Evaluating defect prediction approaches: a benchmark and an extensive comparison
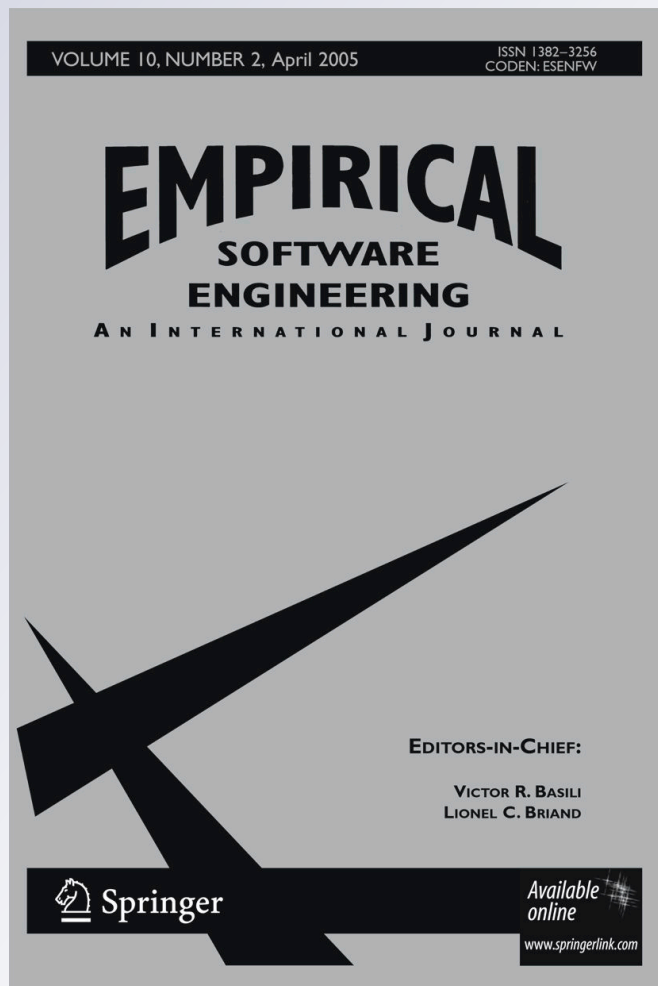
Marco D'Ambros, Michele Lanza & Romain Robbes

Springer

Springer

Evaluating defect prediction approaches: a benchmark and an extensive comparison

Marco D'Ambros · Michele Lanza · Romain Robbes

Abstract Reliably predicting software defects is one of the holy grails of software engineering. Researchers have devised and implemented a plethora of defect/bug prediction approaches varying in terms of accuracy, complexity and the input data they require. However, the absence of an established benchmark makes it hard, if not impossible, to compare approaches. We present a benchmark for defect prediction, in the form of a publicly available dataset consisting of several software systems, and provide an extensive comparison of well-known bug prediction approaches, together with novel approaches we devised. We evaluate the performance of the approaches using different performance indicators: classification of entities as defect-prone or not, ranking of the entities, with and without taking into account the effort to review an entity. We performed three sets of experiments aimed at (1) comparing the approaches across different systems, (2) testing whether the differences in performance are statistically significant, and (3) investigating the stability of approaches across different learners. Our results indicate that, while some approaches perform better than others in a statistically significant manner, external validity in defect prediction is still an open problem, as generalizing results to different contexts/learners proved to be a partially unsuccessful endeavor.

Keywords Defect prediction · Source code metrics · Change metrics

M. D'Ambros (✉) · M. Lanza
REVEAL @ Faculty of Informatics,
University of Lugano, 6900, Lugano, Switzerland
e-mail: marco.dambros@usi.ch

M. Lanza
e-mail: michele.lanza@usi.ch

R. Robbes
PLEIAD Lab @ Computer Science Department (DCC),
University of Chile, Santiago, Chile
e-mail: rrobbes@dcc.uchile.cl

🙂 Springer

1 Introduction

Defect prediction has generated widespread interest for a considerable period of time. The driving scenario is resource allocation: Time and manpower being finite resources, it seems sensible to assign personnel and/or resources to areas of a software system with a higher probable quantity of bugs.

Many approaches have been proposed to tackle the problem, relying on diverse information, such as code metrics (e.g., lines of code, complexity) (Basili et al. 1996; Ohlsson and Alberg 1996; Briand et al. 1999; El Emam et al. 2001; Subramanyam and Krishnan 2003; Gyimóthy et al. 2005; Nagappan and Ball 2005a; Nagappan et al. 2006), process metrics (e.g., number of changes, recent activity) (Nagappan and Ball 2005b; Hassan 2009; Moser et al. 2008; Bernstein et al. 2007), or previous defects (Kim et al. 2007; Ostrand et al. 2005; Hassan and Holt 2005). The jury is still out on the relative performance of these approaches. Most of them have been evaluated in isolation, or were compared to only few other approaches. Moreover, a significant portion of the evaluations cannot be reproduced since the data used for their evaluation came from commercial systems and is not available for public consumption. In some cases, researchers even reached opposite conclusions: For example, in the case of size metrics, Gyimóthy et al. (2005) reported good results, as opposed to the findings of Fenton and Ohlsson (2000).

What is sorely missing is a baseline against which the approaches can be compared. We provide such a baseline by gathering an extensive dataset composed of several open-source systems. Our dataset contains the information required to evaluate several approaches across the bug prediction spectrum.

How to actually evaluate the performance of approaches is also subject to discussion. Some use binary classification (i.e., predicting if a given entity is buggy or not), while others predict a ranking of components prioritizing the ones with the most defects. Finally, some prediction models take into account the effort required to inspect an entity as a performance evaluation metric. Hence, a comparison of how each approach performs over several evaluation metrics is also warranted.

Contributions   The contributions of this paper are:

– A public benchmark for defect prediction, containing sufficient data to evaluate a variety of approaches. For five open-source software systems, we provide, over a five-year period, the following data:

   – process metrics on all the files of each system,
   – system metrics on bi-weekly versions of each system,
   – defect information related to each system file, and
   – bi-weekly models of each system version if new metrics need to be computed.

– Two novel bug prediction approaches based on bi-weekly source code samples:

   1. The first, similarly to Nikora and Munson (2003), measures code churn as deltas of high-level source code metrics instead of line-based code churn.
   2. The second extends Hassan's concept of entropy of changes (Hassan 2009) to source code metrics.

– The evaluation of several defect prediction approaches in three distinct series of experiments, aimed respectively at (1) comparing the performance of the approaches across different systems, (2) testing whether the differences are statistically significant, and (3) studying the stability of approaches across learners. We evaluate the approaches according to various scenarios and performance measures:

  – a binary classification scenario, evaluated with ROC curves;
  – a ranking-based evaluation using cumulative lift charts of the numbers of predicted bugs; and
  – an effort-aware ranking-based evaluation, where effort is defined as the size in lines of code of each entity.

– An extensive discussion of the overall performance of each approach on all case studies, according to several facets: (1) the performance criteria mentioned above; (2) the variability of performance across a number of runs of the experiment; and (3) the variability of the performance across learners. We also comment on the quantity of data necessary to apply each approach, and the performance of approximations of sets of metrics by single metrics.

Structure of the Paper In Section 2 we present an overview of related work in defect prediction. In Section 3 we detail the approaches that we reproduce and the ones that we introduce. We describe our benchmark and evaluation procedure in Section 4. We report on the performance in Section 5 and Section 6, and investigate issues of stability across various learners in Section 7. We discuss finer-grained aspects of the performance in Section 8. In Section 9, we discuss possible threats to the validity of our findings, and we conclude in Section 10.

2 Related Work in Defect Prediction

We describe several defect prediction approaches, the kind of data they require and the various data sets on which they were validated. All approaches require a defect archive to be validated, but they do not necessarily require it to actually perform their analysis. When they do, we indicate it.

Change Log Approaches use process metrics extracted from the versioning system, assuming that recently or frequently changed files are the most probable source of future bugs.

Khoshgoftaar et al. (1996) classified modules as defect-prone based on the number of past modifications to the source files composing the module. They showed that the number of lines added or removed in the past is a good predictor for future defects at the module level.

Graves et al. (2000) devised an approach based on statistical models to find the best predictors for modules' future faults. The authors found that the best predictor is the sum of contributions to a module in its history.

Nagappan and Ball performed a study on the influence of code churn (i.e., the amount of change to the system) on the defect density in Windows Server 2003. They

found that relative code churn is a better predictor than absolute churn (Nagappan and Ball 2005b).

Hassan (2009) introduced the concept of entropy of changes, a measure of the complexity of code changes. Entropy was compared to the amount of changes and the amount of previous bugs, and was found to be often better. The entropy metric was evaluated on six open-source systems: FreeBSD, NetBSD, OpenBSD, KDE, KOffice, and PostgreSQL.

Moser et al. (2008) used metrics (including code churn, past bugs and refactorings, number of authors, file size and age, etc.), to predict the presence/absence of bugs in files of Eclipse.

Nikora and Munson (2003) introduced the churn of code metrics, measuring the differences of various size and control flow characteristics of the source code, over multiple releases of evolving software systems. They showed, on C/C++ systems, that such measures can be used to predict defects.

The mentioned techniques do not make use of the defect archives to predict bugs, while the following ones do.

Hassan and Holt's (2005) top ten list approach validates heuristics about the defect-proneness of the most recently changed and most bug-fixed files, using the defect repository data. The approach was validated on six open-source case studies: FreeBSD, NetBSD, OpenBSD, KDE, KOffice, and PostgreSQL. They found that recently modified and fixed entities are the most defect-prone.

Ostrand et al. (2005) predicted faults on two industrial systems, using change and defect data.

The bug cache approach by Kim et al. (2007) uses the same properties of recent changes and defects as the top ten list approach, but further assumes that faults occur in bursts. The bug-introducing changes are identified from the SCM logs. Seven open-source systems were used to validate the findings (Apache, PostgreSQL, Subversion, Mozilla, JEdit, Columba, and Eclipse).

Bernstein et al. (2007) used bug and change information in non-linear prediction models. Six eclipse plugins were used to validate the approach.

Single-Version Approaches assume that the current design and behavior of the program influences the presence of future defects. These approaches do not require the history of the system, but analyze its current state in more detail, using a variety of metrics.

One standard set of metrics used is the Chidamber and Kemerer (CK) metrics suite (Chidamber and Kemerer 1994). Basili et al. (1996) used the CK metrics on eight medium-sized information management systems based on the same requirements. Ohlsson and Alberg (1996) used several graph metrics including McCabe's cyclomatic complexity on an Ericsson telecom system. El Emam et al. (2001) used the CK metrics in conjunction with Briand's coupling metrics (Briand et al. 1999) to predict faults on a commercial Java system. Subramanyam and Krishnan (2003) used the CK metrics on a commercial C++/Java system. Gyimóthy et al. (2005) performed a similar analysis on Mozilla.

Researchers also used other metric suites: Nagappan and Ball (2005a) estimated the pre-release defect density of Windows Server 2003 with a static analysis tool. Nagappan et al. (2006) also used a catalog of source code metrics to predict post-release defects at the module level on five Microsoft systems, and found that it was

possible to build predictors for one individual project, but that no predictor would perform well on all the projects. Zimmermann et al. (2007) used a number of code metrics on Eclipse.

Menzies et al. (2007) argued that the exact static source code metric used is not as important as which learning algorithm is used. Based on data from the NASA Metrics Data Program (MDP) to arrive to the above conclusion, the authors compared the impact of using the LOC, Halstead and McCabe metrics (and a variety of other metrics), versus the impact of using the Naive Bayes, OneR, and J48 algorithms.

Effort-Aware Defect Prediction   Several recent works take into account the effort necessary to inspect a file during defect prediction. The time spent reviewing a potentially buggy file depends on its size and complexity. Hence for an equivalent amount of bugs to discover, a shorter file involves less effort (Arisholm et al. 2010). The intent of effort-aware bug prediction is hence not to predict if a file is buggy or not, but rather to output a set of files for which the ratio of effort spent for number of bugs found is maximized.

Mende and Koschke (2009) showed that if a trivial defect prediction model— predicting that large files are the most buggy—performs well with a classic evaluation metric such as the ROC curve, it performs significantly worse when an effort-aware performance metric is used. Later, they evaluated two effort-aware models and compared them to a classical prediction model (Mende and Koschke 2010).

Likewise, Kamei et al. (2010) revisited common findings in defect prediction when using effort-aware performance measurements. One finding that they confirmed is that process metrics (i.e., extracted from the version control system or the defect database) still perform better than product metrics (i.e., metrics of the software system itself).

Arisholm and Briand (2006) conducted a study on a large object-oriented legacy system, building logistic regression models with a variety of metrics such as structural measures, source code metrics, fault measures, and history data from previous releases. To take the effort into account, the authors introduced a methodology to assess the cost-effectiveness of the prediction to focus verification effort.

Koru et al. showed—initially on Mozilla (Koru et al. 2007), and in a subsequent work on ten KOffice open source products and two commercial systems (Koru et al. 2008)—that smaller modules are proportionally more defect prone than larger ones. They thus recommended to focus quality assurance resources on smaller modules, as they are more cost effective, i.e., more defects will be found in the same amount of code.

Menzies et al. (2010) also took into account the effort in their defect prediction approach, and found that some prediction algorithms are outperformed by manual methods, when using static code metric data. They however did not consider process metrics, as they focused more on the performance differences of algorithms, and not on the impact of different data sources.

Other Approaches   Ostrand et al. conducted a series of studies on the whole history of different systems to analyze how the characteristics of source code files can predict defects (Ostrand and Weyuker 2002; Ostrand et al. 2004, 2007). On this basis, they proposed an effective and automated predictive model based on these characteristics (e.g., age, lines of code, etc.) (Ostrand et al. 2007).

Binkley and Schach ([1998](#)) devised a coupling dependency metric and showed that it outperforms several other metrics in predicting run-time failures. Zimmermann and Nagappan ([2008](#)) used dependencies between binaries in Windows server 2003 to predict defects with graph-centric metrics.

Marcus et al. ([2008](#)) used a cohesion measurement based on the vocabulary used in documents, as measured by Latent Semantic Indexing (LSI) for defect prediction on several C++ systems, including Mozilla.

Arnaoudova et al. ([2010](#)) showed initial evidence that identifiers that are used in a variety of contexts may have an impact on fault-proneness.

Neuhaus et al. ([2007](#)) used a variety of features of Mozilla (past bugs, package imports, call structure) to detect vulnerabilities.

Shin et al. ([2009](#)) also investigated the usefulness of the call structure of the program, and found that the improvement on models based on non-call structure is significant, but becomes marginal when history is taken into account.

Pinzger et al. ([2008](#)) empirically investigated the relationship between the fragmentation of developer contributions and the number of post-release defects. To do so, they measured the fragmentation of contributions with network centrality metrics computed on a developer-artifact network.

Wolf et al. ([2009](#)) analyzed the network of communications between developers to understand how they are related to issues in integration of modules of a system. They conceptualized communication as based on developer's comments on work items.

Zimmermann et al. ([2009](#)) tackled the problem of cross-project defect prediction, i.e., computing prediction models from a project and applying it on a different one. Their experiments showed that using models from projects in the same domain or with the same process does not lead to accurate predictions. Therefore, the authors identified important factors influencing the performance of cross-project predictors.

Turhan et al. ([2009](#)) also investigated cross-project defect prediction. In a first experiment, they obtained results similar to the ones of Zimmermann et al., where a defect prediction model learned from one project provides poor performances on other projects, in particular with respect to the rate of false positives. With a deeper inspection, they found out that such low performances were due to irrelevancies, i.e., the model was learning from numerous irrelevant details from other projects. Turhan et al. thus applied a relevancy filtering technique, which drastically reduced the ratio of false positives.

The same authors reported that importing data from other projects can indeed improve the performance of the prediction, at the cost of increasing also false positives (Turhan et al. [2010](#)).

