The Journal of Systems & Software 168 (2020) 110673

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Using source code density to improve the accuracy of automatic commit classification into maintenance activities

Sebastian Hönel *, Morgan Ericsson, Welf Löwe, Anna Wingkvist

Faculty of Technology - Department of Computer Science and Media Technology, Linnaeus University, 351 95 Växjö, Sweden

ARTICLE INFO

Article history:
Received 15 September 2019
Received in revised form 24 May 2020
Accepted 27 May 2020
Available online 3 June 2020

Keywords:
Software quality
Commit classification
Source code density
Maintenance activities
Software evolution

ABSTRACT

Source code is changed for a reason, e.g., to adapt, correct, or adapt it. This reason can provide valuable insight into the development process but is rarely explicitly documented when the change is committed to a source code repository. Automatic commit classification uses features extracted from commits to estimate this reason.

We introduce *source code density*, a measure of the net size of a commit, and show how it improves the accuracy of automatic commit classification compared to previous size-based classifications. We also investigate how preceding generations of commits affect the class of a commit, and whether taking the code density of previous commits into account can improve the accuracy further.

We achieve up to 89% accuracy and a Kappa of 0.82 for the cross-project commit classification where the model is trained on one project and applied to other projects. Models trained on single projects yield accuracies of up to 93% with a Kappa approaching 0.90. The accuracy of the automatic commit classification has a direct impact on software (process) quality analyses that exploit the classification, so our improvements to the accuracy will also improve the confidence in such analyses.

1. Introduction

Every change to the source code of a software system has a purpose, e.g., to correct, perfect, adapt, or extend the system. This purpose can provide valuable insight into the development process, but is rarely documented as part of the change; developers either forget to do so or rely on default classifications, which are often wrong (Hindle et al., 2009). If we could automatically determine the purpose of a change, we could improve the documentation of the change and detect the kind of work done in a software project. This could support, e.g., to identify behavioral patterns, i.e., the developers' behavior and interaction with source code repositories. Such approaches put the developers' work in focus, augment their maintenance profile (Levin and Yehudai, 2016), and influence the development team composition. Regardless of the purpose, it is desirable to move away from error-prone or subjective classification of changes and to introduce objective approaches, as the accuracy to which we can determine the purpose of a change has a direct impact on the validity of our conclusions.

We suggest that *source code density*, the ratio between net- and gross size, can improve the accuracy of change classification. We define net size as the size of the unique code in the system and gross size as the size of everything, including clones, comments, and whitespace. Hönel et al. (2018) studied the size of changes to source code and found significant variances of the source code density but only a weak correlation to the change sizes. The purpose of a change could explain these variances. We measure the density of source code on commit-level. A commit may affect multiple files and different types of changes. We compare the gross-size of that commit, i.e., the sum of files or lines it affects, to its net-size, which is derived by reducing each change to its actual functionality. In Section 2.1, we outline how we define source code density, how it can improve the results of automated commit classification, and why commit size is an important predictor.

Previous work by Mockus and Votta (2000) considered features such as meta-information from the committed change, e.g., keywords and comments, properties of the changed source code, and external meta-information from, for example, the bug tracking systems. We hypothesize that changes to source code density better reflect the purpose and that this alone or in combination with previously considered features can improve the classification accuracy. The source code density of a change is, just like the size of the change, cheaper (in terms of effort and computation) to obtain or more convenient to use than some of the other features previously considered. So, even if a combination of features is needed, source code density may be used as a drop-in replacement for some more expensive or inconvenient features. It

* Corresponding author.
E-mail addresses: sebastian.honel@lnu.se (S. Hönel), morgan.ericsson@lnu.se (M. Ericsson), welf.lowe@lnu.se (W. Löwe), anna.wingkvist@lnu.se (A. Wingkvist).

https://doi.org/10.1016/j.jss.2020.110673
0164-1212/

is noteworthy that the density is a language-agnostic metric that does not require compilation of the underlying software, hence its inexpensiveness. To measure the effectiveness of our proposed features, we reproduce the current state of the art, then add to it, then derive from it, and finally suggest a combined model that delivers the best possible accuracy, improving the state of the art by double digits.

Software re-engineering and maintenance constitute a large part of acquiring knowledge about a system. Large portions of the knowledge in software systems are tacit or inaccessible. While external information and documentation may be used to gather knowledge, those are not always available. It is estimated that up to 60% of maintenance work is actually spent on comprehension (Kuhn et al., 2007; Abran et al., 0000). Automatic classification of changes has many applications and may help to reduce this time drastically. For instance, it allows us to understand the quality-related aspects of the software development process better. *Software aging* may be avoided by making *change* central in such processes (Fluri and Gall, 2006). Changes indicate that the process alternates between maintenance phases. In modern projects, features are developed in parallel, bugs are fixed out of band, and maintenance can be done during any of these activities. So, phases can and do overlap, which emphasizes the importance of understanding all ongoing phases.

Such improved understanding can be exploited in a multitude of ways, such as planning resources and personnel for maintenance activities, or to validate that the correct or expected type of planned work is carried out. This is particularly important for projects that are supposedly, e.g., in a feature-freeze phase, as in such phases, no adaptive activities shall be carried out. The type of carried out activity might also be used as a quality indicator when examining the activities' ratio over time, as one could expect, e.g., a project with more perfective and corrective than adaptive commits to be of comparatively higher quality.

We underline two aspects of the software development process in particular that commit classification has a high potential of improving: *process pattern detection* and *software quality monitoring*. A process pattern is an observable and reoccurring sequence of activities (Rising and Janoff, 2000; Alexander, 1977) followed in a software development lifecycle. Others have shown that identifying such patterns is valuable (Fluri et al., 2008), as it allows for, e.g., demonstrating that coding guidelines are not followed, or that newcomers lack sufficient training. While some patterns support the process, others are harmful and can be classified as anti-patterns. Various cures to each anti-pattern exist, but it is vital to detect them early to deal with them efficiently. Otherwise, they might result in delays or a decline in productivity or quality. Anti-pattern detection is often governed by data from Application Lifecycle Management (ALM) tools. Such tools extract data from project management applications to draw conclusions from ongoing and historical activities. However, they do not consider the underlying software artifacts that are developed or maintained (Pícha et al., 2017). Classifying the current activities can reduce ambiguities in detecting patterns by their symptoms. Others have demonstrated that such pattern detection best involves project-level metrics as well as developer-level information (Levin and Yehudai, 2016). A short anecdote may emphasize our case: During a collaboration, where our colleagues had access to such ALM data, we brought in quality information about the source code and were able to detect process anti-patterns, such as *Nine Pregnant Women*,[1] or the *Lone Wolf Programmer*.[2] However, since some patterns share certain symptoms, we were unable to distinguish which pattern occurred in certain phases. Our situation would have been relieved by having a proper commit classification at our disposal. This was the initial incentive to begin work on this study and to incorporate data and tools we already had at our disposal.

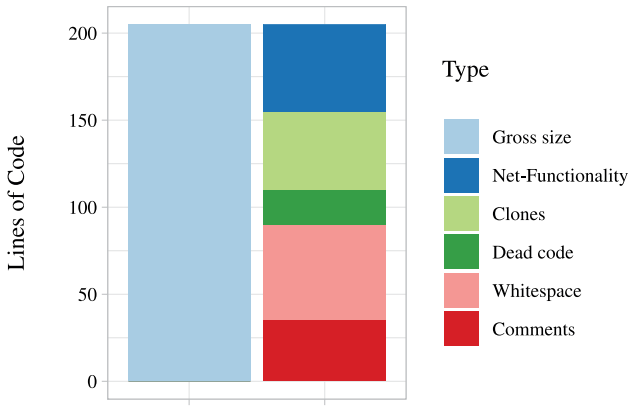

Fig. 1. Exemplary comparison of sizes measured for a single file containing some typical source code.

