.645	0.463
eclipse JDT	0.452	0.637	0.422	0.46	0.316
mozilla	0.363	0.582	0.327	0.413	0.326
eclipse Platform	0.432	0.704	0.406	0.476	0.434
postgreSQL	0.432	0.615	0.439	0.433	0.4

a data for five open-source software systems written in Java: *eclipse*[10], *equinox*[11], *lucene*[12], *mylyn*[13] and *pde*[14]. Table 7 shows statistics for these systems. The number of buggy classes is the number of classes with defects in the prediction release, i.e., the version the of software system for which the prediction is made. The total number of bugs is the number of non-trivial defects reported within six months after the release.

The benchmark contains metrics values for each class of each software system for 6 Chidamber & Kemerer (CK) metrics and 11 Object-Oriented (OO) metrics. Table 8 summarizes these source code metrics that we use in the comparison.The same CK and OO metrics are also used in the aggregation approaches; we refer to the as the *source code metrics* CK+OO.

5.3.2 Models

In addition to CK+OO collected for a single version, the benchmark also contains metrics calculated based on several versions of software systems. The *Churn of source code metrics* is the change of the metrics over time. It is calculated on consecutive versions of a software system sampled every two weeks. For each metric, a simple churn could be computed as a sum over all classes of the deltas of source code metrics values for each consecutive pair of versions. D'Ambros et al. (2010) suggest several variants, e.g., a weighted churn model that weights the frequency of change, i.e., if delta > 0, more than the actual change (delta) value, but also others that model a decay of churn over the version history.

The *Entropy of source code metrics* is a measure of the distribution of the churn over the classes of a system. For example, if the WMC churn is 100 in a system but, only one class changed, the entropy is minimum; if 10 classes are changed each contributing with a WMC of 10 to the churn then the entropy is higher. The entropy is calculated based on the deltas of source code metrics values computed for the churn. Again, D'Ambros et al. (2010) suggest several variants, e.g., modelling a decay of entropy over the version history.

Details about the CK+OO metrics, their churn and entropy, and the variants thereof can be found in the original paper by D'Ambros et al. (2010). They found out that the model variant based on weighted churn achieved the best performance among of models build from the churn of CK+OO, and that the model based on linearly decayed entropy achieved the best performance among of models build from entropy of these metrics. Therefore, in

[10]Eclipse JDT Core, https://eclipse.org/jdt/core/, prediction release 3.4.
[11]Equinox framework, https://eclipse.org/equinox/, prediction release 3.4
[12]Apache Lucene, https://lucene.apache.org/, prediction release 2.4.0
[13]Mylyn, https://eclipse.org/mylyn/, prediction release 3.1
[14]Eclipse PDE UI, https://eclipse.org/pde/pde-ui/, prediction release 3.4.1

Table 6 Ranking Code Changes. Evaluation by *F1 score*

	LT	CCUM	OneWay	*WQS*	*JQS*
bugzilla	0.378	0.448	0.349	0.416	0.356
columba	0.265	0.362	0.315	0.373	0.276
eclipse JDT	0.181	0.186	0.183	0.177	0.135
mozilla	0.062	0.071	0.074	0.068	0.063
eclipse Platform	0.178	0.201	0.167	0.193	0.174
postgreSQL	0.246	0.144	0.292	0.321	0.252

addition to the CK+OO metrics, we use these two best performing supervised models in the comparative study (ii). Details follow below.

To be consistent with D'Ambros et al. (2010), we use generalized linear regression[15] as a framework for the supervised models: given a set of metrics and their values, it computes their set of principal components to avoid a possible multicollinearity among the metrics. It builds the model only on the first k principle components that cumulatively account for (at least) 95% of the variance. D'Ambros et al. (2010) give no further details about the prediction models themselves, except for using generalized linear models. We thus used the first k principal components to build a generalized regression model using the *R* function *glm* with default parameters [16]

As mentioned, we include the following three supervised models in the comparison with the *(W)QS* models.

The ***Model built from source code metrics*** Basili et al. (1996) uses a single version of a software system to compute the source code metrics. This model fits the three first principal components of CK+OO using generalized linear regression to predict the number of defects and the efficiency of detection of post-release defects for each class, respectively. In our comparison, we refer to this model as the ***CKOO*** model.