Bacchelli et al. ([2010](#)) investigated whether integrating information from e-mails complements other data sources. The intuition is that problematic classes are more often discussed in email conversations than classes that have less problems.

2.1 Observations

We observe that both *case studies* and the *granularity of approaches* vary. Distinct case studies make a comparative evaluation of the results difficult. Validations performed on industrial systems are not reproducible, because it is not possible to

obtain the underlying data. There is also variation among open-source case studies, as some approaches have more restrictive requirements than others. With respect to the granularity of the approaches, some of them predict defects at the class level, others consider files, while others consider modules or directories (subsystems), or even binaries. While some approaches predict the presence or absence of bugs for each component (the classification scenario), others predict the amount of bugs affecting each component in the future, producing a ranked list of components.

Replication   The notion of replication has gained acceptance in the Mining Software Repositories community (Robles 2010; Juristo and Vegas 2009). It is also one of the aims of the PROMISE conference series.[1] Mende (2010) attempted to replicate two defect prediction experiments, including an earlier version of this one (D'Ambros et al. 2010). Mende was successful in replicating the earlier version of this experiment, but less so for the other one. This does cast a shadow on a substantial amount of research performed in the area, which can only be lifted through *benchmarking*.

Benchmarking allows comparison or replication of approaches and stimulate a community (Sim et al. 2003). The PROMISE data set is such a benchmark, as is the NASA MDP project. The existence of such data sets allows for systematic comparison, as the one by Menzies et al. (2010), or Lessmann et al. (2008), where different prediction algorithms (Regression Models, Support Vector Machines, Random Forests, etc.), were trained on the same data and compared. The authors could only show that the difference between at least two algorithms was significant, but could not show that most classifiers performed significantly different from one another. We could however not use these data sets, as they do not include all the data sources we need (e.g., bi-weekly snapshots of the source code are still a rare occurrence). This lack of data sources limits the extensibility of these existing benchmarks to evaluate novel metrics.

These observations explain the lack of comparison between approaches and the occasional diverging results when comparisons are performed. For this reason, we propose a benchmark to establish a common ground for comparison. Unlike most other datasets, our benchmark provides means of extension through the definition of additional metrics, since it also includes sequences of high-level models of the source code in addition to the metrics themselves. We first introduce the approaches that we compare, before describing our benchmark dataset and evaluation strategies.

3 Bug Prediction Approaches

Considering the bulk of research performed in this area it is impossible to compare all existing approaches. To cover a range as wide as possible, we selected one or more approaches from each category, summarized in Table 1.

---

[1]http://promisedata.org

Table 1 Categories of bug prediction approaches

Type	Rationale	Used by
Process metrics	Bugs are caused by changes.	Moser et al. (2008)
Previous defects	Past defects predict future defects.	Kim et al. (2007)
Source code metrics	Complex components are harder to change, and hence error-prone.	Basili et al. (1996)
Entropy of changes	Complex changes are more error-prone than simpler ones.	Hassan (2009)
Churn (source code metrics)	Source code metrics are a better approximation of code churn.	Novel
Entropy (source code metrics)	Source code metrics better describe the entropy of changes.	Novel

3.1 Process Metrics

We selected the approach of Moser et al. as a representative, and describe three additional variants.

MOSER We use the catalog of file-level process metrics introduced by Moser et al. (2008) listed in Table 2.

The metric NFIX represents the number of bug fixes as extracted from the versioning system, not the defect archive. It uses a heuristic based on pattern matching on the comments of every commit. To be recognized as a bug fix, the comment must match the string "%fix%" and not match the strings "%prefix%" and "%postfix%". The bug repository is not needed, because all the metrics are extracted from the CVS/SVN logs, thus simplifying data extraction. For systems versioned using SVN (such as Lucene) we perform some additional data extraction, since the SVN logs do not contain information about lines added and removed.

Table 2 Change metrics used by Moser et al. (2008)

Name	Description
NR	Number of revisions
NREF	Number of times a file has been refactored
NFIX	Number of times a file was involved in bug-fixing
NAUTH	Number of authors who committed the file
LINES	Lines added and removed (sum, max, average)
CHURN	Codechurn (sum, maximum and average) Codechurn is computed as $\sum_R(added\ LOC - deleted\ LOC)$, where R is the set of all revisions
CHGSET	Change set size, i.e., number of files committed together to the repository (maximum and average)
AGE	Age (in number of weeks) and weighted age computed as $\frac{\sum_{i=1}^N Age(i) \times added\ LOC(i)}{\sum_{i=1}^N added\ LOC(i)}$, where $Age(i)$ is the number of weeks starting from the release date for revision i, and $added\ LOC(i)$ is the number of lines of code added at revision i

NFIX   Zimmermann et al. (2007) showed that the number of past defects has the highest correlation with number of future defects. We inspect the accuracy of the bug fix approximation in isolation.

NR   In the same fashion, since Graves et al. (2000) showed that the best generalized linear models for defect prediction are based on number of changes, we isolate the number of revisions as a predictive variable.

NFIX+NR   We combine the previous two approaches.

3.2 Previous Defects

This approach relies on a single metric to perform its prediction. We also describe a more fine-grained variant exploiting the categories present in defect archives.

BUGFIXES   The bug prediction approach based on previous defects, proposed by Zimmermann et al. (2007), states that the number of past bug fixes extracted from the repository is correlated with the number of future fixes. They then use this metric in the set of metrics with which they predict future defects. This measure is different from the metric used in **NFIX-ONLY** and **NFIX+NR**: For **NFIX**, we perform pattern matching on the commit comments. For **BUGFIXES**, we also perform the pattern matching, which in this case produces a list of potential defects. Using the defect id, we check whether the bug exists in the bug database, we retrieve it and we verify the consistency of timestamps (i.e., if the bug was reported before being fixed).

BUG-CATEGORIES   We also use a variant in which as predictors we use the number of bugs belonging to five categories, according to severity and priority. The categories are: All bugs, non trivial bugs (severity>trivial), major bugs (severity>major), critical bugs (critical or blocker severity) and high priority bugs (priority>default).

3.3 Source Code Metrics

Many approaches in the literature use the CK metrics. We compare them with additional object-oriented metrics, as well as lines of code (LOC). Table 3 lists all source code metrics we use.

CK   Many bug prediction approaches are based on metrics, in particular the Chidamber & Kemerer suite (Chidamber and Kemerer 1994).

OO   An additional set of object-oriented metrics.

CK+OO   The combination of the two sets of metrics.

LOC   Gyimóthy et al. (2005) showed that lines of code (**LOC**) is one of the best metrics for fault prediction. In addition to incorporating it with the OO metrics, we evaluate its accuracy as a defect predictor in isolation.

Table 3 Class level source code metrics

Type	Name	Description
CK	WMC	Weighted method count
CK	DIT	Depth of inheritance tree
CK	RFC	Response for class
CK	NOC	Number of children
CK	CBO	Coupling between objects
CK	LCOM	Lack of cohesion in methods
OO	FanIn	Number of other classes that reference the class
OO	FanOut	Number of other classes referenced by the class
OO	NOA	Number of attributes
OO	NOPA	Number of public attributes
OO	NOPRA	Number of private attributes
OO	NOAI	Number of attributes inherited
OO	LOC	Number of lines of code
OO	NOM	Number of methods
OO	NOPM	Number of public methods
OO	NOPRM	Number of private methods
OO	NOMI	Number of methods inherited

3.4 Entropy of Changes

Hassan predicted defects using the entropy (or complexity) of code changes (Hassan 2009). The idea consists in measuring, over a time interval, how distributed changes are in a system. The more spread the changes are, the higher the complexity. The intuition is that a change affecting one file only is simpler than another affecting many different files, as the developer who has to perform the change has to keep track of all of them. Hassan proposed to use the Shannon Entropy defined as

$$H_n(P) = -\sum_{k=1}^{n} p_k * \log_2 p_k \tag{1}$$

where p_k is the probability that the file k changes during the considered time interval. Figure 1 shows an example with three files and three time intervals.

In the fist time interval $t1$, we have four changes, and the change frequencies of the files (i.e., their probability of change) are $p_A = \frac{2}{4}$, $p_B = \frac{1}{4}$, $p_C = \frac{1}{4}$.

The entropy in $t1$ is: $H = -(0.5 * \log_2 0.5 + 0.25 * \log_2 0.25 + 0.25 * \log_2 0.25) = 1$.

In $t2$, the entropy is higher: $H = -(\frac{2}{7} * \log_2 \frac{2}{7} + \frac{1}{7} * \log_2 \frac{1}{7} + \frac{4}{7} * \log_2 \frac{4}{7}) = 1.378$.

Fig. 1 An example of entropy of code changes
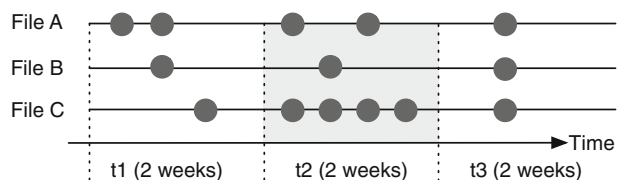
As in Hassan's approach (Hassan 2009), to compute the probability that a file changes, instead of simply using the number of changes, we take into account the amount of change by measuring the number of modified lines (lines added plus deleted) during the time interval. Hassan defined the Adaptive Sizing Entropy as:

$$H' = -\sum_{k=1}^{n} p_k * \log_{\bar{n}} p_k \tag{2}$$

where n is the number of files in the system and $\bar{n}$ is the number of recently modified files. To compute the set of recently modified files we use previous periods (e.g., modified in the last six time intervals). To use the entropy of code change as a bug predictor, Hassan defined the History of Complexity Metric (HCM) of a file j as

$$HCM_{\{a,..,b\}}(j) = \sum_{i \in \{a,..,b\}} HCPF_i(j) \tag{3}$$

where $\{a, .., b\}$ is a set of evolution periods and $HCPF$ is:

$$HCPF_i(j) = \begin{cases} c_{ij} * H'_i, & j \in F_i \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

where i is a period with entropy H'_i, F_i is the set of files modified in the period i and j is a file belonging to F_i. According to the definition of c_{ij}, we test two metrics:

- **HCM:** $c_{ij} = 1$, every file modified in the considered period i gets the entropy of the system in the considered time interval.
- **WHCM:** $c_{ij} = p_j$, each modified file gets the entropy of the system weighted with the probability of the file being modified.
- $c_{ij} = \frac{1}{|F_i|}$ the entropy is evenly distributed to all the files modified in the i period. We do not use this definition of c_{ij} since Hassan showed that it performs less well than the other.

Concerning the periods used for computing the History of Complexity Metric, we use two weeks time intervals.

Variants We define three further variants based on **HCM**, with an additional weight for periods in the past. In **EDHCM** (Exponentially Decayed HCM, introduced by Hassan), entropies for earlier periods of time, i.e., earlier modifications, have their contribution exponentially reduced over time, modelling an exponential decay model. Similarly, **LDHCM** (Linearly Decayed) and **LGDHCM** (LoGarithmically decayed), have their contributions reduced over time in a respectively linear and logarithmic fashion. Both are novel. The definition of the variants follows (ϕ_1, ϕ_2 and ϕ_3 are the decay factors):

$$EDHCM_{\{a,..,b\}}(j) = \sum_{i \in \{a,..,b\}} \frac{HCPF_i(j)}{e^{\phi_1 * (|\{a,..,b\}| - i)}} \tag{5}$$

$$LDHCM_{\{a,..,b\}}(j) = \sum_{i \in \{a,..,b\}} \frac{HCPF_i(j)}{\phi_2 * (|\{a,..,b\}| + 1 - i)} \tag{6}$$

$$LGDHCM_{\{a,..,b\}}(j) = \sum_{i \in \{a,..,b\}} \frac{HCPF_i(j)}{\phi_3 * \ln(|\{a,..,b\}| + 1.01 - i)} \tag{7}$$

3.5 Churn of Source Code Metrics

The first—and to the best of our knowledge the only one—to use the churn of source code metrics to predict post release defects were Nikora and Munson (2003). The intuition is that higher-level metrics may better model code churn than simple metrics like addition and deletion of lines of code. We sample the history of the source code every two weeks and compute the deltas of source code metrics for each consecutive pair of samples.

For each source code metric, we create a matrix where the rows are the classes, the columns are the sampled versions, and each cell is the value of the metric for the given class at the given version. If a class does not exist in a version, we indicate that by using a default value of -1. We only consider the classes which exist at release x for the prediction.

We generate a matrix of deltas, where each cell is the absolute value of the difference between the values of a metric—for a class—in two subsequent versions. If the class does not exist in one or both of the versions (at least one value is -1), then the delta is also -1.

Figure 2 shows an example of deltas matrix computation for three classes. The numbers in the squares are metrics; the numbers in circles, deltas. After computing the deltas matrices for each source code metric, we compute churn as:

$$CHU(i) = \sum_{j=1}^{C} \begin{cases} 0, & D(i, j) = -1 \\ PCHU(i, j), & \text{otherwise} \end{cases} \tag{8}$$

$$PCHU(i, j) = D(i, j) \tag{9}$$

where i is the index of a row in the deltas matrix (corresponding to a class), C is the number of columns of the matrix (corresponding to the number of samples considered), $deltas(i, j)$ is the value of the matrix at position (i, j) and $PCHU$ stands for partial churn. For each class, we sum all the cells over the columns –excluding the ones with the default value of -1. In this fashion we obtain a set of churns of source code metrics at the class level, which we use as predictors of post release defects.

Fig. 2 Computing metrics deltas from sampled versions of a system
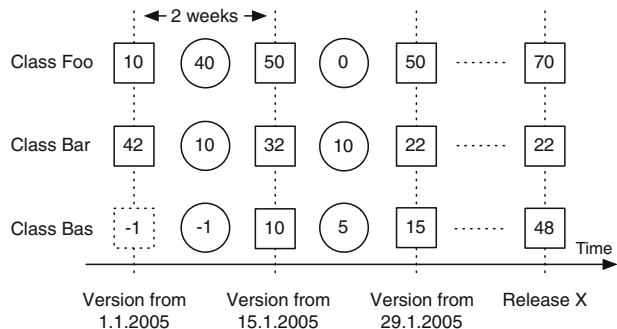
Variants   We define several variants of the partial churn of source code metrics (PCHU): The first one weights more the frequency of change (i.e., delta > 0) than the actual change (the delta value). We call it **WCHU** (weighted churn), using the following partial churn:

$$WPCHU(i, j) = 1 + \alpha * deltas(i, j) \tag{10}$$

where α is the weight factor, set to 0.01 in our experiments. This avoids that a delta of 10 in a metric has the same impact on the churn as ten deltas of 1. We consider many small changes more relevant than few big changes. Other variants are based on weighted churn (**WCHU**) and take into account the decay of deltas over time, respectively in an exponential (**EDCHU**), linear (**LDCHU**) and logarithmic manner (**LGDCHU**), with these partial churns (ϕ_1, ϕ_2 and ϕ_3 are the decay factors):

$$EDPCHU(i, j) = \frac{1 + \alpha * deltas(i, j)}{e^{\phi_1 * (C - j)}} \tag{11}$$

$$LDPCHU(i, j) = \frac{1 + \alpha * deltas(i, j)}{\phi_2 * (C + 1 - j)} \tag{12}$$

$$LGDPCHU(i, j) = \frac{1 + \alpha * deltas(i, j)}{\phi_3 * \ln(C + 1.01 - j)} \tag{13}$$