The central question of this research is whether the source code density can improve the accuracy of classifying a change by its purpose. We rely on the definitions of maintenance activities by Mockus and Votta (2000), and classify changes as either *(a)daptive*, *(c)orrective*, or *(p)erfective*. Adaptive activities add new features, corrective activities fix faults, and perfective activities restructure code to accommodate future changes.

The paper is structured as follows. Section 2 provides a deep and qualitative insight into the importance of size and source code density. Section 3 introduces the research questions and the approach to answer them. Section 4 presents the results. Section 5 discusses threats to validity. Section 6 gives an overview of related work. Section 7 concludes the research and Section 8 shows directions of future work.

2. Background

We have previously examined the impact of *code density* on effort- and productivity estimations (Hönel et al., 2018). Due to the unavailability of precise measures of spent time, we were unable to establish strong correlations. However, we found significant deviations of code density and size for various notions of the size of code committed to software repositories.

In this section, we first elaborate on the importance of change size and the potential of density. Then, we present the most relevant studies for size-based applications. The section is concluded by introducing the extended dataset used throughout this study, and size-based metrics therein.

2.1. The importance of size and the potential of density

There are various ways of quantifying the number of changes in (or the size of) a commit, as outlined in Section 6. This study focuses on measuring the size using varying forms of lines of code (LOC), however. Maintenance of software is such an integral part of its evolutionary process that it consumes much of the total resources available, according to a field study carried out

[1] https://github.com/ReliSA/Software-process-antipatterns-catalogue/blob/master/catalogue/Nine_Pregnant_Women.md.

[2] https://github.com/ReliSA/Software-process-antipatterns-catalogue/blob/master/catalogue/Lone-Wolf.md.

by Lientz et al. (1978). At the same time, maintenance phases lack a sufficiently strong understanding.

In its simplest form, determining the size by counting LOC neglects what these lines comprise. A typical example of a file containing source code is given in Fig. 1. In it, we usually find *cloned code*, that is, functional equivalent or even identical code that is to be found in at least one other file, or even in another portion of the same file. Dead code is an aggregation of code that cannot be called, such as statements that occur after the return statement (where valid) or non-reachable *if*/*else*-branches, and code that is never called within the application. Whitespace is any excessive empty lines and empty characters that do not contribute to the code's functionality. Comments, while useful or even necessary, are not counted, either. This is because of how we define the Density:

$$Density = \frac{\text{Functionality}}{\text{Size}_{\text{gross}}}. \tag{1}$$

The functionality comprises all code that purely contributes to the application's functioning; thus it does not include comments, whitespace, cloned (duplicated) functionality, and dead code. Hence, the density can maximally approach 1, and minimally be 0.

Determining the density for a single file is useful, and extending the approach to the entirety of a software may have a significant meaning for estimation models. For example, the International Software Benchmarking Standards Group (ISBSG) further defines the project metrics *effort* and *productivity*, and bases them mainly on the size of software,[3] and change over time thereof:

$$Effort_{ISBSG} = \frac{\text{Size}_{\text{gross}}}{\text{time}} \tag{2}$$

$$Effort_{net} = \frac{\text{Density} \times \text{Size}_{\text{gross}}}{\text{time}} \tag{3}$$

$$= \frac{\text{Functionality}}{\text{time}} \tag{4}$$

Eqs. (2) and (5) show how ISBSG defines effort (size per time) and productivity (effort per size). The most common method used by them for determining size is function points (FP). A function point is a unit of measurement to express the amount of business functionality in a software. Various types of FP exist and those are specified mostly as ISO standards. FP are usually counted manually. Albrecht and Gaffney found a strong positive correlation between FP and LOC (Albrecht and Gaffney, 1983), thus questioning the value of FP, given its cost compared to counting LOC. Adapting the ISBSG-equations to use notions of net-size ($Size_{net} = \text{Density} * \text{Size}_{\text{gross}}$) instead of gross-size or size measured in terms of function points, leads to Eqs. (3), (4) and (6).

$$Productivity_{ISBSG} = \frac{\text{Effort}}{\text{Size}_{\text{gross}}} \tag{5}$$

$$Productivity_{net} = \frac{\text{Density}}{\text{time}} \tag{6}$$

Estimation models with a strong focus on software size may behave differently, given these alterations. In this study, we report significant deviations between the net- and gross-size of software. While strongly positively correlated, the correlation is non-linear. Estimating the net-size, if counted as LOC, can be automated conveniently, unlike counting function points. Furthermore, while net-size allows a better approximation of effort and productivity, it may also allow increasing the confidence in automatic commit classification. This comes into play when the different activities in a software project are estimated individually.

2.2. The most relevant studies for size and density

It is crucial to outline why our study, which evolves around the size of commits, is important. More specifically, the following short qualitative study of the most relevant related work emphasizes the two questions:

- Is the size of a commit an important predictor?
- Why may *source code density* improve the results of automated commit classification?

The size of software, regardless of how it is measured, is often considered a low-level metric for software and its evolutionary process (Herraiz et al., 2006). While it is a simple and thus computationally cheap metric, compared to metrics such as cyclomatic complexity or coupling/cohesion, discourse about its applicability and how it should be obtained, exist. Herraiz et al. point out that discrepancies about measurements of size, especially between *libre* (free, open source) and *'traditional'* software, exist. While the method of measuring the size is different, the evolution of software belonging to either system, according to the laws of software evolution by Lehman and Belady (1985), appears to be the same. This work is only concerned with libre software. We report significant deviations of the obtained net- and gross size measurements. In the context of software evolution analysis, the automated classification of commits, based on density rather than on raw LOC, has the potential to resemble the actual size of software more closely, and thus to improve the analysis process. A repetition of the study by Herraiz et al. (2006) using density is thus likely to yield different results.

The size of a commit can also reveal developmental aspects and practices of the software evolutionary process, as studied by Hindle et al. (2008). Large commits, for example, often happen when branches in a repository are merged, which hints at a process where features and bugfixes are developed separately and then integrated once they achieve a certain level of maturity or tests are passed. It has become quite common to rely on external packages that are downloaded whenever the software is built or run for the first time. However, there is also software that incorporates external code, such as libraries or frameworks, instead of referencing it (Hindle et al., 2008). Large commits reflect such behavior in the first place and can give an indication of the software's maintainability, as incorporated code is much less frequently updated. Large commits may also be more correlated with automatically generated code or documentation. As for the relation to this work, it was found that large commits are most often *perfective*. The size is, therefore, relevant to reveal such aspects. Using the density instead of (or in conjunction with) the size may help to better identify developmental aspects and to reduce ambiguities that would otherwise arise from only counting LOC. Like the metadata of commits, such as its keywords and message, author, and timestamps etc., is attractive because it does not require any extra computation. The same is true for LOC-based size methods, which are, on top of that, language-agnostic. Such methods can, therefore, be effortlessly integrated into existing metadata-based solutions and improve and aid the automatic classification, which is still a challenge (Hindle et al., 2008). Having an automated classifier is desirable, especially for software where tagging the commits with a purpose was not previously enforced or must be done retroactively.