The ***Model built from churn of source code metrics*** D'Ambros et al. (2010) uses several versions of a software system to sample the history of CK+OO every two weeks to compute the corresponding weighted churn metrics. This model fits the three first principal components of these churn metrics using generalized linear regression to predict the number of defects and the efficiency of defect detection, resp. In our experiment, we refer to this model as the ***WCHU*** model.

The ***Model built from entropy of source code metrics*** D'Ambros et al. (2010) uses several versions of a software system to sample the history of CK+OO every two weeks to compute the corresponding linearly decayed entropy metrics. This model fits the three first principal components of these entropy metrics using generalized linear regression to predict the number of defects and efficiency of defect detection. In our experiment, we refer to this model as the ***LDHH*** model.

In contrast to these *supervised* models, we use the CK+OO metrics to build ***WQS*** (3) and ***QS*** (2) models in an *unsupervised* way, i.e., without using a training data set with a ground truth of defects. We normalize the metric LOC by CDF. The reason is the same as before, larger software classes contribute with more bugs (Gyimothy et al. 2005), but smaller classes

[15]It is a generalization of ordinary linear regression that allows for predicted variables to have arbitrary distributions.

[16]https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/glm.

Table 7 Software systems used in ranking classes evaluation

System	#classes	#buggy classes	#bugs
eclipse	997	206	374
equinox	324	129	244
lucene	691	64	97
mylyn	1 862	245	340
pde	1 497	209	341

have a higher bug density (D'Ambros et al. 2010; 2012). This is the only expert knowledge we use in our model. Then directions of other metrics are defined according to Algorithm 2. Note that we use the ascending *(W)QS* ranking for predicting the efficiency of defect detection and the descending *(W)QS* ranking for predicting the number of defects per class.

We rank software classes based on their (un-)weighed aggregated scores and compare the ranking to the rankings of the regression approaches. There is no detailed information in D'Ambros et al. (2010, 2012) how to handle tied ranks. Therefore, we use the same strategy as in the first study, i.e., in case two classes have the same predicted values, the smaller class will be ranked earlier.

Note that *CKOO* and our models *WQS* and *QS* only require CK+OO from one version of the respective software system, while *WCHU* and *LDHH* require at least two versions.

Since unsupervised models *QS* and *WQS* do not use the ground truth (number of post-release defects) to build the prediction models, they are not expected to perform better than the supervised models *CKOO*, *WCHU*, and *LDHH*.

For the study (ii), we only compare with the respectively best metrics, churn, and entropy based contenders suggested by D'Ambros et al. (2010) who's benchmark we adopted. It

Table 8 Source code metrics used in ranking software classes evaluation

Type	Metric	Description
CK	WMC	Weighted Method Count
	DIT	Depth of Inheritance Tree
	RFC	Response For Class
	NOC	Number Of Children
	CBO	Coupling Between Objects
	LCOM	Lack of Cohesion in Methods
OO	FanIn	Number of other classes that reference the class
	FanOut	Number of other classes referenced by the class
	NOA	Number of attributes
	NOPA	Number of public attributes
	NOPRA	Number of private attributes
	NOAI	Number of attributes inherited
	LOC	Number of lines of code
	NOM	Number of methods
	NOPM	Number of public methods
	NOPRM	Number of private methods
	NOMI	Number of methods inherited

makes sense to compare our approaches to regression models as they too can be easily interpreted as aggregation operators, i.e., as a weighted sum. Yet again, we do not aim to build the best possible (supervised) prediction model and, therefore, spare the effort of implementing and comparing with other such models based on, e.g., random forests. Instead, we aim to study the performance of unsupervised metrics aggregation that needs to be used when a ground truth is not available.

5.3.3 Evaluation Measures

To be consistent with the related studies (D'Ambros et al. 2010, 2012), we compare the approaches using the evaluation metrics suggested therein:

Predictive Power, wCorr. The models predicting defects of classes output a list of classes, ranked by their predicted number of defects. D'Ambros et al. (2010) use correlation to evaluate the predictive power of these rankings. Classes with high numbers of bugs that appear at the wrong ranks are more critical than classes with low numbers of bugs appearing at the wrong ranks. We, therefore, measure the *weighted Spearman's rho correlation* between the predicted and the ground truth rankings to assess the ordering, relative spacing, and possible functional dependency.