3.6 Entropy of Source Code Metrics

In the last bug prediction approach we extend the concept of code change entropy (Hassan 2009) to the source code metrics listed in Table 3. The idea is to measure the complexity of the variants of a metric over subsequent sample versions. The more distributed over multiple classes the variants of the metric is, the higher the complexity. For example, if in the system the WMC changed by 100, and only one class is involved, the entropy is minimum, whereas if ten classes are involved with a local change of 10 WMC, then the entropy is higher. To compute the entropy of source code metrics, we start from the matrices of deltas computed as for the churn metrics. We define the entropy, for instance for WMC, for the column j of the deltas matrix, i.e., the entropy between two subsequent sampled versions of the system, as:

$$H'_{WMC}(j) = -\sum_{i=1}^{R} \begin{cases} 0, & deltas(i, j) = -1 \\ p(i, j) * \log_{\bar{R}_j} p(i, j), & \text{otherwise} \end{cases} \tag{14}$$

where R is the number of rows of the matrix, $\bar{R}_j$ is the number of cells of the column j greater than 0 and $p(i, j)$ is a measure of the frequency of change (viewing frequency as a measure of probability, similarly to Hassan) of the class i, for the given source code metric. We define it as:

$$p(i, j) = \frac{deltas(i, j)}{\sum_{k=1}^{R} \begin{cases} 0, & deltas(k, j) = -1 \\ deltas(k, j), & \text{otherwise} \end{cases}} \tag{15}$$

Equation (14) defines an adaptive sizing entropy, since we use $\bar{R}_j$ for the logarithm, instead of R (number of cells greater than 0 instead of number of cells). In the example in Fig. 2 the entropies for the first two columns are:

$$H'(1) = -\frac{40}{50} * \log_2 \frac{40}{50} - \frac{10}{50} * \log_2 \frac{10}{50} = 0.722$$

$$H'(2) = -\frac{10}{15} * \log_2 \frac{10}{15} - \frac{5}{15} * \log_2 \frac{5}{15} = 0.918$$

Given a metric, for example WMC, and a class corresponding to a row i in the deltas matrix, we define the history of entropy as:

$$HH_{WMC}(i) = \sum_{j=1}^{C} \begin{cases} 0, & D(i,\, j) = -1 \\ PHH_{WMC}(i,\, j), & \text{otherwise} \end{cases} \tag{16}$$

$$PHH_{WMC}(i,\, j) = H'_{WMC}(j) \tag{17}$$

where PHH stands for partial historical entropy. Compared to the entropy of changes, the entropy of source code metrics has the advantage that it is defined for every considered source code metric. If we consider "lines of code", the two metrics are very similar: HCM has the benefit that it is not sampled, i.e., it captures all changes recorded in the versioning system, whereas HH_{LOC}, being sampled, might lose precision. On the other hand, HH_{LOC} is more precise, as it measures the real number of lines of code (by parsing the source code), while HCM measures it from the change log, including comments and whitespace.

Variants In (17) each class that changes between two version (delta > 0) gets the entire system entropy. To take into account also how much the class changed, we define the history of weighted entropy HWH, by redefining PHH as:

$$HWH(i,\, j) = p(i,\, j) * H'(j) \tag{18}$$

We also define three other variants by considering the decay of the entropy over time, as for the churn metrics, in an exponential ($EDHH$), linear ($LDHH$), and logarithmic ($LGDHH$) fashion. We define their partial historical entropy as (ϕ_1, ϕ_2 and ϕ_3 are the decay factors):

$$EDHH(i,\, j) = \frac{H'(j)}{e^{\phi_1 * (C - j)}} \tag{19}$$

$$LDHH(i,\, j) = \frac{H'(j)}{\phi_2 * (C + 1 - j)} \tag{20}$$

$$LGDHH(i,\, j) = \frac{H'(j)}{\phi_3 * \ln(C + 1.01 - j)} \tag{21}$$

From these definitions, we define several prediction models using several object-oriented metrics: **HH**, **HWH**, **EDHHK**, **LDHH** and **LGDHH**.

System url	Prediction release	Time period	#Classes	#Versions	#Transactions	#Post-rel. defects
Eclipse JDT Core www.eclipse.org/jdt/core/	3.4	1.01.2005 6.17.2008	997	91	9,135	463
Eclipse PDE UI www.eclipse.org/pde/pde-ui/	3.4.1	1.01.2005 9.11.2008	1,562	97	5,026	401
Equinox framework www.eclipse.org/equinox/	3.4	1.01.2005 6.25.2008	439	91	1,616	279
Mylyn www.eclipse.org/mylyn/	3.1	1.17.2005 3.17.2009	2,196	98	9,189	677
Apache Lucene lucene.apache.org	2.4.0	1.01.2005 10.08.2008	691	99	1,715	103

Fig. 3 Systems in the benchmark

4 Benchmark and Experimental Setup

We compare different bug prediction approaches in the following way: *Given a release* x *of a software system* s, *released at date* d, *the task is to predict, for each class of* x, *the presence (classification), or the number (ranking) of post release defects, i.e., the presence/number of defects reported from* d *to six months later*. We chose the last release of the system in the release period and perform class-level defect prediction, and not package- or subsystem-level defect prediction, for the following reasons:

– Package-level information can be derived from class-level information, while the opposite is not true.
– Classes are the building blocks of object-oriented systems, and are self-contained elements from the point of view of design and implementation.
– Predictions at the package-level are less helpful since packages are significantly larger. The review of a defect-prone package requires more work than a class.

We use post-release defects for validation (i.e., not all defects in the history) to emulate a real-life scenario. As in the work of Zimmermann et al. (2007) we use a six months time interval for post-release defects.

4.1 Benchmark Dataset

To make our experiments reproducible, we created a website[2] where we share our bug prediction dataset. The dataset is a collection of models and metrics of five software systems and their histories. The goal of such a dataset is to allow researchers to compare different defect prediction approaches and to evaluate whether a new technique is an improvement over existing ones. We designed the dataset to perform defect prediction at the class level. However, package or subsystem information can be derived by aggregating class data, since, for each class, the dataset specifies the package that contains it. Our dataset is composed of the change, bug and version information of the five systems detailed in Fig. 3.

---

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

The criteria for choosing these specific systems are:

Size and lifetime	All systems have been released for several years, feature thousands of SCM commits, and feature on the order of hundreds to thousand of classes, so are representative of medium to large systems with a significant history.
Homogeneous language	All systems are written in Java to ensure that all the code metrics are defined identically for each system. By using the same parser, we can avoid issues due to behavior differences in parsing, a known issue for reverse engineering tools (Kollmann et al. 2002). This also allows to ensure that the definition of all metrics are consistent among systems, and not dependent on the presence/absence of a particular language feature.
Availability of the data	All systems provide unrestricted access to version archives that make the gathering of bi-weekly snapshots of the source code possible.

We provide, for each system: the data extracted from the change log, including reconstructed transaction and links from transactions to model classes; the defects extracted from the defect repository, linked to the transactions and the system classes referencing them; bi-weekly versions of the systems parsed into object-oriented models; values of all the metrics used as predictors, for each version of each class of the system; and post-release defect counts for each class.

Our bug prediction dataset is not the only one publicly available. Other datasets exist, such as the PROMISE dataset,[3] and the NASA Metrics Data Program.[4] but none of them provides all the information that ours includes.

The metrics we provide include process measures extracted from versioning system logs, defect information and source code metrics for hundreds of system versions. This extensive set of metrics makes it possible to compute additional metrics such as the novel churn and entropy of source code metrics, and to compare a wider set of defect prediction techniques. To compute some of these metrics, one needs bi-weekly versions of the systems, which our dataset is—to our knowledge— the sole one to provide.

Extensibility of the Benchmark The presence of the FAMIX models makes our dataset extensible by third parties. FAMIX models are fine-grained models of the source code of each snapshot, containing packages, classes, methods, method invocations, and variable access information, among others. This allows the definition of additional metrics that can be expressed with that information, such as network metrics based on method call information (Zimmermann and Nagappan 2008).

The fine-grained representation of the data offered by the FAMIX metamodel makes parsing the data again unnecessary in most cases. This lowers the barrier to entry in defining new metrics based on the available data.

---

[3]http://promisedata.org/data

[4]http://mdp.ivv.nasa.gov, also part of PROMISE.

Moreover, since the data is available in bi-weekly snapshots, every new metric can be also computed in its churn and entropy variants, to take into account the changes of metric values over time and their distribution over entities. We regard the availability of successive versions of FAMIX models as key to increase the durability of our benchmark dataset, as this makes it more adaptable to the needs of other researchers wishing to experiment with additional metrics. Gathering and parsing successive versions of several large systems is a costly endeavor: By providing the snapshots as a prepackaged on the website, we significantly cut down on the time needed by other researchers to gather the data themselves.

Data Collection   Figure 4 shows the types of information needed by the compared bug prediction approaches. In particular, to apply these approaches, we need the following information:

–   change log information to extract process metrics;
–   source code version information to compute source code metrics; and
–   defect information linked to classes for both the prediction and validation.

Figure 5 shows how we gather this information, given an SCM system (CVS or Subversion) and a defect tracking system (Bugzilla or Jira).

Creating a History Model   To compute the various process metrics, we model how the system changed during its lifetime by parsing the versioning system log files. We create a model of the history of the system using the transactions extracted from
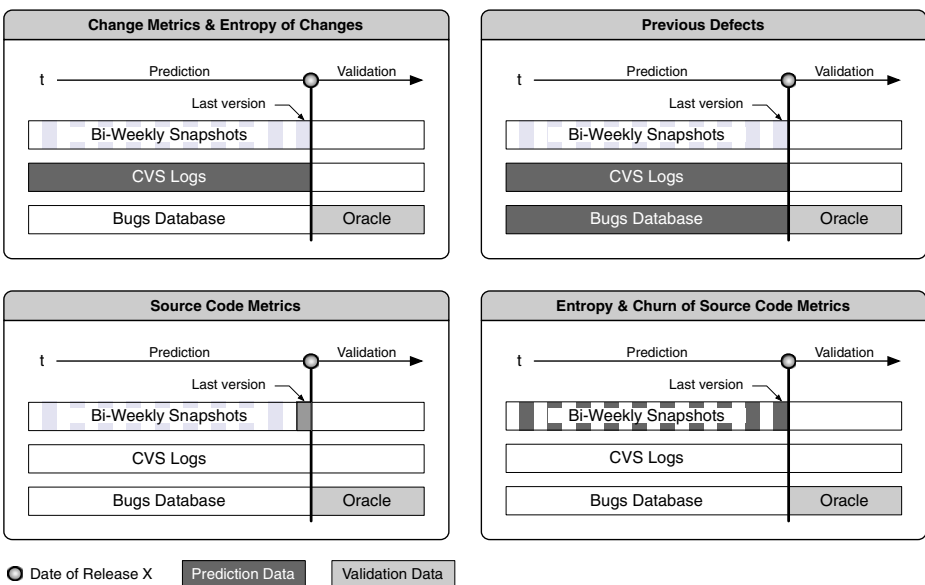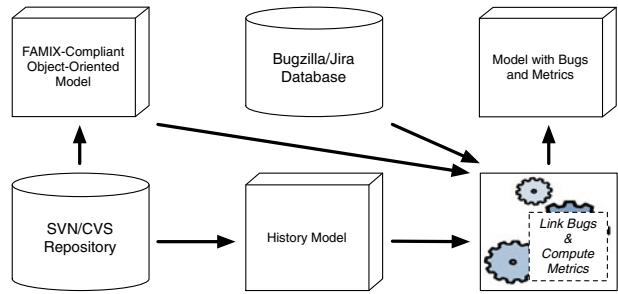

Fig. 4 The types of data used by different bug prediction approaches

Fig. 5 Model with bug, change and history

the system's SCM repository. A transaction (or commit) is a set of files which were modified and committed to the repository, together with the timestamp, the author and the comment. SVN marks co-changing files at commit time as belonging to the same transaction while for CVS we infer transactions from each file's modification time, commit comment, and author.

Creating a Source Code Model We retrieve the source code from the SCM repository and we extract an object-oriented model of it according to FAMIX, a language independent meta-model of object oriented code (Demeyer et al. 2001). FAMIX models the pivotal concepts of object oriented programming, such as classes, methods, attributes, packages, inheritance, etc.

Since we need several versions of the system, we repeat this process at bi-weekly intervals over the history period we consider.

In selecting the sampling time interval size, our goal was to sample frequent enough to capture all significant deltas (i.e., differences) between system versions. We initially selected a time window of one week, as it is very unlikely that a large system dramatically changes over a one week period. Subsequently, we noticed that we could even double the time window—making it two weeks long—without losing significant data and, at the same time, halving the amount of information to be processed. Therefore, we opted for a sampling time interval of two weeks.

Linking Classes with Bugs To reason about the presence of bugs affecting parts of the software system, we first map each problem report to the components of the

Fig. 6 Linking bugs, SCM files and classes
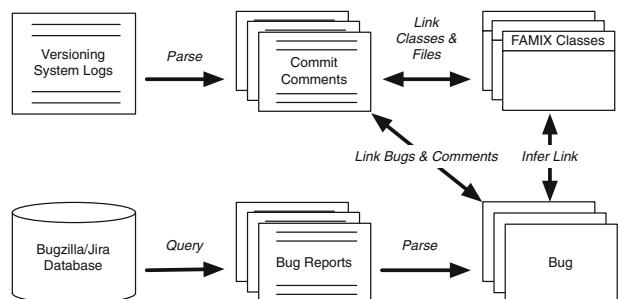
system that it affects. We link FAMIX classes with versioning system files and bugs retrieved from Bugzilla and Jira repositories, as shown in Fig. 6.

A file version in the versioning system contains a developer comment written at commit time, which often includes a reference to a problem report (e.g., "fixed bug 123"). Such references allow us to link problem reports with files in the versioning system (and thus with classes). However, the link between a CVS/SVN file and a Bugzilla/Jira problem report is not formally defined: We use pattern matching to extract a list of bug id candidates (Fischer et al. 2003; Zimmermann et al. 2007). Then, for each bug id, we check whether a bug with such an id exists in the bug database and, if so, we retrieve it. Finally we verify the consistency of timestamps, i.e., we reject any bug whose report date is after the commit date.

Due to the file-based nature of SVN and CVS and to the fact that Java inner classes are defined in the same file as their containing class, several classes might point to the same CVS/SVN file, i.e., a bug linking to a file version might be linking to more than one class. We are not aware of a workaround for this problem, which in fact is a shortcoming of the versioning system. For this reason, we do *not* consider inner classes, i.e., they are excluded from our experiments. We also filter out test classes[5] from our dataset.

Computing Metrics   At this point, we have a model including source code information over several versions, change history, and defects data. The last step is to enrich it with the metrics we want to evaluate. We describe the metrics as they are introduced with each approach.

Tools   To create our dataset, we use the following tools:

– *inFusion*[6] (developed by the company intooitus in Java) to convert Java source code to FAMIX models.
– *Moose*[7] (Ducasse et al. 2005) (developed in Smalltalk) to read FAMIX models and to compute a number of source code metrics.
– *Churrasco*[8] (D'Ambros and Lanza 2010) (developed in Smalltalk) to create the history model, extract bug data and link classes, versioning system files and bugs.

4.2 Evaluating the Approaches

We evaluate the performance of bug prediction approaches with several strategies, each according to a different usage scenario of bug prediction. We evaluate each

---

[5]We employ JUnit 3 naming conventions to detect test classes, i.e., classes whose names end with "Test" are detected as tests.

[6]Available at http://www.intooitus.com/.

[7]Available at http://www.moosetechnology.org.

[8]Available at http://churrasco.inf.usi.ch.

technique in the context of classification (defective/non-defective), ranking (most defective to least defective), and effort-aware ranking (most defect dense to least defect dense). In each case, we use performance evaluation metrics recommended by Jiang et al. (2008).

Classification   The first scenario in which bug prediction is used is classification: One is interested in a binary partition of the classes of the system in defective and non-defective classes. Since our prediction models assign probabilities to classes, we need to convert these probabilities to binary labels. The commonly used evaluation metrics in this case, namely precision and recall, are sensitive to the thresholds used as cutoff parameters. As an alternative metric we use the Receiver Operating Characteristic (ROC) curve, which plots the classes correctly classified as defective (true positives) against the classes incorrectly classified as defective (false positives). Figure 7 shows an example ROC curve, evaluating the performance of the BUGFIX approach on the Eclipse system. The diagonal represents the expected performance of a random classifier.