Small commits have shown to be a significant predictor for faults (Leszak et al., 2002), with LOC being an effective predictor (Bell et al., 2011). Purushothaman and Perry point out

[3] ISBSG: "Software size as the main input parameter to cost estimation models.", http://isbsg.org/software-size.

that the impact of small changes to source code is often underrated (Purushothaman and Perry, 2005). This leads to less rigorous processes in the software evolutionary process, such as lack of testing. The implications on the system's architecture originating from small changes are often associated with small risk, too. There are cases where one-line changes cost more than one billion US dollars, as reported by Weinberg (1983). As we demonstrate, the majority of such small changes are related to correcting bugs, followed by introducing new features. These are also the activities mainly correlated with introducing faults. An automated commit classification could aid the process of flagging such potentially fault-introducing commits for a more thorough audit. Whether small, large, or anywhere in between, the size of a commit is a piece of valuable information worth exploiting. Mockus and Votta (2000) report strong relationships between the type and size of a change. The impact of using density instead of raw LOC is significant when classifying commits, and hence to triage them for further inspection.

2.3. The extended dataset

Our extended dataset (Hönel, 2019) is based on the dataset by Levin and Yehudai (2017a) but extended with size data. The original dataset (Levin and Yehudai, 2017b) consists of more than 1 150 manually labeled commits from eleven projects. We used our tool suite, *Git-Density* (Hönel, 2020), to collect size data for the 11 projects. For all but the two projects, namely *Intellij Community Edition* and *Kotlin*, we have added size data for all commits of each project's repository, as of January 2019. From those two projects, we have analyzed all commits that were contained in the initial dataset, plus the first 30 000 and 35 000 commits respectively. We merged the two datasets using each commit's unique SHA1 hash. During this process, we identified two duplicate commits in the original dataset, effectively reducing it to 1 149 samples.

We define the *size* of a commit to be either the number of files or the LOC that were changed, across all types of changes, i.e., files/lines added, deleted, modified, or renamed. The *Gross size* is the size without considering whether a line affects the functionality of the source code or not. *Net size*, on the other hand, is the gross size minus the number of files or LOC that did not contribute to actual changes to the software's functionality. We consider empty lines, whitespace, as well as single- and multi-line comments to be without such effect. Conversely, if any other source code line was changed, it is undecidable whether or not the functionality changes, and we conservatively assume it does. If none of the changed lines in a file is considered to contribute to changed functionality, then neither is the file.

The *Change density* (or short density) is the ratio between *net* to *gross* size of a change. If all lines changed in a commit potentially contribute to the software's functionality, then the density takes its maximum value of 1.0. Conversely, if no line changes, the density takes its lowest value of 0.0.

The extended dataset contains the following eight features (gross and net sizes) that describe the number of files that have been added, deleted, renamed, or modified in a commit. Renaming a file means that a file is deleted in one place and reappears in another place, without having its content changed (pure rename). If its content is similar by 50% or more (but less than 100%), the change is considered an impure rename (common git threshold). If the similarity undercuts the threshold, the commit exhibits one deleted and one added file instead. For brevity, we sometimes refer to a feature by its number (or its number followed by an **a** if we refer to its corresponding net-version).

1. Number of Files Added (Gross and Net)
2. Number of Files Deleted (Gross and Net)
3. Number of Files Renamed (Gross and Net)
4. Number of Files Modified (Gross and Net)

There may be one or more changes per file. These are called *Hunks*. We can determine the density of an entire file in a commit by aggregating the properties of its hunks. If the aggregated changes of all hunks of a file amount to zero lines affected, then the respective net-feature does not count the file as being affected. This is an important measure, as previous researchers have also determined the size of a commit by the number of files it affects (Herraiz et al., 2006).

5. Number of Lines Added by Added Files (Gross and Net)
6. Number of Lines Deleted by Deleted Files (Gross and Net)
7. Number of Lines Added by Modified Files (Gross and Net)
8. Number of Lines Deleted by Modified Files (Gross and Net)
9. Number of Lines Added by Renamed Files (Gross and Net)
10. Number of Lines Deleted by Renamed Files (Gross and Net)
11. Affected Files Ratio Net
12. Density

An additional feature, Affected Files Ratio Net (11), expresses the ratio between the sums of all gross (1–4) and net (1a–4a) files above. We also added the feature Density (12) to the dataset. It describes the ratio between the two sums of all lines added, deleted, modified, and renamed and their resp. gross-version. A density of zero means that the sum of net-lines is zero (i.e., all lines changed were just clones, dead code, whitespace, comments, etc.). A density of 1.0 means that all changed net-lines contribute to the gross-size of the commit (i.e., no lines considered useless with, e.g., only comments or whitespace).

These twelve attributes count the gross amount of lines of code affected by added, deleted, modified, and renamed files, while their corresponding net-version counts the net-amount. Note that added files can never go along with deleted lines, and that deleted files can never include added lines. As for renamed files, the standard 50% similarity threshold applies; therefore, those can have either type of change.

3. Methodology

Classification is a common problem of statistics and machine learning. Classification models are fit using labeled data, with the intention to correctly label previously unseen observations based on some of their features. Commit classification models may facilitate and learn from a number of different features to achieve this task of generalizing from examples. Previously, some of these models were based on, e.g., keywords and comments (commit messages) (Mockus and Votta, 2000; Levin and Yehudai, 2017a). Some other models used notions of commit size (Hindle et al., 2008, 2009; Herraiz et al., 2006; Purushothaman and Perry, 2005). In this study, we attempt to use a more well-defined size metric, the *density*, to build classifiers that can assign commits to maintenance activities. Such classifiers, once trained, can be used for *automatic* classification of previously unseen and uncategorized commits. Without these, one would have to resort to manually labeling instances, which is error-prone and may require an extensive set of rules.

In the remainder of this section we pose our research questions and devise several experiments to resolve them empirically. We then outline the statistical methods that we apply to examine the gathered data.

3.1. Research questions

RQ 1: Does the net-size of commits as gathered in the extended dataset reveal significant differences, compared to the gross-size that was used in prior studies?

A. Are the evolutionary patterns the same for classifying commits when gross- and net size of file- and line counts are considered?
B. Do the size and frequency change when considering the net size?

RQ 2: Using the existing maintenance activities (labels), how well do source code density (including gross- and net size) alone allow for a classification of maintenance activities?

A. Is there a difference in accuracy for cross- and single-project classification?
B. Do the net size features perform better in classification, compared to their gross size counterparts?

RQ 3: Are the size- and density features suitable for improving state of the art in commit classification accuracy?

A. Are the previous results as obtained by Levin and Yehudai (2017a) reproducible?
B. If we extend their models with size and density data, does the accuracy improve?
C. Is there a best subset of features, that combines source code density features and those from Levin and Yehudai, i.e., a set of best features across all datasets?

RQ 4: What is the impact of size features of commits in previous generations when classifying a (principal) commit?

A. What are the most important features of the principal commit?
B. How important are density- and size features in preceding generations?
C. Is there a significant difference between cross- and single-project classification?

3.2. Statistical methods

Previous studies (Fernández-Delgado et al., 2014) found Random forests to perform well in general. Based on this, we mainly use Random forests to obtain rankings of predictors (variable importance) (Breiman, 2001; R Core Team, 2017). The research we relate our work to reports classification results for single projects and cross-project. They achieved the best results using Gradient Boosting Machines (GBM) (Friedman, 2002) and Random forests. To evaluate and compare their accuracy, we apply the Zero Rule (ZeroR) classifier to set a baseline. For the prediction of categorical variables, that classifier always predicts the most common class. We use *R* (R Core Team, 2017) to perform all experiments and analyses.

Throughout this paper we do only report classification accuracy and Cohen's Kappa. Some of the related work we refer to reports other or additional metrics, such as precision, recall, or F1-score. However, accuracy and Kappa is to be found in most of the other studies, and thus makes our work comparable to them. Kappa is a metric that, if it is reported along with accuracy, mitigates some of the caveats of the F1-score. It is defined as:

$$Kappa = \frac{\text{Accuracy}_{\text{total}} - \text{Accuracy}_{\text{random}}}{1 - \text{Accuracy}_{\text{random}}}. \quad (7)$$

Given an uneven or strongly skewed distribution of classes in a dataset, the reported accuracy and F1-score may very well be high using, e.g., the ZeroR classifier, as those metrics do not correct for the bias of such skewed distributions, or how the agreement between raters could occur by chance. Cohen's Kappa however corrects for those, and would give an indication of the low agreement between predicted and true classes. We deem the combination of classification accuracy and Kappa as metrics therefore to be sufficient.