There is no general interpretation of correlation. But, the so-called *Correlation Coefficient Rule of Thumb* (Newbold et al. 2013) suggests a lower bound of *Pearson's* $r \geq 2n^{-1/2}$ (n the sample size) indicating a linear relationship. For the smallest system *equinox* with 324 classes this would lead to $r \geq 0.11$, for the others even below that. We choose to be more conservative than this suggestion and set a lower bound of *weighted Spearman's* $|\rho| \leq 0.3$ indicating a negligible correlation in our experiments. We compute the weighted Spearman's correlation of two rankings, respectively, the classes ordered by an actual number of bugs and classes ordered by the prediction model, i.e., the aggregated quality score or the number of predicted bugs for the supervised approaches, respectively.

Effort-Aware Prediction Performance, P_{opt}. As before, we order the software artifacts (here classes) by their *bug density* to find an effort-aware ranking. Consistent with D'Ambros et al. (2012) we use LOC as a proxy of effort involved in inspecting a class (x axis) and the number of bugs (y axis). We rank all changes descendingly by y and break the ties by x in ascending order. A model performs better than a random predictor only if $P_{opt} > 0.5$. We measure P_{opt} at the 20% effort point, i.e., developers inspect the top ranked classes until 20% of the total lines of code is inspected.

5.3.4 Experimental Design

As D'Ambros et al. (2010) we perform 50 times 10-fold cross-validation, i.e., we split the dataset into 10 folds, using 9 folds as a training set to build the prediction model and the remaining fold as a validation set. In detail, we repeat each experiment 50 times within a software system. In each cross-validation, we randomly divide data from each system into 10 sub-samples of approximately equal size, each sub-sample is used once as the testing data and the remaining data is used for training. In this way, we obtain 500 values of *wCorr* and P_{opt}, resp., in each system for each supervised model. For our unsupervised models, we also apply the 50 times 10-fold cross-validation setting and use the same test data (ignoring the training data) each time to make a fair comparison.

To statistically compare the difference between ours and the other approaches in terms of predictive power *wCorr*, we use Steiger's test of significance for correlations at the 0.05

level. We compute the z-score to measure the effect size of *wCorr* and values $|z| \geq 1.96$ are considered significant (Steiger 1980).

To statistically compare the difference between ours and the other approaches in terms of P_{opt}, we use Wilcoxon's single ranked test to compare the performance scores of the approaches, and Benjamini-Hochberg's adjusted p-value to test whether two distributions are statistically significant. This is consistent with study (i) on the rankings of code changes. To measure the effect size of P_{opt} among our and other approaches, we compute Cliff's δ, we evaluate the magnitude of the effect size as before.

5.3.5 Summary of Results

Table 9 compares the predictive power in terms of weighted correlation between the supervised and unsupervised models, resp., and the ground truth, i.e., classes ranked by number of bugs, on each benchmark software system. As in study (i), the gray shades represent if the top performing approaches ignoring insignificant differences. Figure 2 and Table 10 compare the model's performance in terms of *Popt*. Similar to the first study, we use boxplots to visualize how *Popt* is distributed, and colors to represent if the differences are statistically significant. We observe that our approach *WQS* often performs better than or equally good as the supervised approaches. Details follow below.

wCorr: Table 9 shows that *WCHU* and *LDHH* perform statistically better than *CKOO* and our *WQS* and *QS* approaches. This does not come at any surprise, since the winning models use the CK+OO metrics from several versions of the system to compute churn and entropy of metrics. We observe that the *QS* model performs second-best only on 2 of 5 software systems whereas the *WQS* model performs second-best on 3 of 5 software systems, and best on 1 software system. However, it performs better than the supervised *CKOO* model on 3 of 5 software systems. We conclude that considering weights is important for the predictive power and that aggregating metrics by *WQS* often improves the results for the models build on metrics from a single version of the software system.

P_{opt}: Figure 2 shows that all approaches perform statistically worse, sometimes equally good compare to our WQS approach, with *LDHH* and *WCHU* performing better in *lucene* as the only exception. No surprise that *WCHU* and *LDHH* might perform better, since they are supervised models and use CK+OO from several versions of the system.