   To have a comprehensive measure that eases comparison across approaches, we report the Area Under the ROC Curve (AUC), as a single scalar value: an area of 1 represents a perfect classifier, whereas for a random classifier an area of 0.5 would be expected. Of course we expect all our approaches to perform better than a random classifier, but how much so remains yet to be determined.

Ranking   A scenario that is more useful in practice is to rank the classes by the predicted number of defects they will exhibit. The general concept is known as the Module-Order Model (Khoshgoftaar and Allen 1999). In the context of defect prediction, the prediction model outputs a list of classes, ranked by the predicted

**Fig. 7** ROC curve for the BUGFIX prediction approach on Eclipse
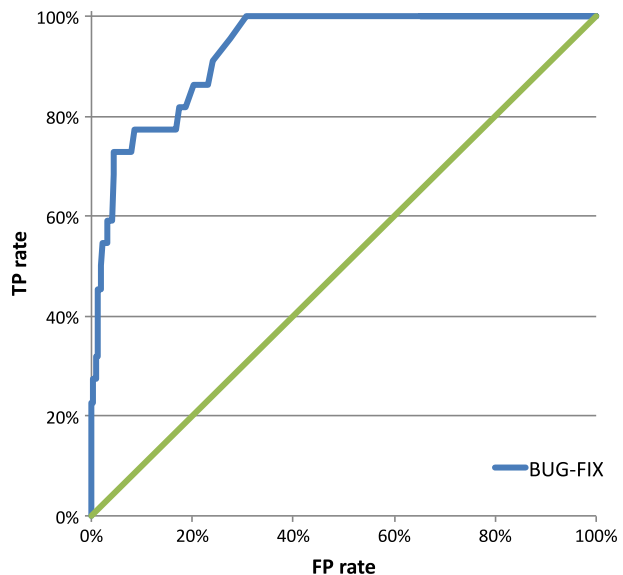
number of defects they will have. One way to measure the performance of the prediction model—used in our previous experiment (D'Ambros et al. 2010)—is to compute the Spearman's correlation coefficient between the list of classes ranked by number of predicted defects and number of actual defects. The Spearman's rank correlation test is a non-parametric test that uses ranks of sample data consisting of matched pairs. The correlation coefficient varies from 1, i.e., ranks are identical, to −1, i.e., ranks are the opposite, where 0 indicates no correlation. However, this prediction performance measure is not indicated when there is a considerable fraction of ties in the considered lists. Therefore, we employ another evaluation performance strategy, based on cumulative lift charts, which is compatible with the works of Mende and Koschke (2009) and Kamei et al. (2010). Further, cumulative lift charts are easily used in a practical setting.

Given the list of classes ranked by the predicted number of defects, a manager is expected to focus resources on as many items in the beginning of the list as possible. The question to answer in this context is: what is the percentage of defects that can be encountered, when reviewing only $n\%$ of the classes. This can be shown visually via a cumulative lift chart, where the classes are ordered according to the prediction model on the *x*-axis, and the cumulative number of actual defects is plotted on the *y*-axis, as shown in Fig. 8, for the same system and defect predictor as above.

In the chart, the bottom curve represents the expected performance of a random classifier; the middle curve, the performance of the BUGFIX approach; and the top curve, the performance of a perfect classifier, delivering a perfect ranking. The chart tells us that upon inspecting for instance 20% of the files of Eclipse using BUGFIX as a predictor, one can expect to encounter roughly 75% of the bugs. As an aside, this chart shows that roughly 20% of the classes present post-release defects, while the other 80% of the classes are defect-free.

Fig. 8 Cumulative lift chart for the BUGFIX prediction approach on Eclipse
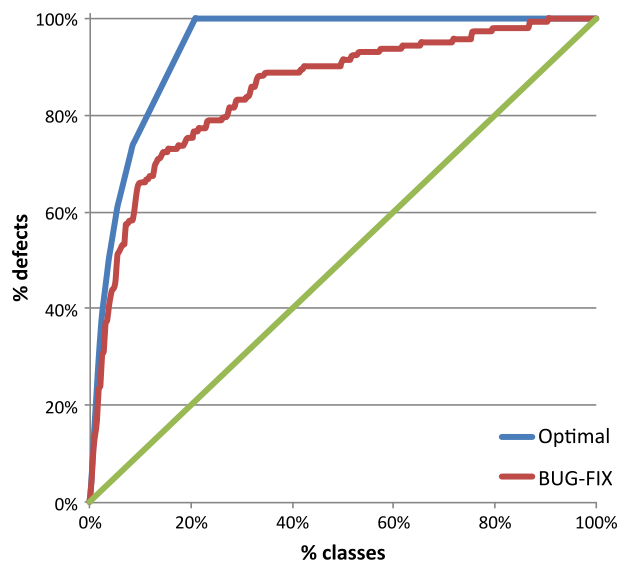
In order to extract a comprehensive performance metric from the lift chart, we use Mende and Koschke's p_{opt} metric (Mende and Koschke 2009). Δ_{opt} is defined as the difference between the area under the curve of the optimal classifier and the area under the curve of the prediction model. To keep the intuitive property that higher values denote better performance, $p_{opt} = 1 - \Delta_{opt}$.

Effort-Aware Ranking   In order to take the effort needed to review a file in account, we use the LOC metric as a proxy for effort, similarly to Mende and Koschke (2009), Mende (2010), Kamei et al. (2010) and Menzies et al. (2010). The intuition is that a larger file takes a longer time to review than a smaller file, hence one should prioritize smaller files if the number of predicted defects is the same. In this case, we evaluate the performance of the approaches with a cumulative lift chart, as done above, but ordering the classes according to their defect density (number of defects divided by number of lines of code, instead of their defect count). Additionally the chart plots LOC on the x axis instead of the number of classes. The result can be seen in Fig. 9. Compared to the previous lift chart, we see that when using BUGFIX as a predictor, inspecting the files that represent 20% of the lines of code of the system allows us to only encounter 35% of the defects, much less than the previous measure indicated. Even with an optimal ranking, we need to review 60% of the lines of code to find 100% of the defects, when taking effort into account.

In this case, we use again the p_{opt} metric to evaluate the performance of the approaches more comprehensively. In the following, we refer to it as p_{effort} to distinguish it from the previous metric.

Experimental Evaluations   We perform three experimental evaluations of all the bug prediction approaches. We start by ranking the approaches in terms of performance (Section 5); we then analyze the variability of selected approaches, and test

Fig. 9 Cumulative effort-aware lift chart for the BUGFIX prediction approach on Eclipse
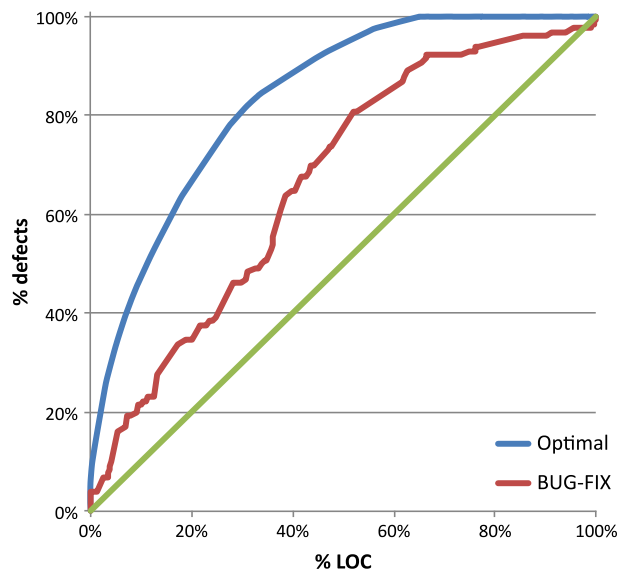
the ranking of these approaches for statistical significance (Section 6); finally, we explore the stability of the selection of attributes across several machine learning algorithms (Section 7).

5 Experiment 1: Comparing Approaches

To compare bug prediction approaches, we apply them on the same software systems and, for each system, on the same data set. We consider the last major releases of the systems and compute the predictors up to the release dates. To ease readability, in Table 4 we list all the acronyms used in the paper.

5.1 Methodology

In our first experiment, we follow the methodology detailed below and summarized in Algorithm 1:

Algorithm 1: Outline of the first experiment.

; // Let M be the set of metrics and b_{row} the number of post-release bugs
if classification **then**
     ; // if we do classification, b is a vector of boolean (with or without bugs)
     $b := b_{row} > 0$
     $A :=$ apply log transformation(M)
else
     ; // if we do ranking, we apply the log transformation also to b
     $(A, b) :=$ apply log transformation(M, b_{row})
end
$folds :=$ stratified partition(b)
; // starting from the empty set, and evaluating all attributes in A, select the best attributes to fit b doing cross validation in $folds$. To fit b, it uses generalized linear regression models
$A_{best} :=$ wrapper$(A, b, folds,$ "generalized linear regression"$)$
performance $:=$ average(crossvalidation$(A_{best}, b, folds,$ "generalized linear regression"$))$

---

1. *Preprocessing the data*. Considering the exponential nature of many of our data sets distributions, before creating any model we apply a log transformation to the data to better comply with the assumptions of linear regression models.
2. *Attribute selection*. A large number of attributes might lead to over-fitting a model, severely degrading its predictive performance on new data. Similarly, highly correlated attributes are problematic since they make it harder to judge the effect of a single attribute. To account for this, we employ an attribute selection technique called *wrapper subset evaluation*, which selects a subset of attributes that provides the best predictive performance by sequentially selecting attributes until there is no improvement in prediction.
   Starting from an empty attribute set, the wrapper creates candidate attribute subsets by sequentially adding each of the attribute as yet unselected. For each candidate attribute subset, the technique performs stratified ten-fold cross-validation by repeatedly evaluating the prediction performance with different

Table 4 List of acronyms used in the paper

Acronym	Description
MOSER	Change metrics used by Moser et al. (2008)
NFIX-ONLY	Number of times a file was involved in bug-fix (part of MOSER)
NR	Number of revisions (part of MOSER)
BUG-FIX	Number of past bug fixes
BUG-CAT	Number of past bug fixes categorized by severity and priority
CK	Chidamber & Kemerer source code metrics suite (Chidamber and Kemerer 1994)
OO	11 object-oriented code metrics (listed in Table 3)
LOC	Number of lines of code (part of OO)
HCM	Entropy of changes (Hassan 2009)
CHU	Churn of source code metrics
HH	Entropy of source code metrics
W{HCM, CHU, HH}	Weighted version of HCM or CHU or HH
ED{HCM, CHU, HH}	Exponentially decayed version of HCM or CHU or HH
LD{HCM, CHU, HH}	Linearly decayed version of HCM or CHU or HH
LGD{HCM, CHU, HH}	Logarithmically decayed version of HCM or CHU or HH

training and test subsets. In other words, for each candidate attribute subset, the wrapper creates ten prediction models and measures their performances on ten test sets (one per fold), returning the average of their prediction performance. The attribute selection technique chooses the candidate attribute subset that maximizes the performance (or minimize the prediction error): This process continues until adding more attributes does not increase the performance.

3. *Building regression models.* We base our predictions on generalized linear regression models built from the metrics we computed. The independent variables—or attributes used for the prediction—are the set of metrics under study for each class, while the dependent variable—the predicted attribute—is the number of post-release defects. Note that generalized linear regression models are also built within the attribute selection process, to evaluate the performance of each attribute subset.

4. *Ten-folds cross validation.* We do stratified ten-fold cross validation, i.e., we split the dataset in ten folds, using nine folds (90% of the classes) as training set to build the prediction model, and the remaining fold as a validation set to evaluate the accuracy of the model. Each fold is used once as a validation set. Stratified cross-validation means that the folds are selected so that the distribution of the dependent variable in each fold is consistent with the entire population.

5.2 Results

Tables 5 and 6 (top and bottom) follow the same format: Each approach is described on a row, where the first five cells show the performance of the predictor on the five subject systems, according to the metric under study. To highlight the best performing approaches, values within 90% of the best value are bolded. The last cell shows the average ranking (AR) of the predictor over the five subject systems. Values lesser or equal than 10 (top 40% of the ranking) denote good overall performance; they are underlined.

Table 5 AUC values for all systems and all predictors

Predictor	Eclipse	Mylyn	Equinox	PDE	Lucene	AR
MOSER	**0.921**	**0.864**	**0.876**	**0.853**	**0.881**	6
NFIX-ONLY	0.795	0.643	0.742	0.616	0.754	24.2
NR	0.848	0.698	0.854	0.802	0.77	19.4
NFIX+NR	0.848	0.705	0.856	0.805	0.761	19.4
BUG-CAT	**0.893**	0.747	**0.876**	**0.868**	0.814	12.2
BUG-FIX	**0.885**	0.716	**0.875**	**0.854**	0.814	13.8
CK+OO	**0.907**	**0.84**	**0.92**	**0.854**	0.764	7.2
CK	**0.905**	0.803	**0.909**	0.798	0.721	13.8
OO	**0.903**	**0.836**	**0.889**	**0.854**	0.751	10.8
LOC	**0.899**	**0.823**	0.839	0.802	0.636	17.2
HCM	0.87	0.638	0.846	0.798	0.78	20.8
WHCM	**0.906**	0.667	0.839	**0.848**	0.768	17.2
EDHCM	0.81	0.677	0.854	**0.846**	0.834	18
LDHCM	0.812	0.667	0.862	**0.849**	0.834	17
LGDHCM	0.802	0.655	0.853	0.82	0.81	20.6
CHU	**0.91**	**0.825**	0.854	**0.849**	**0.851**	10.2
WCHU	**0.913**	0.8	**0.893**	**0.851**	**0.854**	8
LDCHU	**0.89**	0.789	**0.899**	**0.87**	**0.885**	7
EDCHU	0.867	0.807	**0.892**	**0.876**	**0.884**	7.2
LGDCHU	**0.909**	0.788	**0.904**	**0.856**	**0.882**	6.2
HH	**0.91**	0.811	**0.897**	**0.843**	**0.88**	7.6
HWH	**0.897**	0.789	**0.887**	**0.853**	0.83	11.6
LDHH	**0.896**	0.806	**0.882**	**0.869**	**0.874**	9
EDHH	0.867	0.812	0.872	**0.875**	**0.879**	9.6
LGDHH	**0.907**	0.786	**0.891**	**0.845**	**0.875**	10.8

Classification with AUC Table 5 contains the prediction performance measurements according to the *AUC* metric of accuracy for classification. The last column shows the average rank of the predictor among all 25 approaches.

The best performers are **MOSER** (6), **LGDCHU** (6.2), **LDCHU** (7), **CK+OO**, and **EDCHU** (tied 7.2). After that, performance drops gradually until approaches ranked around 14, when it suddenly drops to ranks around 17. Overall, process, churn of source code, and regular source code metrics perform very well. Entropies of source code perform next (ranking from 7.6 to 11.6), followed by defect metrics. Somewhat surprisingly, entropy of change metrics perform quite badly (sometimes worse than LOC), while simple approximations of process metrics (NR, NFIX-ONLY), close the march. A possible explanation of the counter-performance of the variants of entropy metrics is that each approach amounts to a single metric, which may not have enough explaining power to correctly classify files as defective by itself. This is also the case for the defect metrics and approximations of process and source code metrics, which likewise feature one or very few metrics.

Ranking with p_{opt} We start by recalling the results of our previous experiment (D'Ambros et al. 2010), where we used the spearman correlation to evaluate the performance of the approaches. In this experiment, we found that the best overall approaches where **WCHU** and **LDHH**, followed by the previous defect approaches **BUG-FIX** and **BUG-CAT**.