Research Question 1 addresses statistical properties, such as distributions of the commits' labels of the extended dataset. These properties are useful for putting the dataset into relation to the work of other researchers, such as Hattori and Lanza (2008) and Purushothaman and Perry (2005). As statistical tools, we make use of (Empirical) Cumulative Distribution Functions (E)CDF and empirical densities to find similarities and differences between the nature of commits concerning the different notions of size. It is important to note that related work used different manually labeled datasets, not a common benchmark suite.

For Research Question 2A, we apply the methods to the entire dataset and report the models' accuracy. The goal was to understand the importance of each new attribute, not to predict validation samples. Therefore, we apply the following methods across all projects and then to every single project separately:

- Remove zero-variance predictors. Within the scope of a single project, some features do not exhibit any variance any longer and are therefore removed. Between projects, such zero-variance features varied.
- Identify highly correlated (coefficient larger than 0.75) features using the *Pearson* co-variance. However, we keep the features for further analyses and eliminate later, when assessing variable importance and doing a separate Recursive Feature Elimination (RFE, elaborated below).
- Assess *variable importance* using a Receiver Operating Characteristic (ROC) (Hanley and McNeil, 1982) curve analysis by applying the Learning Vector Quantization (LVQ) (Kohonen, 1995) method. We prefer LVQ over other methods, as it reports importance for each label and outperforms other methods. We also tried to use GBMs and *eXtreme Gradient Boosting* (xGB) (Chen et al., 2015). However, those turned out to be significantly slower (runtime) and do not report importance per label.
- Run an RFE across all projects and for each project individually. Attempt to use between just one and all available features (i.e., those that withstood the previous RFE). Use Random forests to extract variable importances, using ten-fold cross-validation, while computing at least three sets of complete folds.

The R package *Caret* (Kuhn, 2008) implements RFE and provides sets of interchangeable methods to fit models and allows re-sampling using, e.g., cross-validation. In RFE, the underlying fitting method first fits a model to the training data using all of its predictors. Then, low-weight features are removed recursively with each iteration. Ideally, such method also provides the variable importance to rank the features. Random forests hence is a suitable candidate, and we have used it, also because of its favorable accuracy.

For a set S_i of attribute sizes referring to the top-ranked i attributes, the model is refit using those attributes, then that model's accuracy is assessed, and in the end, the best model is retained.

We follow suggestions that recommend having the model selection process use external validation through re-sampling by cross-validation (Ambroise and McLachlan, 2002; Svetnik et al., 2004). The described procedure for model-fitting on best-ranking attribute subsets was therefore nested in a k-fold cross-validation, using three or more complete sets of folds.

For Research Question 2B, we compare the net- vs. gross-size attributes of the extended datasets. We report the results for individual projects and across projects. The steps undertaken are:

- Vertically separate the extended dataset into one that contains only the net- and one that contains only the gross-version of each attribute. Both datasets retain the labels.
- Assess the variable importance of either dataset using a ROC curve analysis, based on Random forests, using a 10-fold and three times repeated cross-validation.
- Repeat the last step and gather results for each project.

Research Question 3 is three-fold. Previously, Levin and Yehudai (2017a) demonstrated strong classification results using commit keywords and source code changes. We are interested in examining whether the predictive power of their models can be further enhanced using size attributes.

- Attempt to reproduce the previous researchers' results by using their dataset (Levin and Yehudai, 2017b) and methods. They report training- and validation accuracy of J48-, GBM- and Random forest-based models. As their champion model uses Random forests, we only reproduce those models.
- Preliminary split the data into 85% training and 15% validation samples. Then further vertically split the training data into one dataset containing only keywords, one containing only changes and one that combines both of these, so that we may reconstruct all types of classifiers used.
- Use the custom classifier they suggest for compound models (see below this list). The combinations of two models A, B in shape of $\{A, B\}$ and $\{B, A\}$ are considered to be distinct by that classifier.
- Build and train three different models (one per subdivided dataset), using Random forests and five times repeated 10-fold cross-validation (that validation happens entirely on the 85% training data).
- Construct the compound models. Compound models have a left and a right model. For those models based on just one type of model (e.g., keywords), use the same model on both sides.
- Run the custom classifier on each compound model. The classifier uses the left model whenever a sample uses at least one keyword out of the 20.
- Run the classifier on the 15% of the previously unseen validation samples, report accuracy during training and validation, and compare. We combine the numeric votes for each class of each model and select the highest (the models return a probability for each class) when we report the training accuracy for compound models.

There are a total of nine compound models, as there are three types of possible underlying models $\{keywords, changes, combined\}$. The nine models are the result of building all permutations (3^2). A compound model is the combination of two models (a "classifier lattice", cf. Levin and Yehudai (2017a)), such that the routine for classifying a commit uses a different model, depending on whether the commit's message has any of the keywords the keyword-classifier was trained on. This notion of a 2-compound model was introduced, as the keyword-based classifiers outperformed the other classifiers if keywords were present. These compound models do not overlap because each single model may or may not be a reduced-feature model.

For simplicity, we refer to the two models in a compound model simply as left and right model. For the second part of this research question, we add one more type of underlying model, namely $model_{\text{density}}$, resulting in 16 compound models (4^2). That model is based and trained on size data only. Also, we alter the $model_{\text{combined}}$ model to also span the size attributes. The models using $\{keywords, changes\}$ remain the same. We then apply the same procedures as in the previous list to report training- and validation-accuracy.

Table 1
Types of principal commits used in the four datasets of RQ 4.

Dataset	Type of the principal commit
A	Using the features of Levin and Yehudai (2017b) (keywords, code-changes).
B	Using only density- and size related features from our extended dataset (Hönel, 2019).
C	Using both the features of datasets A and B (keywords, code-changes, density-/size features).
D	Same as C, but without keywords.

As for the last part of Research Question 3, we attempt to further tune and prune the 16 compound models. The steps involved are:

- Create a density-only dataset, based only on net-attributes.
- Attempt further optimizations to that dataset, by conditionally leaving out zero- and near-zero-variance attributes and preprocessing it. Attempt various (combinations of) preprocessing, such as scaling (divide by mean), centering (subtract mean), or Yeo–Johnson transformations (Yeo and Johnson, 2000) (suitable as we are dealing with power-distributed data that can be zero).
- Analyze the variable importance of that dataset to find the optimum amount of variables for further training.
- Since the previous authors achieved the best results using Random forest, attempt to manually tune such a model with regard to m_{try}.
- Evaluate other classification methods that may be suitable and pick the best-performing for further optimizations.

3.3. Incorporation of parent commits

Research Question 4 was conceived in a way that allows us to validate the results as obtained in the previous questions. There, the results compare cross- vs. single-project commit classification, the accuracy of density- and size features, and how well the attributes of our extended dataset perform, also in comparison with the datasets of Levin and Yehudai (2017b).

To build a dataset that is made up of chains of commits, where the nature of the youngest child commit (the principal commit) is to be predicted, one needs to know about the direct predecessors (one or more generations of direct parents) of it. The labeled dataset from Levin and Yehudai does not feature consecutively labeled commits. However, we have gathered such relational information within our extended dataset, which also covers all the commits from Levin and Yehudai. That implies that all of the parent commits are sourced from our extended dataset, and therefore can only include its features (i.e., we do not have keyword- or code-change-features at our disposal). However, the principal commit is allowed to have any of the features. None of the commits involved is a merge-commit. We were stringent about excluding such, as those need to be further investigated due to their potentially mixed nature.