Table 10 confirms that our *WQS* model often performs better than or equally good as the supervised models. Only in 2 of 20 comparisons in total another approach is significantly better than *WQS*. We observe that the *QS* model never performs second best, but the *WQS* model does in 1 of 5 systems. It performs significantly better than all the other models in 2 of 5 systems. The *WQS* model performs better than *CKOO* in 4 of 5 systems, and equally good in the remaining system. *WQS* model performs significantly better than *WCHU* and

Table 9 Ranking software classes. Evaluation of predictive power by *wCorr*

	CKOO	WCHU	LDHH	WQS	QS
eclipse	0.6279658	0.6007369	0.6976652	0.6297236	0.5297825
equinox	0.7531017	0.7516816	0.7415858	0.7509734	0.5532562
lucene	0.3888639	0.5465674	0.5795628	0.4342752	0.4250039
mylyn	0.4779209	0.4563062	0.4568767	0.4391833	0.2871854
pde	0.2628426	0.6381531	0.705627	0.5266991	0.5261085

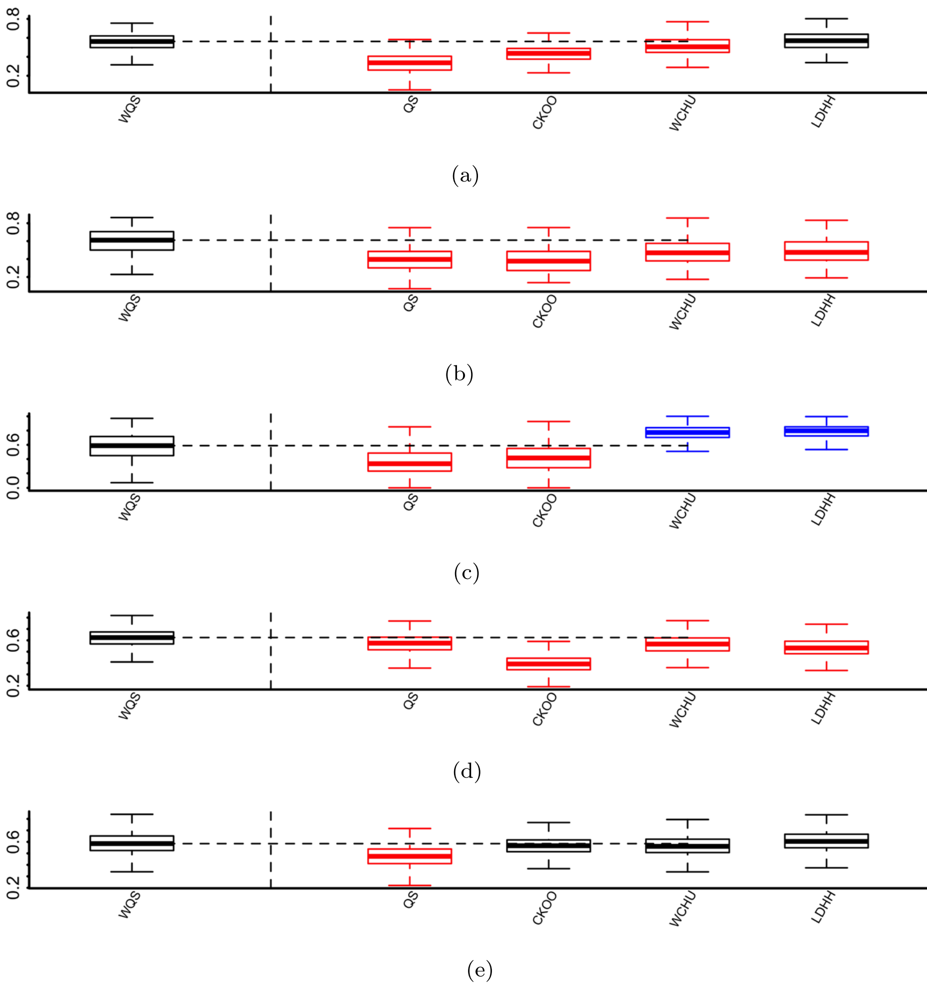

Fig. 2 Ranking software classes. Comparisons by P_{opt} between the proposed WQS and models from a benchmark over 5 software systems

LDHH models in 3 of 5 and in 2 of 5 systems, respectively. We conclude that considering weights is important for the effort-aware evaluation and that aggregating metrics by *WQS* often improves the results for the models build on metrics from a single version of the software system.

5.4 Discussion

5.4.1 Study (i) — Defect prediction of changes

The weighted *WQS* aggregation applied to code changes performs better than the unweighted *QS* approach in the benchmark proposed by Kamei et al. (2013) in terms of P_{opt} and *Recall*; it often performs better and never worse in terms of *Precision* and *F1 score*. *QS* only captures the relevant metrics, while *WQS* also captures dependencies, i.e., the relative

Table 10 Ranking software classes. Effort-aware evaluation by P_{opt}

	CKOO	WCHU	LDHH	WQS	QS
eclipse	0.436	0.506	0.57	0.562	0.337
equinox	0.377	0.469	0.475	0.611	0.396
lucene	0.416	0.773	0.798	0.588	0.335
mylyn	0.392	0.568	0.532	0.624	0.575
pde	0.566	0.561	0.604	0.585	0.474

functional importance of a metric, and degree of diversity, i.e., the information content carried in a metrics. We conclude that considering both the diversity within a single metric and possible interactions between several metrics are important for defect detection efficiency prediction on a code change level.