Table 6 p_{opt} and p_{eff} values for all systems and all predictors

Predictor	Eclipse	Mylyn	Equinox	PDE	Lucene	AR
p_{opt}						
MOSER	**0.871**	**0.832**	**0.898**	**0.776**	**0.843**	9.4
NFIX-ONLY	0.783	0.642	0.831	0.636	0.707	23.8
NR	0.835	0.647	**0.895**	0.765	0.798	19
NFIX+NR	0.835	0.667	**0.888**	0.767	0.787	19.8
BUG-CAT	**0.865**	0.733	**0.887**	**0.8**	**0.857**	11.4
BUG-FIX	**0.865**	0.661	**0.886**	**0.79**	**0.857**	13.4
CK+OO	**0.863**	**0.808**	0.904	0.794	0.801	10
CK	**0.857**	0.763	**0.894**	0.769	0.731	17.8
OO	**0.852**	**0.808**	**0.897**	**0.79**	0.78	13.8
LOC	**0.847**	0.753	**0.881**	0.758	0.692	21.4
HCM	**0.868**	0.579	**0.893**	0.764	0.807	18.6
WHCM	**0.872**	0.624	**0.888**	**0.788**	0.804	17
EDHCM	0.823	0.625	**0.892**	0.772	**0.829**	18.4
LDHCM	0.826	0.619	**0.898**	**0.788**	0.825	17.2
LGDHCM	0.822	0.603	**0.898**	0.778	0.826	17.8
CHU	**0.873**	**0.8**	**0.883**	0.791	0.826	12.2
WCHU	**0.88**	**0.799**	**0.899**	0.793	0.828	8.2
LDCHU	**0.875**	0.789	**0.904**	0.813	0.847	5.2
EDCHU	**0.858**	**0.795**	**0.901**	0.806	0.842	8.4
LGDCHU	**0.878**	0.768	**0.908**	0.803	0.851	5.8
HH	**0.879**	**0.803**	**0.903**	0.795	0.834	6.6
HWH	**0.884**	0.769	**0.894**	**0.8**	0.808	10.4
LDHH	**0.878**	**0.792**	**0.908**	0.812	0.835	5.6
EDHH	**0.86**	**0.798**	**0.902**	0.805	0.837	8.2
LGDHH	**0.886**	0.777	**0.909**	0.796	0.845	5.2
p_{eff}						
MOSER	**0.837**	**0.787**	**0.863**	**0.748**	**0.828**	6.8
NFIX-ONLY	0.661	0.582	0.798	0.563	0.716	23
NR	0.761	0.578	**0.852**	0.711	0.766	18.2
NFIX+NR	0.757	0.609	**0.85**	0.712	0.767	17.6
BUG-CAT	**0.825**	0.685	**0.837**	**0.764**	**0.841**	10.2
BUG-FIX	**0.825**	0.598	**0.837**	0.744	**0.841**	13.8
CK+OO	**0.822**	**0.773**	**0.861**	0.751	0.753	10.4
CK	**0.816**	0.671	**0.836**	0.718	0.704	18.4
OO	0.805	**0.769**	**0.845**	**0.74**	0.741	14.8
LOC	0.801	0.67	0.793	0.701	0.614	21.4
HCM	**0.815**	0.452	**0.837**	0.69	0.769	19.8
WHCM	**0.827**	0.576	0.812	**0.749**	0.781	16.6
EDHCM	0.753	0.536	0.808	0.728	**0.81**	20.2
LDHCM	0.73	0.499	**0.85**	0.756	0.81	15.8
LGDHCM	0.627	0.453	**0.861**	0.741	0.812	16.6
CHU	**0.832**	**0.753**	**0.843**	0.75	0.812	10.8
WCHU	**0.839**	**0.76**	**0.859**	0.754	0.827	7.2
LDCHU	**0.849**	0.729	**0.85**	0.774	0.849	5.6
EDCHU	**0.832**	**0.753**	0.807	0.76	0.847	9.4
LGDCHU	**0.852**	0.652	**0.862**	0.75	0.846	7.4
HH	**0.844**	**0.77**	**0.867**	0.75	0.819	6
HWH	**0.857**	0.689	**0.835**	0.739	0.811	12.8
LDHH	**0.85**	0.738	**0.855**	0.772	0.844	5.6
EDHH	**0.828**	0.759	0.814	0.761	0.847	8.8
LGDHH	**0.857**	0.673	**0.864**	0.747	0.843	7.2

If we compare our previous results with the average rankings obtained by p_{opt} in Table 6 (top), we see some overlap: **LDHH** and **WCHU** are ranked well (5.6 and 8.2). However, defect prediction metrics have comparatively low ranks (11.4 and 13.4). The top performers are **LDCHU** and **LGDHH** (tied at 5.2), followed by most churn of source code and entropy of source code metrics. Among the nine of these set of metrics, the last one ranks at 12.2, and seven of those rank at 8.2 or below. Still performing better than the defect metrics (**BUG-CAT**), we find process metrics (**MOSER**–9.2) and source code metrics (**CK+OO**–10+). As we observed before, sets of metrics relying on few metrics—entropy of changes, cheap approximations of source code or process metrics—perform comparatively badly (ranks 17 or below).

To explain the counter-performance of defect-based metrics with respect to our previous experiment, we can posit that using a wrapper instead of PCA (as we used before) may be better suited to deal with multicollinearity in models with large amounts of attributes, which were previously disadvantaged by PCA.

Effort-Aware Ranking with p_{effort} Table 6 (bottom) sums up the performance and relative rankings of all the predictors according to the p_{effort} effort-aware metric.

In this case, the best performers are **LDCHU** and **LDHH**, both tied at rank 5.6. These metrics are good performers in both ranking scenarios. Again, most churn and entropy of source code metrics provide good predictive performance: All the ranks below ten are occupied by one of these, with the exception of the process metrics (**MOSER**) which occupy rank 6.8 (placing 4th). Product metrics (**CK+OO**) perform similarly to the effort-unaware scenario (ranked 10.4 for p_{effort} and 10 for p_{opt}).

We continue to see the general trends of approaches employing few metrics performing worse (entropies of changes, **NFIX, NR**), among which defect-related metrics perform better than the other alternatives (ranks 10.2 and 13.8).

These findings corroborate those of Mende and Koschke (2009), and of Kamei et al. (2010), which found that product metrics were outperformed by process metrics when effort was taken into account.

5.3 Statistical Comparison

In an earlier version of this work (D'Ambros et al. 2010), we compared bug prediction approaches using a scoring system we devised. However, this scoring system, while providing an indication of the performance and stability of the approaches across different software systems, did not allow us to obtain a statistically significant ranking of approaches.

The goal of this first experiment is to rank approaches –across several data sets– following a statistically rigorous methodology. To this aim, we employ the approach of Lessmann et al. (2008) and Jiang et al. (2008).

For each performance metric (AUC, p_{opt}, p_{effort}), we compute the ranking of all the classifier on each project, and from there the average rank of each classifier on all data sets. We then apply the Friedman non-parametric test (Friedman 1937) on the rankings in order to determine if the differences of performance in terms of average ranking are statistically significant (the null hypothesis being that all classifiers perform equally). The Friedman test can be calculated using the following

formulas from Demšar ([2006](#)), where k denotes the number of classifiers, N the number of data sets, and R_j the average rank of classifier j on all data sets:

$$\chi_F^2 = \frac{12N}{k(k+1)}\Big(\sum_j R_j^2 - \frac{k(k+1)^2}{4}\Big) \qquad (22)$$

$$F_F = \frac{(N-1)\chi_F^2}{N(k-1) - \chi_F^2} \qquad (23)$$

F_F is distributed according to the F-distribution with $k - 1$ and $(k - 1)(N - 1)$ degrees of freedom. Once computed, we check F_F against critical values of the F-distribution and accept or reject the null hypothesis.

When the test succeeds, we apply the Nemenyi's post-hoc test to check if the performance of each pair of classifier is significantly different. The Nemenyi's test computes a critical difference in the average ranking that two classifiers must have in order to be deemed to perform significantly different. The critical difference can be computed with the following formula:

$$CD = q_\alpha \sqrt{\frac{k(k+1)}{6N}} \qquad (24)$$

where k is the number of classifiers, N represents the number of data sets, and q_α is a critical value for the Nemenyi's test that depends on the number of classifiers and the significance level α. In our experiments, $k = 25$, $\alpha = 0.05$ and $q_{0.05} = 3.658$. Note that the Nemenyi's test is known to be conservative, hence achieving significance is difficult.

Comparing AUC   The Friedman test rejected the null hypothesis that all predictors performed equally. However, the Nemenyi's post-hoc test concluded that it could only distinguish the best performers **MOSER** and **LGD-CHU**, from the worst **NFIX-ONLY**, so our findings on the relative ranking have less support.

Figure [10](#) presents the results of Nemenyi's test, using Lessmann et al.'s ([2008](#)) modified version of Demšar's significance diagrams (Demšar [2006](#)): For each classifier on the y-axis, we plot the average rank on the x-axis, together with a line segment whose length encodes the critical difference CD. All classifiers that do not overlap in this plot perform significantly different.

Comparing p_{opt} and p_{effort}   For p_{opt} and p_{effort} the results are similar to the ones of AUC: While the Friedman test rejected the null hypothesis, Nemenyi's test was only able to separate the best performers from the worst ones. Concerning p_{opt}, **LGDHH**, **LDCHU**, **LDHH**, and **LGDCHU** are statistically better than **NFIX-ONLY**; in the case of p_{effort}, **LDHH** and **LDCHU** do not overlap with **NFIX-ONLY**. We do not show Demšar's significance diagrams, as they are very similar to the AUC one (Fig. [10](#)).

NFIX-ONLY is clearly the worst approach for all evaluation criteria.

5.4 Discussion

General Performance   Each task yields a distinct ordering of the approaches, showing that each problem has differing characteristics. If churn of source code and
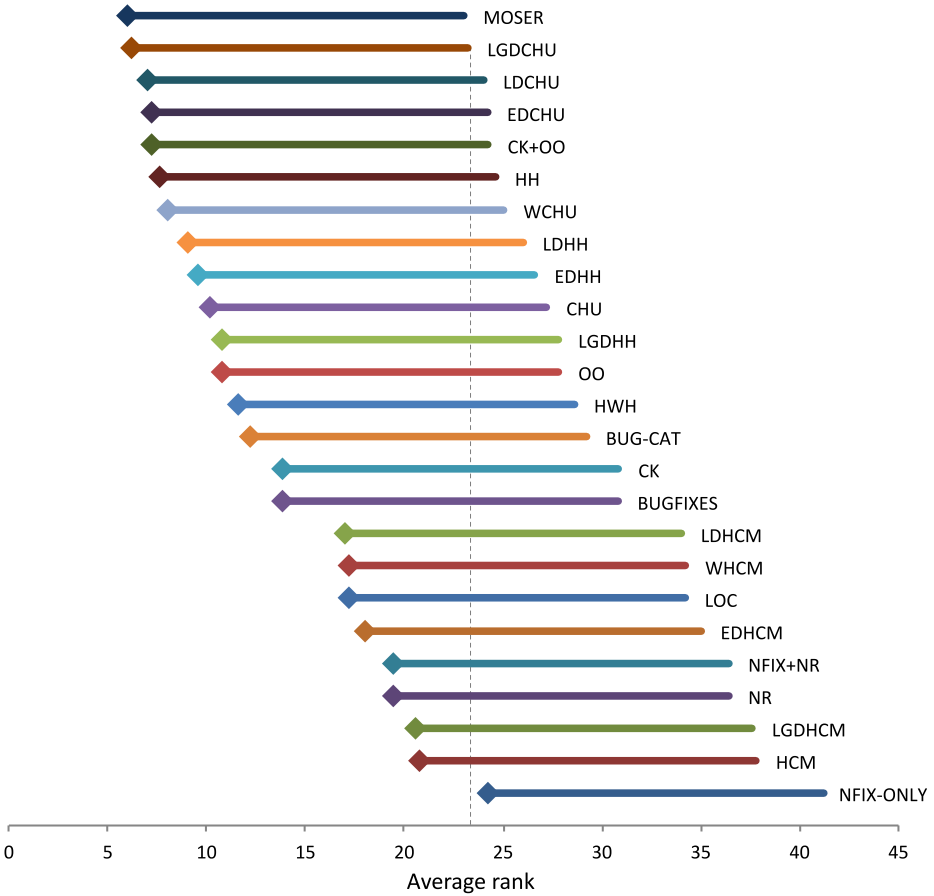
Fig. 10 Nemenyi's critical-difference diagram for AUC

entropy of source code metrics are the best performers for ranking classes (especially **LDCHU** and **LDHH**), they are comparatively less performant for classification, where process metrics outperform these, and product metrics perform very well. Overall, process metrics are a good performer, as their worse rank overall is 9.4 for ranking with p_{opt}. On the other hand, defect metrics alone did not exhibit good performance, and other single-metric predictors (entropy of changes, lines of code, number of revisions) performed very poorly as well.

> *The best performers are churn of source code and entropy of source code metrics for ranking. They are also good performers for classification, but are outperformed by process metrics, and exhibit comparable performance with product metrics.*

Limitations of Nemenyi's Test Similarly to Mende and Koschke (2009), and to Lessmann et al. (2008), we found that the Nemenyi's post-hoc test was not powerful

enough to reliably differentiate the performance of the approaches in terms of relative ranking. More subject systems are needed in order to reduce the critical difference in the rankings; this would also increase the generalizability of the results. Another option is to employ a different and more powerful testing procedure.

Another problem of Nemenyi's test is that it does not take the actual performance of the approaches into account, but only their relative ranking. In fact, looking for example at the results of classification for PDE (Table 5), we observe that most of the AUC values do not differ much—ranging from 0.84 to 0.87—so the ranking among them is weak.

Because of these reasons, we devised and ran a second set of experiments, presented next.

6 Experiment 2: Finding Statistical Significance

Our second set of experiments—inspired by the work of Menzies et al. (2010)—aims at evaluating the variability of the approaches accross several runs, and at finding a statistically significant ranking of the approaches. The main difference between these experiments and the previous ones does not reside in the experiments themselves, but in the way we analyze the results. The only difference in the experiments is that instead of averaging the results for each fold of the cross validation (last line of Algorithm 1), we save the ten results. Moreover, we repeat each experiment ten times (with distinct stratified folds), ending up with 100 data points—ten runs times ten folds—for each bug prediction approach, and for each performance measure (AUC, p_{opt}, p_{effort}). We thus end up with an approximation of the expected variability of the performance for each approach.

To analyze the results we compute the median, first, and third quartile of each set of 100 data points, as Menzies et al. (2010) did. Then, we sort the approaches by their medians, and we display them visually by means of a "mini boxplot": A bar indicates the first–third quartile range; a circle the median. In contrast with normal boxplots, the minimum and maximum values are not displayed. Figure 11 shows the
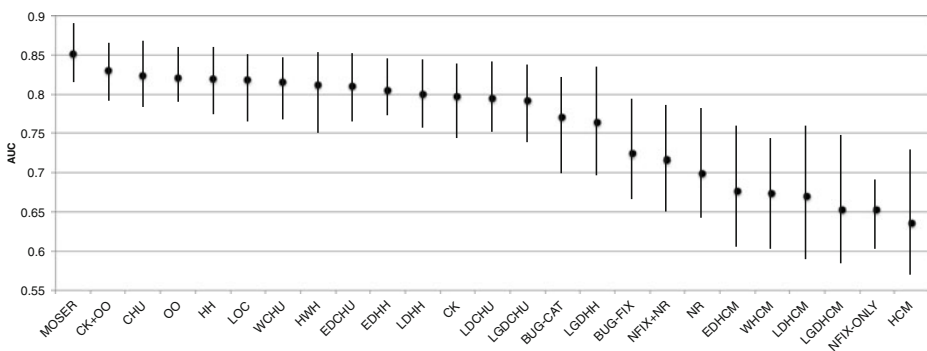

Fig. 11 Mini boxplot for classification in Mylyn, in terms of AUC. The bars represent the first–third quartile range, while the circles indicate the medians

mini boxplot of the results for classification in Mylyn (in terms of AUC), sorted by median. We can see that on this figure, as the performance degrades (lowering median), the variability of the performance of the approaches increases (with the exception of NFIX-ONLY).