We are building four different datasets, which are distinguished from each other by the type of principal commit. We are differentiating four types of principal commits (cf. Table 1). Then for each dataset, a sub-dataset is built, featuring one or more generations of parents. We are selecting $\{1, 2, 3, 5, 8\}$ as the amounts of parents to be included, as those still yield a respectable dataset size (almost 900 commits have eight parents) and resemble the Fibonacci series. In total, we are thus using 20 datasets for Research Question 4. Since the relation of each commit to its project is retained, we can use the same datasets for cross- and single-project classification.

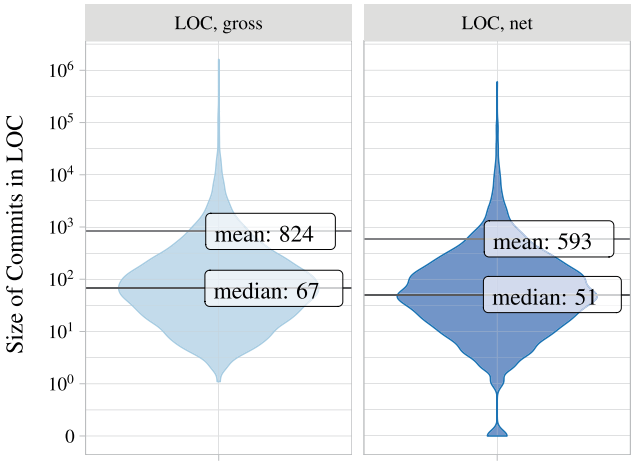

Fig. 2. The size of commits in LOC, across the almost 360 000 commits of the extended dataset.

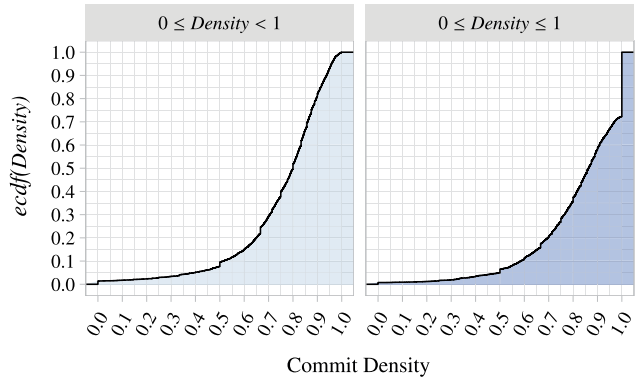

Fig. 3. The empirical cumulative probability distribution of commit density across the almost 360 000 commits of the extended dataset. About every tenth commit has a density of 0.5 or less; about every other commit has a density of 0.8 or less. Considering *Density* = 1 commits (right), more than every fourth has a density of 1.

Research Question 4 is addressed by comparing the accuracy across its A, B, C, and D datasets. The second part is partially covered as well. However, we intend to point out the importance of the net- and gross-attributes separately. Lastly, the accuracy and Kappa of the champion models built across projects and for every single project are evaluated. For statistical models, we are using RFE based on Random forests and repeated cross-validation, using many folds to find champion models.

4. Results

The results are laid out for each research question and summarized at the beginning of each subsection.

RQ1: The statistical properties of the extended dataset

- The evolutionary patterns using file- or line-counts are significantly different, contrary to prior research.
- There is a significant shift in what constitutes a small commit; every tenth commit affects 3–4 lines, every other already 25–50.
- Instead of *corrective*, many zero- or near-zero size commits need to be considered as *perfective* instead.
- Most commits have a high density. This affects commits of all sizes.

Herraiz et al. (2006) found that the evolutionary patterns for commits in open source software are the same, regardless of whether they were based on counting the number of files or lines of code. Purushothaman and Perry (2005) investigate what they consider to be tiny commits, and found that 10% of all commits are small changes, i.e., only affecting a single line.

To address Research Question 1, we gathered descriptive statistics for the datasets used. We find a weak correlation between notions of size based on either amount of affected files or lines of code. This suggests that it is worthwhile to investigate a commit's nature using a LOC-based notion of size. We then confirm previous results, relating the size to the nature, and find that corrective commits are usually the smallest. Research Question 1B investigates the nature of commits that change only a few lines. While we observe a difference between net- and gross-size, we find that just a small ratio of commits affects five or fewer lines. When examining the maintenance activities of such small commits further, we find that more commits should be considered perfective, using a net-notion of size.

It is important to understand the significance of the size of a commit, and especially its density. To demonstrate it, we took the size attributes of the extended dataset of commits, holding almost 360 000 commits, into account. Out of these, 3 279 (1%) had a size of zero. Those are due to, e.g., starting or stopping to track files that are empty or changes to binary files that result in no lines changed. Another 98 869 commits had a density of one, meaning that all affected lines contributed to its net-size. 71.59% of the commits hence were non-empty, and had a density in the range [0, 1), as can be seen in Fig. 2. Following our expectation, the correlation between net- and gross-counts of LOC in these commits is 0.9885 (strong positive).

Taking the distribution of the commit density into account, it is apparent that larger densities are much more common than lower densities. About only every 10th commit has a density of 0.5 or less, while already about every other commit has a density of 0.8 or less, cf. Fig. 3. To further understand commit density and how it relates to a commit's gross size, we prepare a few ranges and visualize the distribution of densities, cf. Fig. 4.

A significant portion of the examined commits in the extended dataset, more than 28%, have the maximum density of 1, meaning that the net-size is the same as the gross-size. Hence, all lines in these commits are considered useful. Therefore, we have considered these separately. This phenomenon affects commits of all sizes, starting from one line up to several hundreds of thousands of lines. However, the majority of such commits has about ten lines or less, cf. Fig. 5.

PART A Research Question 1A seeks to validate whether the size of a commit in terms of affected files or LOC is different. Prior research (Herraiz et al., 2006) found the difference to be insignificant. When comparing the density plots in Fig. 6, we observe that the minimum values for gross values (i.e., LOC or files) are 1.0 (as a commit cannot comprise an empty set of changes), whereas the net-values can be, and in fact are, 0. We have shifted all net-values for these plots by 0.1, so we can use a logarithmic scale. This allows us to observe commits assigned to any of the maintenance activities, that have in fact a size of zero. Refer to Table 2 for the numerical properties of the various notions of size and empirical probabilities. The table also outlines the probabilities of finding commits with a size of 0 for each of the maintenance labels.

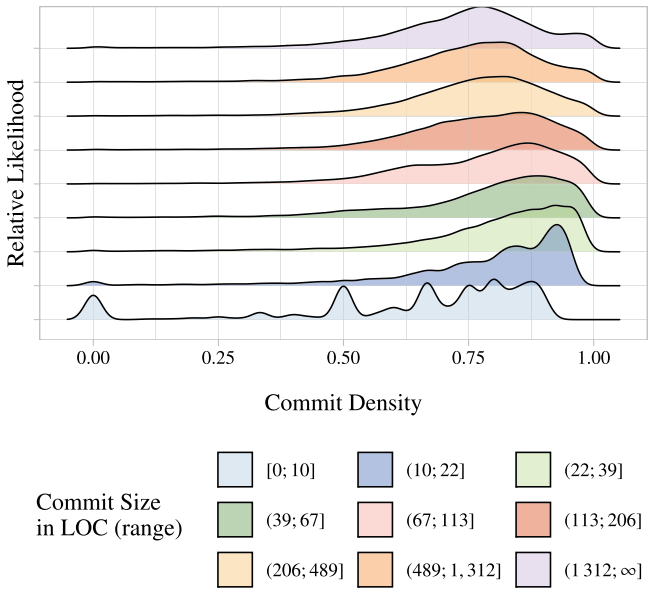

Fig. 4. Ridge-plot of the probability distribution of commits' densities, for a number of delimited ranges, as expressed by gross-size lines of code. The ranges were delimited using the following quantiles: 12.5% (10), 25% (22), 37.5% (39), 50% (67), 62.5% (113), 75% (206), 87.5% (489), 95% (1 312).