We observe that *WQS* performs often better than the unsupervised model *LT* (Yang et al. 2016) and the supervised model *OneWay* (Fu and Menzies 2017) in terms of all four evaluation measures, acknowledging that *OneWay* performs better in terms of *Precision* and *F1 score* on 2 of 6 software systems from the benchmark. We conclude that our unsupervised model can improve the performance of the simple (un-)supervised models based on code change metrics, i.e., applying our approach for aggregation of the whole set of metrics performs better than models built on a single well-chosen (manually or by supervised learning) metric.

We acknowledge that the unsupervised model *CCUM* performs better than our in terms of P_{opt} for 5 of 6 cases, and *Recall* in all the cases. However, we observe that our model performs equally good as or better than *CCUM* in terms of *Precision* for all 6 cases, and in terms of F_1 for 4 out of 6 cases. Note our doubts regarding the methodology applied in *CCUM* using churn in prediction and response variables.

We observe quite a low precision (Table 4) and a relatively high recall (Table 5): on average 0.2 for precision and 0.49 for recall. This is acceptable for highly skewed data sets as the change data of all software systems in the benchmark and was observed before by other researches. For instance, Menzies et al. (2006) also observed a low precision of defect prediction models at the file level, 0.14 on average.

Also, in the benchmark data set, the average total number of changes and the number of defect-inducing changes per day is 9.35 and 1.87 respectively, cf. Kamei et al. (2013). We interpret the results for precision and recall as follows. The *WQS* prediction model indicates, on average 4.58 changes per day as defect inducing ($tp + fp$). Among them, 0.92 changes are truly defect-inducing changes (tp) and developers need to check only 3.66 changes per day without finding any defect (fp). Hence, the prediction model could be useful in practical settings, since for the industrial needs, it is more important to avoid critical mistakes than spending more time for revision, i.e., a high recall is more important than high precision. It is still a statistically relevant tool, especially, when no knowledge (random choice) is the alternative.

5.4.2 Study (ii) — Defect prediction of classes

We found that the weighted *WQS* applied to the ranking software classes consistently performs better than or equally good as the unweighted *QS* in the benchmark proposed by D'Ambros et al. (2010) in predictive power *wCorr* and always better in effort-aware

predictive power P_{opt}. We conclude that both the diversity within a single metric and possible interactions between several metrics are also important for defect prediction on a class level.

The unsupervised *WQS* model performs better than the supervised *CKOO* model on 2, equally good on 2, and worse on only 1 of 5 software systems in terms of *wCorr*. It performs better than *CKOO* on 4 of 5 systems and equally good on the remaining system in P_{opt}. We conclude that aggregation of CK+OO metrics with the unsupervised *WQS* approach performs is (often) better than the supervised *CKOO* approach of generalized linear regression (in the benchmark).

The supervised models *WCHU* and *LDHH* perform better than *WQS* in 2 and 3 out of 5 cases, respectively, in *wCorr* but, better only in only 1 case in P_{opt}. In P_{opt}, *WQS* outperforms *WCHU* and *LDHH* in 3 and 2 cases, respectively. Note that both *WCHU* and *LDHH* are not only supervised, they also use CK+OO from several versions of the system to compute churn and entropy of metrics. Our unsupervised model uses only a single version to make predictions.

We conclude that our *unsupervised WQS* model can improve or is at least on par with the performance of the *supervised* models based on generalized linear regression, but gets along with unlabeled data and does not face a cold start problem.

5.5 Threats to Validity

Our experimental findings are generalizable, and we have high confidence that other researchers, given the same experimental setup, would come to the same conclusions. We discuss possible threats to validity below.

5.5.1 Construct validity

The central constructs are metrics. Their interpretation in the context of and their contribution direction to a quality goal do not need to be known in advance. However, for some of the software metrics too small or too large values or even values neither large nor small could be bad for quality. Once we know that, and often we do from the metrics design, this is not a limitation since we could transform metrics to have the property we assume. Otherwise, it is a threat to construct validity if these metrics dominate a quality model.