However, since our goal is to compare approaches across different systems, instead of reporting the results for each system, we combine them: For each performance measure, we merge the 100 data points that we obtained for each system, resulting in a set of 500 data points that represents all the five systems. Then, as for individual systems, we compute the median, first, and third quartile and we create the mini boxplot. Figure 12 shows the mini boxplot for all systems, for classification.

Additionally, Menzies et al. (2010) create a ranking by performing the Mann–Whitney U test on each consecutive pair of approaches. Two approaches have the same rank if the U test fails to reject the null hypothesis—that the two distributions of performance values are equal—at the 95% confidence level. To have a different rank with other approaches, the test must be successful on all the other approaches of equivalent rank. This procedure is carried on in an ordered fashion, starting with the top two approaches. Menzies et al. (2010) advocate for the usage of the Mann–Whitney test, as it is non parametric and—unlike, for instance, the t-test—does not make assumptions on the distribution of the data.

For example, with respect to Fig. 12, MOSER and LDHH would be ranked 1 and 2 if the test reject the null hypothesis, otherwise they would be both ranked 1.

As shown in Fig. 12, in our experiments, obtaining a meaningful ranking is difficult, since many approaches have similar performances. For this reason, we adapted the technique proposed by Menzies et al. (2010) to better fit our comparison goal: We are more interested in the usefulness of the data sources and categories of approaches (process metrics, product metrics, defect metrics, entropy metrics, churn of source code metrics, entropy of source code metrics), than the performance of individual approaches.

Therefore, for each category of approaches, we select its best representative (the one with the highest median), filtering the others out. For instance, in the case of classification, we choose one candidate for the entropy of source code metrics category; we only select LDHH, filtering out HH, HWH, EDHH and LGDHH (see
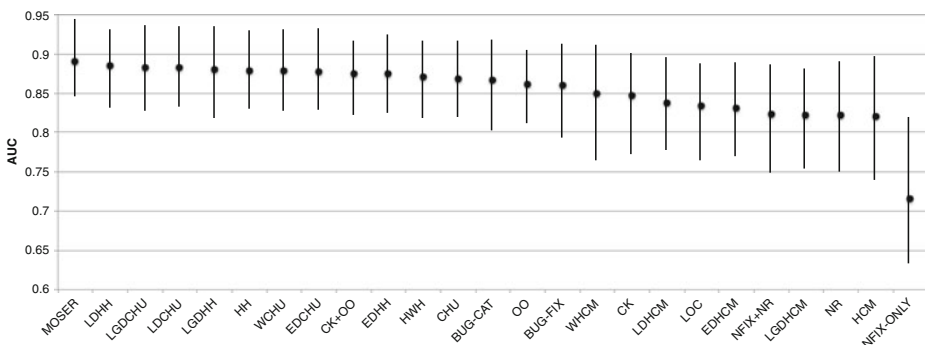

Fig. 12 Mini boxplot for classification in all software systems combined

Fig. 12). This selection greatly simplifies the comparison, as we have six categories of approaches instead of 25 individual approaches.

Then, we create a mini boxplot for the categories, i.e., for the best performing approaches of every category.

Finally, to investigate which approaches are better than others, we perform the Mann–Whitney test (at 95% confidence level) on each possible pair of approaches, and discuss the ranking.

Classification with AUC Figure 13 shows the classification results (in terms of *AUC*) for all systems, including the medians, mini boxplots, and for each pair of approaches, whether the Mann–Whitney test can reject the null hypothesis. The median of all the selected approaches range from 0.849 to 0.891.

The outcomes of the *U* test indicate that the approaches based on process metrics (**MOSER**–rank 1), entropy of code metrics (**LDHH**–rank 1), and churn of code metrics (**LGDCHU**–rank 1), provide better classification performances than the others, i.e., source code metrics (**CK+OO**–rank 4), previous defects (**BUG-CAT**–rank 4) and entropy of changes (**WHCM**–rank 6). Note that this difference is statistically significant at the 95% level. Another interesting fact is that the entropy of changes (WHCM) is statistically significantly worse than all the other approaches.

With respect to the variability, we see that the entropy has a much larger variation accross runs than other approaches—probably caused by its worse performance on Mylyn (see Table 5).

Ranking with p_{opt} Figure 14 shows the results for ranking, in terms of p_{opt}, for all systems. The median of all approaches ranges from 0.823 to 0.852—clearly lower than for classification. We observe that:

– The best performing approaches per category are the same as for classification.
– The order of approaches, sorted by decreasing median, is different: **MOSER** was the top-performer for classification, but its median places it in third position for ranking. However, it still occupies rank 1 (tied with **LDHH** and **LGDCHU**).
– The outcomes of the Mann–Whitney test are comparable with the ones of classification: Process metrics, entropy and churn of code metrics provide better

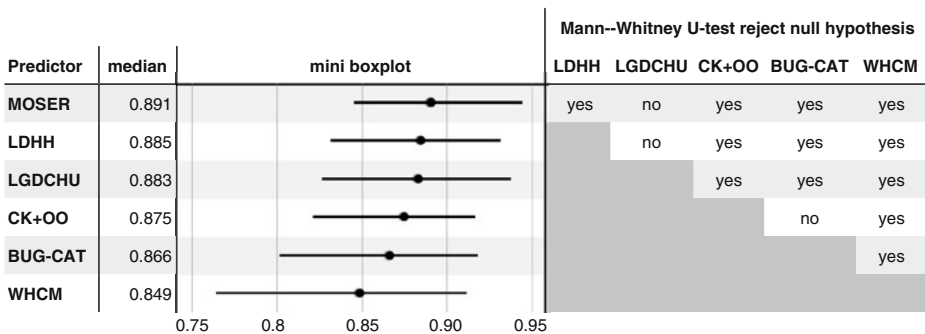| Predictor	median	mini boxplot					Mann--Whitney U-test reject null hypothesis				
							LDHH	LGDCHU	CK+OO	BUG-CAT	WHCM
MOSER	0.891						yes	no	yes	yes	yes
LDHH	0.885							no	yes	yes	yes
LGDCHU	0.883								yes	yes	yes
CK+OO	0.875									no	yes
BUG-CAT	0.866										yes
WHCM	0.849										
		0.75	0.8	0.85	0.90	0.95					

Fig. 13 Comparing the best performing approaches for classification, for all systems (in terms of *AUC*). The results include the medians, the mini boxplots, and the outcomes of the Mann–Whitney test
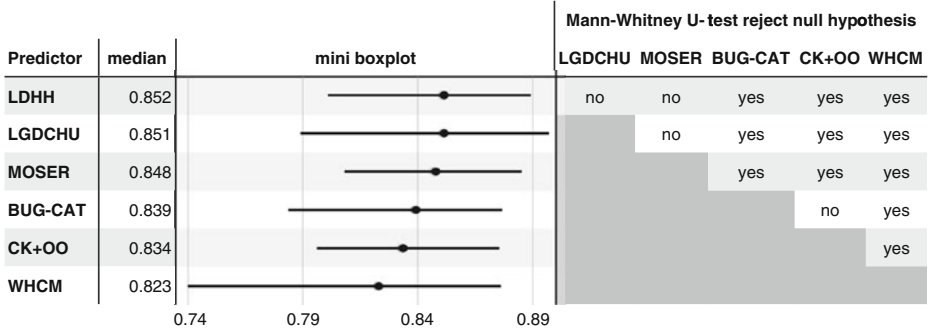
Predictor	median	mini boxplot	Mann-Whitney U- test reject null hypothesis				
			LGDCHU	MOSER	BUG-CAT	CK+OO	WHCM
LDHH	0.852		no	no	yes	yes	yes
LGDCHU	0.851			no	yes	yes	yes
MOSER	0.848				yes	yes	yes
BUG-CAT	0.839					no	yes
CK+OO	0.834						yes
WHCM	0.823						

0.74   0.79   0.84   0.89

Fig. 14 Medians, mini boxplots, and results of the Mann–Whitney test for all systems. The prediction performances are measured with p_{opt} and only the best performing approaches per category are considered

performance than the other approaches (rank 1); and the entropy of changes is worse than all the others (defect and source code metrics occupy rank 4, entropy rank 6).

– Regarding variability, we observe that **WHCM** is the most variable approach. On the other hand, **MOSER** and **CK+OO** exhibit comparatively lower variability.

Effort-Aware Ranking with p_{effort} The results for effort-aware ranking, measured with p_{effort}, are presented in Fig. 15 for all systems. The median performance of selected approaches ranges from 0.78 to 0.819, showing that this task is harder than classification or regular ranking.

We notice that, apart from **LGDHH** that substitutes **LDHH**, all the other best performing approaches per category are the same as for ranking with p_{opt} and classification. Ordering the approaches by the median, we see that churn of source code and entropy of source code switch places (albeit by a hair—0.819 vs 0.817). The other approaches keep their ranks.
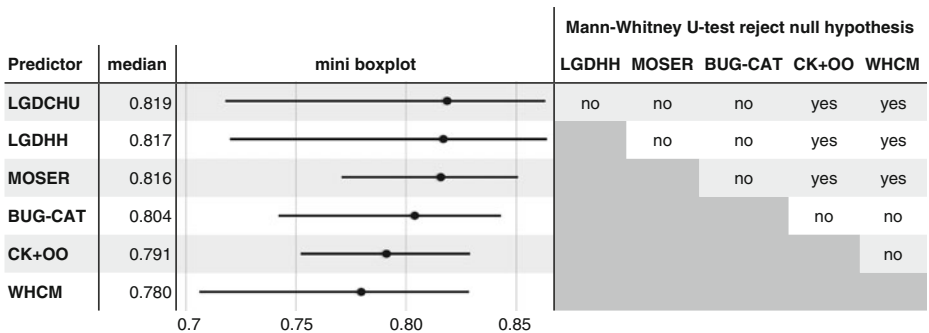
Predictor	median	mini boxplot	Mann-Whitney U-test reject null hypothesis				
			LGDHH	MOSER	BUG-CAT	CK+OO	WHCM
LGDCHU	0.819		no	no	no	yes	yes
LGDHH	0.817			no	no	yes	yes
MOSER	0.816				no	yes	yes
BUG-CAT	0.804					no	no
CK+OO	0.791						no
WHCM	0.780						

0.7   0.75   0.80   0.85

Fig. 15 Results of the effort-aware ranking for all systems, in terms of medians, mini boxplots, and outcomes of the Mann–Whitney test. The performances are measured with p_{effort}

However, the conclusions of the Mann–Whitney *U* test are more conservative: Process metrics, entropy and churn of code metrics are better than code metrics and entropy of changes, but not better than previous defects. On the same line, the entropy of changes is worse than the best three approaches, but not worse than previous defects and code metrics.

In general, for effort-aware ranking it is more difficult to demonstrate that one approach is better than another, as witnessed by the many failures of the (admittedly conservative) Mann–Whitney test.

We notice an increase of variability for **LGDCHU** and **LGDHH**, which exhibit variability even greater than **WHCM**. On the other hand, **MOSER** and **CK+OO** have a smaller amount of variability. The greater variability among top performers may be a reason for the failures of some of the *U* tests.

6.1 Discussion

These results confirm our previous observations, as far as overall performance goes: For classification, process metrics, entropy of source code and churn of source code are the best performers. They are however undistinguishable from a statistically significant point of view. The situation repeats itself for ranking with p_{opt}. For p_{effort}, these approaches are joined in the first rank by defect metrics, even if the visual difference appears larger.

The best performers overall are process, entropy of source code, and churn of source code metrics.

Unsurprisingly, the simpler problem of classification is the one where performance is highest, and variability lowest. We note that effort-aware ranking is a harder problem, with lower median performance and higher variability. We also note that across evaluation types, process and product metrics feature a lower variability of performance, a factor that makes process metrics the preferred metric set when considering both performance and stability. The entropy metric, on the other hand, has a much higher variability in performance.

Process and product metrics exhibit the lowest level of variability across all tasks.

7 Experiment 3: The Revenge of Code Metrics

Several works have shown that performance in prediction can vary wildly among predictors. The "No Free Lunch" theorem (Ho and Pepyne 2002) states that if ones does not have specific knowledge about a particular problem domain, then every approach to optimization problems will have the same average performance on the set of all inputs—even random approaches. This result explains why some studies have encountered contradicting conclusions, when run on other datasets, or with other learning algorithms, such as the opposing conclusions of Gyimóthy et al. (2005), and of Fenton and Ohlsson (2000).

This raises the question of whether the results we outlined above are valid with other machine learning approaches. To account for this possible strong threat to external validity, we once again took inspiration in the study of Hall and Holmes on attribute selection (Hall and Holmes 2003). Instead of reporting performance,

we focus on the attribute selection itself: Do different learning algorithms select the same attributes in the feature selection phase? Attributes that are consistently selected are an indicator of stability of their performance across learners.

We therefore replicated the classification task with two additional learners: Decision trees (DT) and Naïve bayes (NB)—in addition to generalized linear logistic models (GLM). We opted for these two algorithms, as they represent two quite different approaches to learning and are state-of-the-art algorithms that are often used in data mining applications. We only replicated the classification task, as the two machine learning algorithms above are inherently classifiers (giving a label to instances, such as "Defective" and "Non-defective"), and as such are not optimal for ranking.

More than attributes, we are interested in categories of attributes, as the data sources available for each project may vary. All the projects in our benchmark dataset are in the ideal situation: presence of version control data, defect data, source code metrics, bi-weekly snapshots of said metrics. Other projects may not have this luxury. As such, evaluating the stability of attribute selection should also take into account the provenance of each attribute.

The attribute selection experiments are structured as follows:

– First, we combine all the attributes of all the approaches in a single set. For example, in the case of process metrics we take all the metrics listed in Table 2, for code metrics all the ones listed in Table 3, etc.
– Then, we employ an attribute selection technique to select the attributes which yield to the model with the best predictive performance (in terms of AUC). According to which learner we are testing, the model can be a GLM, a DT, or a NB classifier. We again use wrapper subset evaluation. The attribute selection is performed with stratified ten-fold cross validation, i.e., the attributes selected are the ones which provide the best results across the ten folds.
– Finally, instead of saving the performance results, we keep track of which attributes were selected, and map them back to the approaches they belong to (e.g., lines of code belongs to code metrics, number of revision to process metrics).
– As in experiment 2, we run each experiment ten times, obtaining ten sets of attributes per software system, for each learner (for a total of 150 sets—5 systems times 3 learners times 10 sets).

Since the wrapper subset evaluation is computationally expensive, and we have a total of 212 attributes, we reduce the complexity by removing some attributes. Every variation of entropy and churn of code metrics includes 17 attributes, as we consider 17 code metrics. Since these attributes largely overlap,[9] we decided to consider only the best performing variation, as we previously did in experiment 2. As indicated in Fig. 13, the best performing variations for classification are the linearly decayed one for entropy (LDHH) and the logarithmically decayed one for churn (LGDCHU). This leaves us with 78 attributes in total.

Table 7 shows a sample result from Equinox, using decision trees. Every column is a run of the experiment, and every row a category of prediction approaches.

---

[9]For instance, the churn of FanIn linearly decayed and churn of FanIn logarithmically decayed have a very high correlation.

Table 7 Results of ten runs of attribute selection for Equinox, with decision trees

Category of approaches	Run										Count
	1	2	3	4	5	6	7	8	9	10	
Process metrics (MOSER)	0	0	1	0	1	0	0	1	1	1	**5**
Previous defects (BUG-CAT)	1	0	1	0	1	1	0	2	0	2	**6**
Entropy of changes (HCM, WHCM, EDHCM, LDHCM, LGDHCM)	0	0	0	0	0	0	0	0	0	0	**0**
Code metrics (CK+OO)	2	1	1	1	2	1	2	1	1	4	**10**
Churn of code metrics (LGDCHU)	1	0	2	0	2	1	1	0	0	1	**6**
Entropy of code metrics (LDHH)	0	1	1	1	4	1	2	2	0	3	**8**

The value of a cell indicates the number of attributes selected in the corresponding run, belonging to the corresponding prediction category. The rightmost column reports how many runs had one (or more) attribute from the corresponding category selected—10 would mean always; 0, never. In some cases (process metrics, previous defects and code metrics), the category of approaches corresponds to a single approach. This is because the other approaches in the same category are subsets of the general one (e.g., **LOC** or **CK** versus **CK+OO**).