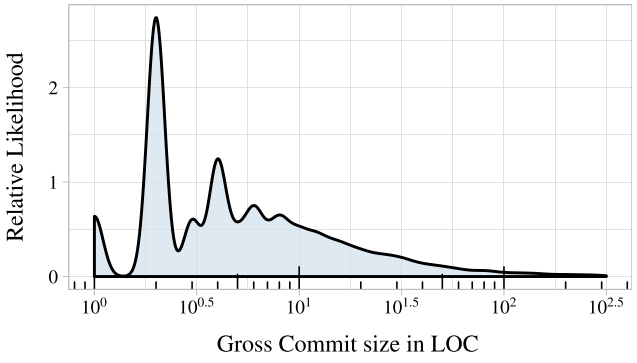

Fig. 5. The distribution of commits having a density of 1, w.r.t. their gross-size in LOC (tail truncated).

The evolutionary patterns between classification using files and LOC are quite different. While the amount of *adaptive* commits increases with size for LOC-based notions, file-based notions attribute most large commits towards *perfective* and *corrective* commits. The latter has its largest commits attributed to *adaptive* activities, while the former identifies the largest commits to be *perfective* and *corrective*. Evolutionary patterns between net- and gross-size notions differ only slightly. However, we can get insights into the densities of net-notions that feature empty commits. Those could explain the differences in density when compared to their gross-sized counterparts.

From the empirical probabilities in Table 2, we can derive some statements. First, the probability of observing an empty commit of activity *adaptive* is zero. If any, then the observed type of commit is either of corrective or adaptive nature. We can see that there are, occasionally, significant differences in observing either type of maintenance activity, given the size is zero or in the interval $[1, 2)$. Therefore, when considering net-sized datasets, we can observe a shift in the distributions for the maintenance activities. This shift can also be observed when examining the density plots in Fig. 6. Additionally, regarding the weak correlations between files- and LOC-based gross- and net

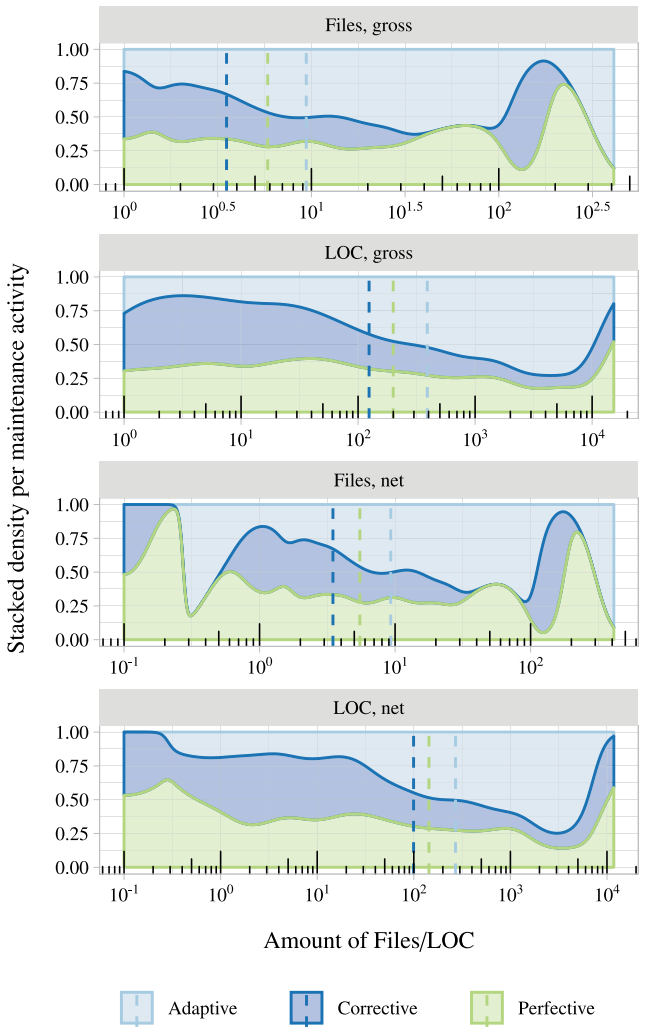

Fig. 6. Density plots of net-/gross amount of files and LOC, across all projects and labels, including the mean for each label.

Table 2
Numerical properties and empirical probabilities of gross- and net datasets w.r.t. the commit's label.

	Files		LOC	
	Gross	Net	Gross	Net
$mean_{\text{adaptive}}$	9.386	9.288	390.297	270.255
$mean_{\text{corrective}}$	3.524	3.480	124.172	99.458
$mean_{\text{perfective}}$	5.843	5.496	199.623	143.439
$mean_{\{a,c,p\}}$	5.592	5.431	207.612	151.452
$median_{\{a,c,p\}}$	2.000	2.000	45.000	33.000
$min_{\{a,c,p\}}$	1.000	0.000	1.000	0.000
$max_{\{a,c,p\}}$	411.000	411.000	15 318.00	11 766.060
$P(a \mid x < 1)$	*n/a*	0.000	*n/a*	0.000
$P(a \mid 1 \leq x < 2)$	0.179	0.179	0.012	0.012
$P(a \mid 2 \leq x < 5)$	0.528	0.528	0.049	0.061
$P(c \mid x < 1)$	*n/a*	0.004	*n/a*	0.004
$P(c \mid 1 \leq x < 2)$	0.390	0.388	0.010	0.020
$P(c \mid 2 \leq x < 5)$	0.818	0.816	0.154	0.188
$P(p \mid x < 1)$	*n/a*	0.005	*n/a*	0.005
$P(p \mid 1 \leq x < 2)$	0.340	0.357	0.007	0.030
$P(p \mid 2 \leq x < 5)$	0.742	0.752	0.101	0.134

datasets (0.347 and 0.296, respectively), it might be worth to investigate the nature of a commit w.r.t. affected LOC, instead of

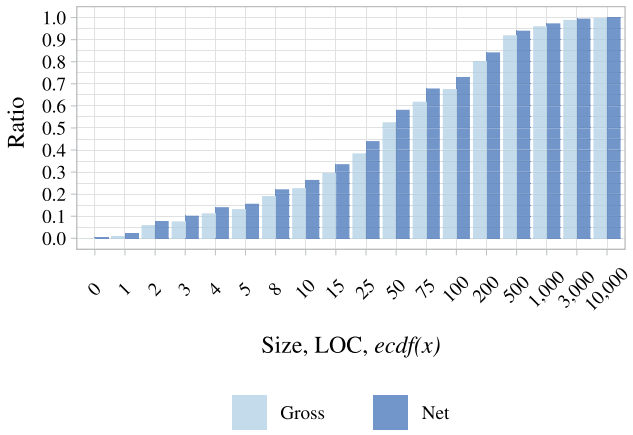

Fig. 7. Empirical distribution of commits w.r.t the affected lines, separated by gross- and net size.

files, as done in prior studies (Herraiz et al., 2006; Hattori and Lanza, 2008; Hindle et al., 2008).

Further expanding on *Do development activities appear mainly in small commits?* (posed by Hattori and Lanza), we conclude that corrective commits are the smallest, followed by perfective, and then adaptive commits. In that same order, it is more to less likely to encounter a commit affecting between two and five lines. Our observations are, therefore, in consensus with those of Hattori and Lanza. The difference between gross and net is insignificant.