The quality goals, the contributing metrics, and the direction of one metric (if possible) are to be defined by experts, hence, subjective. This gives space for different interpretations which, in turn, is a threat to construct validity.

5.5.2 Internal validity

The proposed approach is mathematically specified; its implementation fully automated. The specification allows only minor variations, e.g., using different implementation programming languages or libraries for statistical analysis.

5.5.3 External validity

We compare our approach with defect prediction approaches used the (Kamei et al. 2013) benchmark and approaches reported in D'Ambros et al. (2010). This might be a treat to external validity since we do not compare it with other approaches. In this evaluation, we

consider 14 change metrics and 6 software systems for ranking code changes and 17 source code metrics and 5 software systems for ranking software classes. This specific sample might be a threat to external validity, as well.

We validated our approach on only two defect prediction quality models: in study (i) on rankings of code changes based on risk scores, and in study (ii) on rankings of classes based on their number of bugs and bug density. Validating our general unsupervised aggregation approach in more quality models in software engineering and other application areas would reduce this threat to external validity, and it is a matter of future work.

6 Conclusion and Future Work

We defined a weighted (software) metrics aggregation approach for (software) quality assessment and applied it in bug prediction scenarios. We defined probabilistic scores based on software metrics distributions and weights as a degree of disorder and dependency. We aggregated scores using the weighted product of normalized metrics. As such, the aggregation is agnostic to the actual metrics and the quality assessment goal and could be theoretically applied in any quality scoring and ranking context where multi-modal quality or ranking criteria need to be condensed to a single score. Its benefit is that models can be constructed using unsupervised learning when training data with a ground truth is lacking.

To evaluate the effectiveness of our approach and to illustrate its generalizability, we conducted two empirical studies in bug prediction. We evaluated a benchmark for just-in-time effort-aware bug prediction on a code change level (Kamei et al. 2013), and a publicly available benchmark for bug prediction on a class level (D'Ambros et al. 2010). In the first study we compare our approach with other unsupervised models by Yang et al. (2016) and by Liu et al. (2017), and a supervised model by Fu and Menzies (2017). In second study we compare our approach with supervised models by D'Ambros et al. (2010).

In the benchmark proposed by D'Ambros et al. (2010), models based on churn and entropy of source code metrics were reported as the best performing. These approaches are based on a system's history, i.e. require several versions of a software system. As this history does not exist for newly developed systems, these approaches have a cold start problem. In contrast, our approach only uses the values of the same metrics set of a single version of the software system and still performs well enough compared to churn- and entropy-based, supervised approaches and avoids the cold start problem.

While it turns out that our aggregation approach is quite competitive in this contest, it is neither our claim nor our ambition to suggest the best possible bug prediction approach. It only showed that the quality-goal-agnostic, unsupervised aggregation approach is a suitable alternative (even in bug prediction) when ground truth data is lacking. Inference of the connection between software artifact metrics and their proneness to defect (density) is outside the scope of this paper. In general, for inference, i.e., for understanding the way a response is affected as some predictors change, supervised models are certainly to prefer.

In the future, we want to validate the applicability to other fields inside and outside software engineering. In a recent work, Ferenc et al. (2020) proposed a novel dataset with a wide range of code metrics and bug information. We will use this dataset for the validation of our models on a larger scale.

More importantly, since our approach is agnostic with respect to the application domain, we will apply it on datasets of other application areas in the UC Irvine Machine Learning

Repository (UCI)[17]. Based on this dataset, we currently design a benchmark for an objective comparison of general aggregation approaches that will be made publicly available.

Finally, we plan to apply our approaches of weight generation and weighted aggregation to multi-criteria optimization problems.

Acknowledgements We are grateful to the authors D'Ambros et al. (2010) and Kamei et al. (2013), who made their data publicly available. We are also grateful to the authors (Fu and Menzies 2017), who made their well-packaged and well-documented software available for others[18]. This work was supported by the Linnaeus Excellence Center on Data Intensive Sciences and Applications (DISA)[19]. We thank the anonymous reviewers whose comments and suggestions helped us improve and clarify the research.

Funding Open access funding provided by Linnaeus University.