Table 7 shows us that in the case of decision trees and Equinox, at least one code metric is selected at every experiment; in contrast, entropy of change metrics are never selected. Concerning the other categories, attributes belonging to them are selected in a number of runs ranging from five to eight.

To compare prediction approaches, we evaluate the results for all the subject software systems; we compute the mean and the variance of the selection counts (e.g., last column of Table 7) across the five systems, i.e., across 50 sets of attributes (ten runs times five systems). Table 8 and Fig. 16 show the mean and variance values for the three learners: logistic regression, decision trees, and Naïve Bayes.

7.1 Discussion

Relation to Previous Performance The fact that a prediction approach provides better (in a statistically significant manner) predictive performance than another one, when studied in isolation, does not imply that attributes belonging to the better approach are preferred to attributes from the worse one. According to experiment 2, code metrics for AUC are ranked 4th using a GLM; yet, at least one of them is

Table 8 Mean and variance values of the attribute selection counts across the five software systems, for the three learners

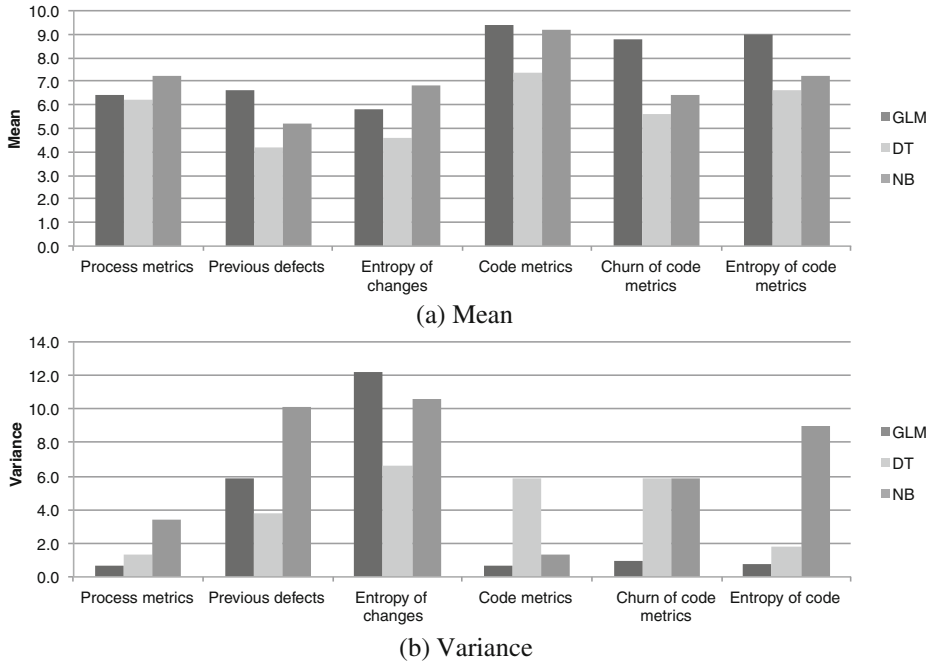| Category of approach	GLM		DT		NB	
	Mean	Var	Mean	Var	Mean	Var
Process metrics (MOSER)	6.4	0.64	6.2	1.36	7.2	3.44
Previous defects (BUG-CAT)	6.6	5.84	4.2	3.76	5.2	10.16
Entropy of changes (HCM, WHCM, EDHCM, LDHCM, LGDHCM)	5.8	12.16	4.6	6.64	6.8	10.56
Code metrics (CK+OO)	9.4	0.64	7.4	5.84	9.2	1.36
Churn of code metrics (LGDCHU)	8.8	0.96	5.6	5.84	6.4	5.84
Entropy of code metrics (LDHH)	9.0	0.81	6.6	1.84	7.2	8.96

(a) Mean

(b) Variance

Fig. 16 Histograms of the attribute selection counts across the five software systems, for the three learners

selected on average more than nine times out of ten, with a very low variance. This is actually the best performance. Clearly, at least some of the attributes do encode information very relevant to the learner.

> *The performance of a set of attributes taken in isolation does not imply their selection when compared to other—worse performing in isolation—attributes.*

Stability Across Learners Different learners select different attributes, with the exception of code metrics: These are frequently selected—and with comparatively low variance to boot—across all learners. We named this section "the revenge of code metrics" for this reason: The previous sets of experiments found that other techniques performed better, whereas this set of experiments shows that they do encode very relevant information most of the time.

Entropy of code metrics is the second most selected across different learners, but for Naïve Bayes it exhibits very high variance, leading us to believe that the information it contains varies across systems for this classifier.

Process and code metrics exhibit the lowest variability, as in the previous experiments; on the other hand, entropy of changes exhibits the highest variability—again. This confirms its status of an erratic performer. Previous defects has also a high variability for logistic regression and Naïve Bayes, and is still quite variable for decision trees; it also had a degree of variability in the previous experiments. These two approaches are also selected somewhat less often than the others. However, the

trend is difficult to discern; more data—additional systems and learners—is needed to confirm or infirm this trend.

> *There is a degree of similarity between the variability in performance in the previous experiments, and the variance in selection in these experiments; more data is needed.*

Variance in Each Learner   Decision trees provide the most balanced results, i.e., the range of variance values across different approaches is more compact. In contrast, Naïve Bayes has a very high variance for half of the categories of approaches. Were it not for entropy of changes, logistic regression would perform better than decision trees in this regards, as four of the six categories of approaches have a very low variance in attribute selection. Obviously, adding more learning algorithms would strengthen our initial observations.

> *Different learners have different attribute selection strategies; some are more stable than others.*

8 Lessons Learned

In performing a number of bug prediction experiments, with a variety of case studies, evaluation criteria and experimental setups, we learned several lessons that we summarize in the following.

On Performance in Isolation   Even with the same learner (GLM) different experiments provide somewhat different results. In experiment 2, we found that process metrics, churn and entropy of code metrics are statistically better (with Mann–Whitney test at 95% level) than code metrics, when studied in isolation. In experiment 3, we discovered that some code metrics are selected most often, and with low variability. Metrics evaluated in isolation and metrics evaluated alongside larger sets of attributes have different behaviors.

On Performance Across Learners and Case Studies   When given the choice, different learners tend to select very different metrics. Some metrics are chosen in irregular patterns even by the same learner, but on different projects. In other words, the "No Free Lunch" theorem (Ho and Pepyne 2002) is in full effect: Based on the data at our disposal, whether a metric or set of metrics performs well in isolation, in a given project, or with a given learner, does not necessarily indicate that it will be selected for learning in another context. We did however see a general trend between best performers in the previous experiments, and both a higher probability to be selected across runs, with a higher stability across projects. Nevertheless, the evidence is very slight and needs to be investigated much further.

In short, our current study did not yet demonstrate stability in the attributes selected across five systems and three learners. Previous studies that advocated the absence of stability in the domains of effort estimation (Foss et al. 2003; Myrtveit et al. 2005) and defect prediction (Menzies et al. 2007) are still unchallenged.

This raises a serious threat to the external validity of any bug prediction study. Beyond very broad guidelines, it is not possible to generalize results based on a limited number of data points (these being either data sources, case studies, or

learning algorithms). Finding general rules appears to be extremely challenging. This leaves us with two options: (1) Gather more data points (increasing the number of case study systems to refine our observations, and increasing the number of learning algorithms); or (2) use more domain knowledge to tailor the technique to the software system and the exact prediction task to be performed.

On the Granularity of Information The churn of source code metrics can be seen as process metrics with a finer grained level of detail, with respect to **MOSER**, as 17 different code metrics are considered. The results of our experiments, especially experiment 2, indicate that the churn of code metrics does not significantly outperform **MOSER** in every prediction task, i.e., classification, ranking and effort-aware ranking.

We conclude that—with process metrics—a finer grained level of information does not improve the performance of the prediction in an observable manner.

On PCA vs Wrapper A number of attribute selection techniques were proposed, such as Relief (Kira and Rendell 1992; Kononenko 1994), principal components analysis (PCA) (Jackson 2003), CFS (Correlation-based Feature Selection) (Hall 2000), and wrapper subset evaluation (Kohavi and John 1997). In our earlier work (D'Ambros et al. 2010) we used PCA, as was also proposed in Nagappan et al. (2006). However, according to a study of Hall and Holmes (2003), who performed an extensive comparison of attribute selection techniques, PCA is one of the worst performers. They instead recommend wrapper subset evaluation, which might not scale to a large number of attributes. In all our experiments we used wrapper subset evaluation, and found it to scale to our problems (up to 78 attributes in experiment 3).

Additionally, we compared the performance of PCA and wrapper subset evaluation. We ran experiment 1 with both techniques (on identical folds), and found wrapper subset evaluation to be better than PCA—statistically—for AUC, p_{effort} (but not p_{opt}), confirming the finding of Hall and Holmes:

– For classification, the maximum performance improvement was +24.77%; the minimum −13.06%; and the average, +3,06%. The Mann–Whitney U test found that the difference between distributions was statistically significant at the 99% level ($p < 0.01$).
– For ranking with p_{opt}, the range was from −5.05% (min) to +6.15% (max), with average of +0.05%. This was not found to be statistically significant.
– Finally, for ranking with p_{effort}, the range of differences was from −33.22% (min) to +18.22% (max), with average +1,08%. The U test found that the difference in distribution was significant at 95%, with $p < 0.02$.

Comparing the individual results, we generated the same tables that we reported in experiment 1 (Tables 5 and 6) for PCA (omitted for brevity). We observed that the improvement was not constant: Larger attribute sets (e.g., process metrics, product metrics) benefitted more from wrapper subset evaluation than smaller attribute sets (e.g., previous defect and entropy of changes). This might be the reason why some of the results we report here are different from the ones of our previous work (D'Ambros et al. 2010), in which previous defects performed much better than in these experiments.

All in all, we recommend the use of wrapper subset evaluation instead of principal component analysis for future defect prediction experiments, or—even better—using both and comparing the results.

On the Evaluation of the Approaches Data-Wise   If we take into account the amount of data and computational power needed for the churn and entropy of code metrics, one might argue that downloading and parsing dozens of versions of the source code is a costly process. It took several days to download, parse and extract the metrics for about ninety versions of each software system.

A lightweight approach that is an all-around good performer is **MOSER**, which also exhibits relatively low variability. However, approaches based on process metrics have limited usability, as the history of the system is needed, which might be inaccessible or, for newly developed systems, not even existent. This problem does not hold for the source code metrics **CK+OO**, as only the last version of the system is necessary to extract them.

On the Approximations of Metrics Sets   In experiment 1, we tried an approximation of source code metrics—**LOC**— and two approximations of change metrics—**NR** and **NFIX-ONLY**. We adopted these metrics in isolation, as according to previous studies (Graves et al. 2000; Gyimóthy et al. 2005; Zimmermann et al. 2007) they exhibit a very high correlation with post-release defects. However, our results indicate quite low prediction performances for these isolated metrics:

– **LOC**. It features an average ranking ranging from 17.2 for classification (see Table 5) to 21.4 for ranking and effort-aware ranking (see Table 6). This is in contrast with the average ranking of **CK+OO**, which ranges from 7.2 (classification) to 10.4 (effort-aware ranking).
– **NR**. The number of revisions **NR** is simpler to compute than the entire set of process metrics **MOSER** extracted from the versioning system. However, this approximation yields poor predictive power, as witnessed by the average ranking of **NR**, varying from 18.2 for effort-aware ranking, to 19 for ranking, to 19.4 for classification (see Tables 5 and 6).
– **NFIX-ONLY**. Prediction models based on this metric consistently provided the worst performances across all evaluation criteria. In fact, this approach was also discarded by Nemenyi's test (see Fig. 10)—it was the only one.
  If we compare the performance of **NFIX-ONLY** against **BUGFIXES**, we see that the heuristic searching bugs from commit comments is a very poor approximation of actual past defects: **BUGFIXES** has an average ranking ranging from 13.4 (ranking) to 13.8 (classification and effort-aware ranking), whereas **NFIX-ONLY** ranges from 23 (effort-aware ranking) to 24.2 (classification).
  On the other hand, there is no significant improvement in **BUG-CAT** with respect to **BUGFIXES**, meaning that categorizing bugs does not improve the predictive performance with respect to just counting them.

As discussed in the description of experiment 1 (see Section 5.2), a possible reason for the low performances of the approximations is that they amount to single metrics, which might not have enough explanative power to provide an accurate prediction.

9 Threats to Validity

Threats to Construct Validity   regard the relationship between theory and observation, i.e., the measured variables may not actually measure the conceptual variable.

A first threat concerns the way we link bugs with versioning system files and subsequently with classes. In fact, all the links that do not have a bug reference in a commit comment cannot be found with our approach. Bird et al. studied this problem in bug databases (Bird et al. 2009): They observed that the set of bugs which are linked to commit comments is not a fair representation of the full population of bugs. Their analysis of several software projects showed that there is a systematic bias which threatens the effectiveness of bug prediction models. While this is certainly not a satisfactory situation, nonetheless this technique represents the state of the art in linking bugs to versioning system files (Fischer et al. 2003; Zimmermann et al. 2007).

Another threat is the noise affecting Bugzilla repositories. Antoniol et al. (2008) showed that a considerable fraction of problem reports marked as bugs in Bugzilla (according to their severity) are indeed "non bugs", i.e., problems not related to corrective maintenance. We manually inspected a statistically significant sample (107) of the Eclipse JDT Core bugs we linked to CVS files, and found that more than 97% of them were real bugs.[10] Therefore, the impact of this threat on our experiments is limited.

Threats to Statistical Conclusion Validity   concern the relationship between the treatment and the outcome.

We used Nemenyi's post-hoc test to determine if the difference in performance of each approach was significant, but the test only succeeded in separating the very best from the very worst performers. Nemenyi's test is a very conservative test, which has hence a higher chance of committing a type II error (in this case, failing to reject the null hypothesis that the approaches perform equally). For this reason, we conducted a second series of experiments, testing the differences between pair of approaches with the Mann–Whitney U test. We ran the test at 95% confidence level.

Threats to External Validity   concern the generalization of the findings.

We have applied the prediction techniques to open-source software systems only. There are certainly differences between open-source and industrial development, and in particular because some industrial settings enforce standards of code quality. We minimized this threat by using parts of Eclipse in our benchmark, a system that, while being open-source, has a strong industrial background. A second threat concerns the language: All considered software systems are written in Java. Adding non-Java systems to the benchmark would increase its value, but would introduce problems since the systems would need to be processed by different parsers, producing variable results.

The bias between the set of bugs linked to commit comments and the entire population of bugs, that we discussed above, threatens also the external validity of our approach, as results obtained on a biased dataset are less generalizable.

---

[10]This is not in contradiction with Antoniol et al. (2008): Bugs mentioned as fixes in CVS comments are intuitively more likely to be real bugs, as they got fixed.

To decrease the impact of a specific technology/tool, in our dataset we included systems developed using different versioning systems (CVS and SVN) and different bug tracking systems (Bugzilla and Jira). Moreover, the software systems in our benchmark are developed by independent development teams and emerged from the context of two unrelated communities (Eclipse and Apache).

Having said that, we argue that the threats to external validity of our experiments in particular, and of all defect prediction studies in general, are very strong and should not be underestimated. The controversial results of the experiments presented in this article, show how findings obtained on a certain software system, using a certain learning algorithm, and measured with a certain evaluation criterion, are difficult to generalize to other systems, learning algorithms, and evaluation criteria.