PART B Research Question 1B concerns the size of commits with regard to their gross- and net-size. Purushothaman and Perry found that every tenth commit changed only a single line of code and that nearly 50% changed ten lines or less. Given our extended dataset that spans eleven projects, we found that one out of ten commits affected about four lines or less, as can be seen in Fig. 7. In general, we can observe a shift towards an increased ratio of net-sized commits. More than half of the commits affected 50 lines or less in our data.

The distribution of maintenance activities across commits up to a specific size is different for gross- and net-size, as depicted by Fig. 9. In the upper plot, more than 25% of the commits are considered to be adaptive when only one line is affected by them. Regardless of the examined sizes, about 25% of the commits are of perfective nature. The lower plot, which depicts high-density commits, finds that commits that affected zero lines in actuality are either corrective or perfective. Almost all of the zero-lines commits that were adaptive previously, need now considered to be corrective.

RQ2: Commit classification using only size and density

- Classification accuracy and Kappa show a wider spread for individual projects. Less variables are required for single projects.
- Net-versions of attributes are deemed more important than their respective gross-counterpart.
- While models using net-size variables profit from each added variable, this leads to larger models. Gross-size based models are less complex and perform slightly better.

We have found differences in accuracy between classifiers trained across the entire dataset and for each project individually. This is also partly due to the different distributions of maintenance activities in each project, esp. when compared across all projects. In Fig. 8 the most common maintenance activity across projects is *corrective*. However, for individual projects, we observe that this is not always the most prevailing activity. By correlating the attributes of our extended dataset, we find strong positive correlations and can prune it considerably. It becomes clear that net-versions of attributes are deemed more important than their respective gross-counterpart.

PART A We use the methods described in Section 3.2 to perform a series of experiments. From this, we report the following results for Research Question 2A. Generally, classification accuracy and Kappa had a wider spread for individual projects (cf. Table 3 and Figs. 10 & 11), compared to cross-project classification.

Our tool, *Git-Density* (Hönel, 2020), adds 22 size attributes (cf. Section 2.3). Using a correlation coefficient of 0.75, we have identified eleven highly correlated attributes (Number of Files Added Gross, Number of Files Deleted Net, Number of Files Renamed Gross, Number of Files Modified Gross, Number of Lines Added by Added Files Gross, Number of Lines Deleted by Deleted Files Net, Number of Lines Added by Modified Files Gross, Number of Lines Deleted by Modified Files Gross, Number of Lines Added by Renamed Files Gross, Number of Lines Added by Renamed Files Net, Number of Lines Deleted by Renamed Files Gross) across projects.

Attribute correlation across projects kept eleven attributes, nine of which were the *net*-version of an attribute; the attributes *Density* and *Affected Files Ratio Net* were kept. For the eleven single projects, we instead counted the most highly correlated attributes for each project. Those were (followed by the count) Number of Lines Deleted by Deleted Files Gross (5), Number of Files Deleted Gross (4), Number of Files Renamed Net (4), Number of Lines Added by Added Files Net (4), Number of Lines Deleted by Renamed Files Net (4), Number of Files Added Net (2), Number of Lines Added by Modified Files Net (2), and Number of Lines Deleted by Modified Files Net (2).

Since correlation is always done pairwise, the variable with the largest mean absolute correlation is identified for removal. Note that we have not removed the highly correlated attributes, though, as the next step was applying the variable importance, which determines the importance of each predictor independent of the correlation.

We can report low variance for features derived from deleted or renamed files. This is somewhat expected, as deleting and renaming files occurs much more infrequent than adding new or modifying existing files. The overall *Density* attribute we engineered shows high variance and is the fifth most important predictor (out of eleven) across all size attributes, with average and maximum importances of 61.91% resp. 72.51% across projects.

Since there is less data available to examine the variable importance for each single project, it frequently happened that specific attributes showed no variance within the scope of a project. The seven attributes that consistently remained across all projects were Number of Lines Added by Modified Files Net, Number of Lines Added by Added Files Net, Number of Files Added Net, Number of Files Modified Net, Density, Number of Lines Deleted by Modified Files Net, and Affected Files Ratio Net (ordered descending by average importance across labels). Note that all of these attributes are the net-version of their feature. We have further examined the importance of each remaining attribute for each of the maintenance labels *{a,c,p}* separately. It is noteworthy that our engineered feature *Affected Files Ratio Net* improved considerably by 45% (averaging at 58.37%) and that *Density* gained about 3% in importance. Refer to Fig. 12 for detailed boxplots.

Model-selection for single projects shows peculiarities for the projects *Drools* and *Hadoop*, where the best model uses only one

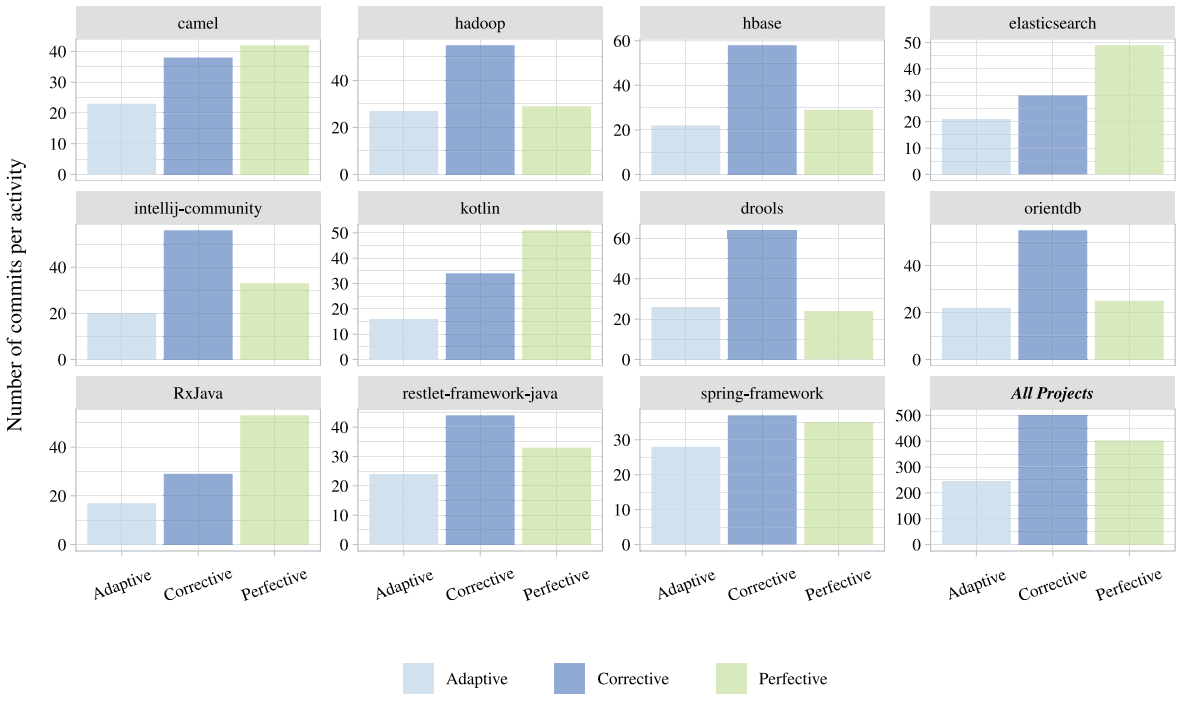

Fig. 8. Distribution of commit maintenance activities across each individual project, and for all projects combined (last plot).