Declarations

Conflict of Interests The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Arisholm E, Briand LC, Johannessen EB (2010) A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. J Sys Softw 83(1):2–17

Baggen R, Correia JP, Schill K, Visser J (2012) Standardized code quality benchmarking for improving software maintainability. Software Quality Journal 20(2):287–307

Bansiya J, Davis CG (2002) A hierarchical model for object-oriented design quality assessment. IEEE Transactions on Software Engineering 28(1):4–17

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

Borsboom D, Mellenbergh GJ, Van Heerden J (2003) The theoretical status of latent variables. Psychol Rev 110(2):203

Calvo T, Kolesárová A, Komorníková M, Mesiar R (2002) Aggregation operators: properties, classes and construction methods. In: Aggregation operators. Springer, pp 3–104

Carlsson C, Fullér R (1995) Multiple criteria decision making: the case for interdependence. Comput Oper Res 22(3):251–260

Chen X, Zhao Y, Wang Q, Yuan Z (2018) Multi: Multi-objective effort-aware just-in-time software defect prediction. Inf Softw Technol 93:1–13

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

Choo EU, Schoner B, Wedley WC (1999) Interpretation of criteria weights in multicriteria decision making. Computers & Industrial Engineering 37(3):527–541

D'Ambros M, Lanza M, Robbes R (2010) An extensive comparison of bug prediction approaches. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010). IEEE, pp 31–41

[17] UCI Machine Learning Repository, https://archive.ics.uci.edu/ml/index.php

[18] https://github.com/WeiFoo/RevisitUnsupervised

[19] Linnaeus Excellence Center on Data Intensive Sciences and Applications (DISA), https://lnu.se/disa

D'Ambros M, Lanza M, Robbes R (2012) Evaluating defect prediction approaches: a benchmark and an extensive comparison. Empirical Softw Eng 17(4-5):531–577

Easterbrook S, Singer J, Storey MA, Damian D (2008) Selecting empirical methods for software engineering research. In: Guide to Advanced Empirical Software Engineering. Springer, pp 285–311

Fenton N, Bieman J (2014) Software metrics: a rigorous and practical approach. CRC press, Boca Raton

Ferenc R, Gyimesi P, Gyimesi G, Tóth Z, Gyimóthy T (2020) An automatically created novel bug dataset and its validation in bug prediction. J Sys Softw, pp 110691

Fu W, Menzies T (2017) Revisiting unsupervised learning for defect prediction. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, pp 72–83

Garvin DA (1984) What does "product quality" really mean. Sloan management review 25

Gil Y, Lalouche G (2017) On the correlation between size and metric validity. Empir Softw Eng 22(5):2585–2611

Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting fault incidence using software change history. IEEE Trans Softw Eng 26(7):653–661

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

Hassan AE (2009) Predicting faults using the complexity of code changes. In: 2009 IEEE 31st international conference on software engineering. IEEE, pp 78–88

Henderson-Sellers B (1995) Object-oriented metrics: measures of complexity. Prentice-Hall, Inc.

Hinton GE, Roweis S (2003) Stochastic neighbor embedding. In: Becker S, Thrun S, Obermayer K (eds) Advances in neural information processing systems, vol 15. MIT Press. https://proceedings.neurips.cc/paper/2002/file/6150ccc6069bea6b5716254057a194ef-Paper.pdf

Huang Q, Xia X, Lo D (2019) Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. Empir Softw Eng 24(5):2823–2862

IEEE (1990) IEEE std 610.12-1990, standard glossary of software engineering terminology. https://doi.org/10.1109/IEEESTD.1990.101064

ISO/IEC (2010) ISO/IEC 25010 system and software quality models. Tech. rep.

Jiarpakdee J, Tantithamthavorn C, Treude C (2020) The impact of automated feature selection techniques on the interpretation of defect models. Empir Softw Eng, pp 1–49

Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2013) A large-scale empirical study of just-in-time quality assurance. IEEE Trans Softw Eng 39(6):757–773

Khoshgoftaar TM, Allen EB (1999) A comparative study of ordering and classification of fault-prone software modules. Empir Softw Eng 4(2):159–186

Kim S, Zimmermann T, Whitehead Jr EJ, Zeller A (2007) Predicting faults from cached history. In: 29th International Conference on Software Engineering (ICSE'07). IEEE, pp 489–498

Koru AG, Zhang D, El Emam K, Liu H (2008) An investigation into the functional form of the size-defect relationship for software modules. IEEE Trans Softw Eng 35(2):293–304

Letouzey JL, Coq T (2010) The sqale analysis model: An analysis model compliant with the representation condition for assessing the quality of software source code. In: 2010 Second International Conference on Advances in System Testing and Validation Lifecycle (VALID). IEEE, pp 43–48