This supports the conclusions of Menzies et al., who argued that if a learner is tuned to a particular evaluation criterion, then this learner will do best according to that criterion (Menzies et al. 2010).

10 Conclusion

Defect prediction concerns the resource allocation problem: Having an accurate estimate of the distribution of bugs across components helps project managers to optimize the available resources by focusing on the problematic system parts. Different approaches have been proposed to predict future defects in software systems, which vary in the data sources they use, in the systems they were validated on, and in the evaluation technique employed; no baseline to compare such approaches exists.

We have introduced a benchmark to allow for a common comparison, which provides all the data needed to apply a large array of prediction techniques proposed in the literature. Our dataset, publicly available at http://bug.inf.usi.ch, allows the reproduction of the experiments reported in this paper and their comparison with novel defect prediction approaches. Further, the dataset comes with additional data beyond metrics that allows researcher to define and evaluate novel metrics, and their variations over time as churn or entropy variants.

We evaluated a selection of representative approaches from the literature, some novel approaches we introduced, and a number of variants. Our comprehensive evaluation compared the approaches according to three scenarios: Binary classification, ranking based on defects, and effort-aware ranking based on defect density. For binary classification, simple process metrics were the best overall performer, slightly ahead of churn of source code and entropy of source code metrics; a subsequent experiment found the differences between the top three not significant. On the other hand, our results showed that for both ranking strategies, the best performing techniques are churn of source code and entropy of source code metrics—even if a second experiment found the difference with process metrics not significant for regular ranking, and was unable to produce a ranking when effort was taken into account.

However, according to a third series of experiments that we performed, these results are to be taken with a (huge) grain of salt: Generalizing the results to other learners proved to be an unsuccessful endeavor, as different learners select very

different attributes. We were only able to observe a very general trend that, to be confirmed (or unconfirmed), needs to be investigated further by adding new datasets and new learning algorithms.

This raises the importance of shared datasets and benchmarking in general: Defect prediction is a field where external validity is very hard to achieve. The only way to gain certainty towards the presence/absence of external validity in defect prediction studies is to broaden our range of case studies; we welcome contributions to our benchmark dataset.

In the absence of certainty, one needs to restrain his goals, and adopt the opposite strategy: Fine-tune a defect predictor to the specific task at hand.

References

Antoniol G, Ayari K, Di Penta M, Khomh F, Guéhéneuc Y-G (2008) Is it a bug or an enhancement?: a text-based approach to classify change requests. In: Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds (CASCON 2008). ACM, New York, pp 304–318

Arisholm E, Briand LC (2006) Predicting fault-prone components in a java legacy system. In: Proceedings of the 2006 ACM/IEEE international symposium on empirical software engineering (ISESE 2006). ACM, New York, pp 8–17

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

Arnaoudova V, Eshkevari L, Oliveto R, Gueheneuc Y-G, Antoniol G (2010) Physical and conceptual identifier dispersion: measures and relation to fault proneness. In: Proceedings of the 26th IEEE international conference on software maintenance (ICSM 2010). IEEE CS, Washington, pp 1–5

Bacchelli A, D'Ambros M, Lanza M (2010) Are popular classes more defect prone? In: Proceedings of the 13th international conference on fundamental approaches to software engineering (FASE 2010). Springer, Berlin, pp 59–73

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

Bernstein A, Ekanayake J, Pinzger M (2007) Improving defect prediction using temporal features and non linear models. In: Proceedings of the ninth international workshop on principles of software evolution (IWPSE 2007). ACM, New York, pp 11–18

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

Bird C, Bachmann A, Aune E, Duffy J, Bernstein A, Filkov V, Devanbu P (2009) Fair and balanced?: bias in bug-fix datasets. In: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering (ESEC/FSE 2009). ACM, New York, pp 121–130

Briand LC, Daly JW, Wüst J (1999) A unified framework for coupling measurement in object-oriented systems. IEEE Trans Softw Eng 25(1):91–121

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

D'Ambros M, Lanza M (2010) Distributed and collaborative software evolution analysis with churrasco. J Sci Comput Program (SCP) 75(4):276–287

D'Ambros M, Lanza M, Robbes R (2010) An extensive comparison of bug prediction approaches. In: Proceedings of the 7th international working conference on mining software repositories (MSR 2010). IEEE CS, Washington, pp 31–41

Demeyer S, Tichelaar S, Ducasse S (2001) FAMIX 2.1—The FAMOOS information exchange model. Technical report, University of Bern

Demšar J (2006) Statistical comparisons of classifiers over multiple data sets. J Mach Learn Res 7:1–30

Ducasse S, Gîrba T, Nierstrasz O (2005) Moose: an agile reengineering environment. In: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on foundations of software engineering (ESEC/FSE 2005). ACM, New York, pp 99–102. Tool demo

El Emam K, Melo W, Machado JC (2001) The prediction of faulty classes using object-oriented design metrics. J Syst Softw 56(1):63–75

Fenton NE, Ohlsson N (2000) Quantitative analysis of faults and failures in a complex software system. IEEE Trans Softw Eng 26(8):797–814

Fischer M, Pinzger M, Gall H (2003) Populating a release history database from version control and bug tracking systems. In: Proceedings of the international conference on software maintenance (ICSM 2003). IEEE CS, Washington, pp 23–32

Foss T, Stensrud E, Kitchenham B, Myrtveit I (2003) A simulation study of the model evaluation criterion mmre. IEEE Trans Softw Eng 29(11):985–995

Friedman M (1937) The use of ranks to avoid the assumption of normality implicit in the analysis of variance. J Am Stat Assoc 32(200):675–701

Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting fault incidence using software change history. IEEE Trans Softw Eng 26(07):653–661

Gyimóthy 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

Hall MA (2000) Correlation-based feature selection for discrete and numeric class machine learning (ICML 2000). In: Proceedings of the seventeenth international conference on machine learning. Morgan Kaufmann, San Mateo, pp 359–366

Hall M, Holmes G (2003) Benchmarking attribute selection techniques for discrete class data mining. IEEE Trans Knowl Data Eng 15(6):1437–1447

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

Hassan AE, Holt RC (2005) The top ten list: dynamic fault prediction. In: Proceedings of the 21st IEEE international conference on software maintenance (ICSM 2005). IEEE CS, Washington, pp 263–272

Ho YC, Pepyne DL (2002) Simple explanation of the no-free-lunch theorem and its implications. J Optim Theory Appl 115(3):549–570

Jackson EJ (2003) A users guide to principal components. Wiley, New York

Jiang Y, Cukic B, Ma Y (2008) Techniques for evaluating fault prediction models. Empir Software Eng 13:561–595

Juristo NJ, Vegas S (2009) Using differences among replications of software engineering experiments to gain knowledge. In: Proceedings of the 3rd international symposium on empirical software engineering and measurement (ESEM 2009). IEEE CS, Washington, pp 356–366

Kamei Y, Matsumoto S, Monden A, Matsumoto K-i, Adams B, Hassan AE (2010) Revisiting common bug prediction findings using effort-aware models. In: Proceedings of the 26th IEEE international conference on software maintenance (ICSM 2010). IEEE CS, Washington, pp 1–10

Kim S, Zimmermann T, Whitehead J, Zeller A (2007) Predicting faults from cached history. In: Proceedings of the 29th international conference on software engineering (ICSE 2007). IEEE CS, Washington, pp 489–498

Kira K, Rendell LA (1992) A practical approach to feature selection. In: Proceedings of the ninth international workshop on machine learning (ML 1992). Morgan Kaufmann, San Mateo, pp 249–256

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

Khoshgoftaar TM, Allen EB, Goel N, Nandi A, McMullan J (1996) Detection of software modules with high debug code churn in a very large legacy system. In: Proceedings of the seventh international symposium on software reliability engineering (ISSRE 1996). IEEE CS, Washington, pp 364–371

Kohavi R, John GH (1997) Wrappers for feature subset selection. Artif Intell 97:273–324

Kollmann R, Selonen P, Stroulia E (2002) A study on the current state of the art in tool-supported UML-based static reverse engineering. In: Proceedings of the ninth working conference on reverse engineering (WCRE 2002). IEEE CS, Washington, pp 22–32

Kononenko I (1994) Estimating attributes: analysis and extensions of relief. Springer, Berlin, pp 171–182

Koru AG, Zhang D, Liu H (2007) Modeling the effect of size on defect proneness for open-source software. In: Proceedings of the third international workshop on predictor models in software engineering (PROMISE 2007). IEEE CS, Washington, pp 10–19

Koru AG, El Emam K, Zhang D, Liu H, Mathew D (2008) Theory of relative defect proneness. Empir Software Eng 13:473–498

Lessmann S, Baesens B, Mues C, Pietsch S (2008) Benchmarking classification models for software defect prediction: a proposed framework and novel findings. IEEE Trans Softw Eng 34(4):485–496

Marcus A, Poshyvanyk D, Ferenc R (2008) Using the conceptual cohesion of classes for fault prediction in object-oriented systems. IEEE Trans Softw Eng 34(2):287–300

Mende T (2010) Replication of defect prediction studies: problems, pitfalls and recommendations. In: Proceedings of the 6th international conference on predictive models in software engineering (PROMISE 2010). ACM, New York, pp 1–10

Mende T, Koschke R (2009) Revisiting the evaluation of defect prediction models. In: Proceedings of the 5th international conference on predictive models in software engineering (PROMISE 2009). ACM, New York, pp 1–10

Mende T, Koschke R (2010) Effort-aware defect prediction models. In: Proceedings of the 14th European conference on software maintenance and reengineering (CSMR 2010). IEEE CS, Washington, pp 109–118

Menzies T, Greenwald J, Frank A (2007) 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, Bener YJA (2010) Defect prediction from static code features: current results, limitations, new approaches. Autom Softw Eng 17:375–407

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 (ICSE 2008). ACM, New York, pp 181–190

Myrtveit I, Stensrud E, Shepperd MJ (2005) Reliability and validity in comparative studies of software prediction models. IEEE Trans Softw Eng 31(5):380–391

Nagappan N, Ball T (2005a) Static analysis tools as early indicators of pre-release defect density. In: Proceedings of the 27th international conference on software engineering (ICSE 2005). ACM, New York, pp 580–586

Nagappan N, Ball T (2005b) Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th international conference on software engineering (ICSE 2005). ACM, New York, pp 284–292

Nagappan N, Ball T, Zeller A (2006) Mining metrics to predict component failures. In: Proceedings of the 28th international conference on software engineering (ICSE 2006). ACM, New York, pp 452–461

Nikora AP, Munson JC (2003) Developing fault predictors for evolving software systems. In: Proceedings of the 9th international symposium on software metrics (METRICS 2003). IEEE CS, Washington, pp 338–349

Neuhaus S, Zimmermann T, Holler C, Zeller A (2007) Predicting vulnerable software components. In: Proceedings of the 14th ACM conference on computer and communications security (CCS 2007). ACM, New York, pp 529–540

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 (2002) The distribution of faults in a large industrial software system. In: Proceedings of the ACM SIGSOFT international symposium on software testing and analysis (ISSTA 2002). ACM, New York, pp 55–64

Ostrand TJ, Weyuker EJ, Bell RM (2004) Where the bugs are. In: Proceedings of the ACM SIGSOFT international symposium on software testing and analysis (ISSTA 2004). ACM, New York, pp 86–96

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

Ostrand TJ, Weyuker EJ, Bell RM (2007) Automating algorithms for the identification of fault-prone files. In: ISST proceedings of the ACM SIGSOFT international symposium on software testing and analysis (ISSTA 2007). ACM, New York, pp 219–227

Pinzger M, Nagappan N, Murphy B (2008) Can developer-module networks predict failures? In: Proceedings of the 16th ACM SIGSOFT international symposium on foundations of software engineering (FSE 2008). ACM, New York, pp 2–12

Robles G (2010) Replicating MSR: a study of the potential replicability of papers published in the mining software repositories proceedings. In: Proceedings of the 7th international working conference on mining software repositories (MSR 2010). IEEE CS, Washington, pp 171–180

Shin Y, Bell RM, Ostrand TJ, Weyuker EJ (2009) Does calling structure information improve the accuracy of fault prediction? In: Proceedings of the 7th international working conference on mining software repositories (MSR 2009). IEEE CS, Washington, pp 61–70

Sim SE, Easterbrook SM, Holt RC (2003) Using benchmarking to advance research: a challenge to software engineering. In: Proceedings of the 25th international conference on software engineering (ICSE 2003). IEEE CS, Washington, pp 74–83

Subramanyam R, Krishnan MS (2003) Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects. IEEE Trans Softw Eng 29(4):297–310

Turhan B, Menzies T, Bener AB, Di Stefano JS (2009) On the relative value of cross-company and within-company data for defect prediction. Empir Software Eng 14(5):540–578

Turhan B, Bener AB, Menzies T (2010) Regularities in learning defect predictors. In: Proceedings of the 11th international conference on product-focused software process improvement (PROFES 2010). Springer, Berlin, pp 116–130

Wolf T, Schröter A, Damian D, Nguyen THD (2009) Predicting build failures using social network analysis on developer communication. In: Proceedings of the 31st international conference on software engineering (ICSE 2009). IEEE CS, Washington, pp 1–11

Zimmermann T, Nagappan N (2008) Predicting defects using network analysis on dependency graphs. In: Proceedings of the 30th international conference on software engineering (ICSE 2008). ACM, New York, pp 531–540

Zimmermann T, Premraj R, Zeller A (2007) Predicting defects for eclipse. In: Proceedings of the 3rd international workshop on predictive models in software engineering (PROMISE 2007). IEEE CS, Washington, pp 9–15

Zimmermann T, Nagappan N, Gall H, Giger E, Murphy B (2009) Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering (ESEC/FSE 2009). ACM, New York, pp 91–100

Marco D'Ambros earned his Ph.D. in October 2010 and is currently a postdoctoral researcher at the REVEAL Group, University of Lugano, Switzerland. He previously received MSc degrees from both Politecnico di Milano (Italy) and the University of Illinois at Chicago. His research interests lie in the domain of software engineering with a special focus on mining software repositories, software evolution, and software visualization.

He authored more than 30 technical papers, and is the creator of several software visualization and program comprehension tools.

Michele Lanza is associate professor of the faculty of informatics, which he co-founded in 2004. His doctoral dissertation, completed in 2003 at the University of Bern, received the prestigious European Ernst Denert award for best thesis in software engineering of 2003. Prof. Lanza received the Credit Suisse Award for best teaching in 2007 and 2009.

At the University of Lugano Prof. Lanza leads the REVEAL research group, working in the areas of software visualization, evolution, and reverse engineering. He authored more than 100 peer-reviewed technical papers and the book "Object-Oriented Metrics in Practice".

Prof. Lanza is involved in a number of scientific communities, and has served on more than 70 program committees. He is vice-president of CHOOSE (the Swiss Object-Oriented Software Engineering society). He was program co-chair of ICSM (the IEEE International Conference on Software Maintenance) in 2010, of MSR (the Working Conference on Mining Software Repositories) in 2007 and 2008, of VISSOFT (the IEEE Workshop on Visualizing Software) in 2009, of IWPSE (the International Workshop on Principles of Software Evolution) in 2007. He was General Chair of ESUG 2007 (15th International Smalltalk Conference). He is steering committee member of ICSM, MSR, VISSOFT, Softvis, and IWPSE.

Romain Robbes is an Assistant Professor at the University of Chile. He earned his Ph.D. in December 2008, from the University of Lugano, Switzerland and received his Master's degree from the University of Caen, France. His research interests lie in Empirical Software Engineering and Mining Software Repositories. He authored more than 30 papers on these topics, including top software engineering venues (ICSE, ASE), and best paper awards at WCRE 2009 and MSR 2011. He is program co-chair of IWPSE-EVOL 2011, and the recipient of a Microsoft SEIF award 2011.