Linstone HA, Turoff M, et al. (1975) The delphi method. Addison-Wesley Reading, Boston

Liu J, Zhou Y, Yang Y, Lu H, Xu B (2017) Code churn: A neglected metric in effort-aware just-in-time defect prediction. In: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, pp 11–19

van der Maaten L, Hinton G (2008) Visualizing data using t-sne. J Mach Learn Res 9(86):2579–2605. http://jmlr.org/papers/v9/vandermaaten08a.html

van der Maaten L, Postma E, van den Herik J (2009) Dimensionality reduction: a comparative review. J Mach Learn Res 10:66–71

Mai JF, Scherer M (2009) Lévy-frailty copulas. J Multivar Anal 100(7):1567–1585

Marichal JL (2000) An axiomatic approach of the discrete choquet integral as a tool to aggregate interacting criteria. IEEE Trans Fuzzy Syst 8(6):800–807

Martin RC (2002) Agile software development: principles, patterns, and practices. Prentice Hall

Mende T, Koschke R (2009) Revisiting the evaluation of defect prediction models. In: Proceedings of the 5th International Conference on Predictor Models in Software Engineering, pp 1–10

Mende T, Koschke R (2010) Effort-aware defect prediction models. In: 2010 14th European conference on software maintenance and reengineering. IEEE, pp 107–116

Menzies T, Greenwald J, Frank A (2006) Data mining static code attributes to learn defect predictors. IEEE Trans Softw Eng 33(1):2–13

Menzies T, Milton Z, Turhan B, Cukic B, Jiang Y, Bener A (2010) Defect prediction from static code features: current results, limitations, new approaches. Automated Softw Eng 17(4):375–407

Mockus A, Weiss DM (2000) Predicting risk of software changes. Bell Labs Technical Journal 5(2):169–180
Moser R, Pedrycz W, Succi G (2008) A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proceedings of the 30th international conference on software engineering, pp 181–190
Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th international conference on software engineering, pp 284–292
Nelsen RB (2007) An introduction to copulas. Springer Science & Business Media, Berlin
Newbold P, Carlson WL, Thorne B (2013) Statistics for business and economics. Pearson, Boston
Nikora AP, Munson JC (2004) Developing fault predictors for evolving software systems. In: Proceedings. 5th International workshop on enterprise networking and computing in healthcare industry (IEEE Cat. No. 03EX717). IEEE, pp 338–350
Ohlsson N, Alberg H (1996) Predicting fault-prone software modules in telephone switches. IEEE Trans Softw Eng 22(12):886–894
Ostrand TJ, Weyuker EJ, Bell RM (2005) Predicting the location and number of faults in large software systems. IEEE Trans Softw Eng 31(4):340–355
Pearson K (1901) On lines and planes of closest fit to systems of points in space. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 2(11):559–572. https://doi.org/10.1080/14786440109462720
Saaty T (2008) Decision making with the analytic hierarchy process. International Journal of Services Sciences 1(1):83–98
Saaty TL (1996) Decision making with dependence and feedback: The analytic network process, vol 4922. RWS Publ
Schenkerman S (1991) Use and abuse of weights in multiple objective decision support models. Decision Sciences 22(2):369–378
Steiger JH (1980) Tests for comparing elements of a correlation matrix. Psychol Bull 87(2):245
Triantaphyllou E (2000) Multi-criteria decision making methods. In: Multi-criteria decision making methods: A comparative study. Springer, pp 5–21
Wagner S (2013) Software product quality control. Springer, Berlin
Wagner S, Goeb A, Heinemann L, Kläs M, Lampasona C, Lochmann K, Mayr A, Plösch R, Seidl A, Streit J, et al. (2015) Operationalised product quality models and assessment: The quamoco approach. Inf Softw Technol 62:101–123
Wang MW, Stanley JC (1970) Differential weighting: A review of methods and empirical studies. Rev Educ Res 40(5):663–705
Yang X, Lo D, Xia X, Sun J (2017) Tlel: A two-layer ensemble learning approach for just-in-time defect prediction. Inf Softw Technol 87:206–220
Yang Y, Zhou Y, Liu J, Zhao Y, Lu H, Xu L, Xu B, Leung H (2016) Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In: Proceedings of the 2016 24th ACM SIGSOFT International symposium on foundations of software engineering, pp 157–168
Zeleny M (2012) Multiple criteria decision making, vol 123. Springer Science & Business Media, Berlin

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.