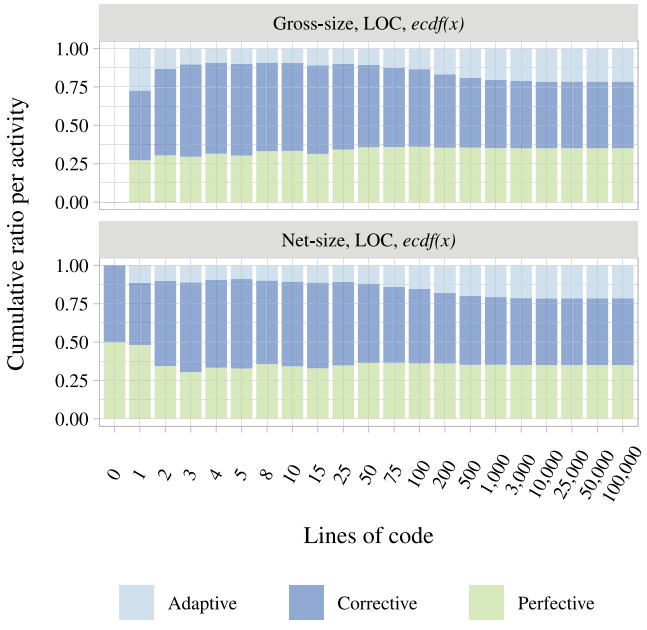

Fig. 9. Empirical distribution of maintenance activities in commits, separated by gross- and net size.

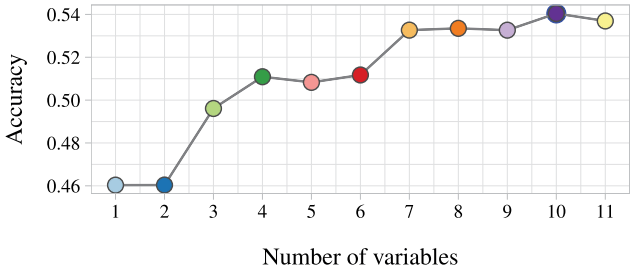

Fig. 10. Cross-project model training performance w.r.t. number of most important variables. Each point represents a champion model based on the amount of variables denoted. The resampling during training was done using a five times repeated, 10-fold cross-validation.

Table 3
Best/worst Accuracy, Kappa and number of variables cross- and per project.

Measurement	Cross-project	Individual project
Accuracy, ZeroR	0.435	0.370–0.561
Accuracy, best	0.547	0.652
Accuracy, worst	0.450	0.462
Kappa, best	0.267	0.395
Kappa, worst	0.092	0.051
Variables, best	10	6
Variables, worst	2	5

variable (Number of Files Modified resp. Number of Lines Added by Added Files Net). This is somewhat surprising but likely explained by underfitting, as there is only a low amount of commits available per project—the other projects used between two and six variables each. The best and worst accuracy for single projects, however, is greater than for cross-project classification. The range for Kappa is considerably larger and reaches higher absolute values (values between 0.21 and 0.4 are considered *fair* (Landis and Koch, 1977)). Also, the amount of variables required is noticeably lower for classification in single projects, cf. Table 3. We applied the ZeroR classifier to obtain a baseline for classification performance. Across projects, all trained models performed better than it. For individual projects, the best-performing models achieved an accuracy that was higher by 7.54% on average, compared to ZeroR. Kappa is used to measure the chance-corrected agreement between the model's predicted classifications and the true labels. It is an important measure because the number of available labels per class differ (cf. Fig. 8).

PART B As for resolving Research Question 2B, we have also examined the classification accuracy and Kappa, both cross- and

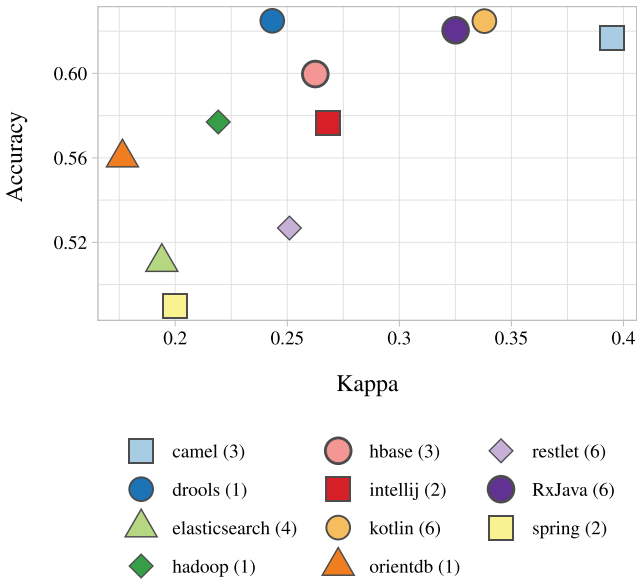

Fig. 11. Accuracy and Kappa for each project. The number in parentheses represents the number of variables used.

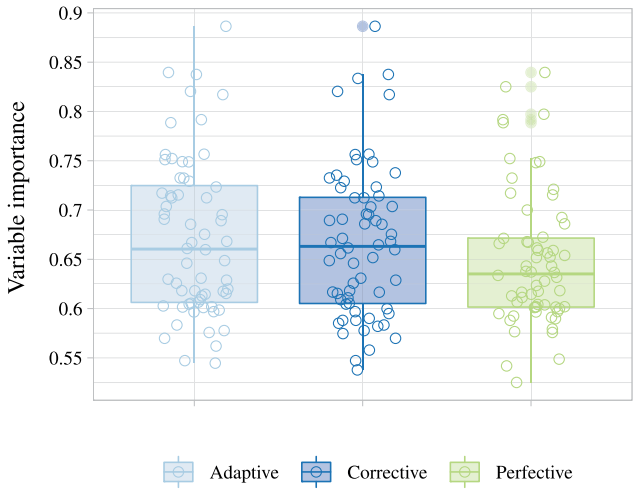

Fig. 12. Boxplots for variable importances for single projects, per maintenance activity. Slight differences for predicting each label from our extended dataset can be observed.

Table 4
Comparison of the training performance of the best cross-project models for net- and gross based datasets.

Model type	# of Vars.	Acc.	Kappa	Acc., SD	Kappa, SD
Gross	9	0.556	0.280	0.031	0.057
Net	10	0.547	0.265	0.038	0.061

Table 5
Best/worst Accuracy & Kappa, net vs. gross, cross- and single-projects.

Aggregation	Acc., net	Acc., gross	Kappa, net	Kappa, gross
max	0.547	0.556	0.265	0.280
min	0.455	0.439	0.095	0.065
avg	0.514	0.519	0.208	0.215
↑ cross-project, single-project ↓				
max	0.652	0.640	0.395	0.331
min	0.462	0.472	0.051	0.034
avg	0.565	0.566	0.234	0.232

per-project, then net- vs. gross-size (cf. Table 5). The most significant result is an accuracy of 65% and Kappa of 0.39 for a single project, using only size data. Again, we obtain better results when training models on a per-project basis, rather than attempting cross-project classification. As for the remaining results, we follow the laid out methods of Section 3.2.

Both datasets feature ten size attributes, and we have also retained the features Affected Files Ratio Net and Density for the net-dataset, yielding twelve attributes for the latter. Because of this comparatively low amount of variables, all possible model sizes were tested using an RFE-approach. The gross-based model performs best using six variables (with no further improvement using up to ten variables), while the net-based model continually improves with each added variable, thus also using all twelve available predictors. The net-based models perform insignificantly worse, see Table 4 for a complete comparison of the best models per type. The baseline for each model to outperform is set at 43.52% using ZeroR.

Given the results from Table 4, the gross-based models should be preferred, as those have a slightly lower complexity, due to the lower amount of variables, while achieving marginally better results as their net-based counterparts. With roughly 12% better performance as compared to the classification results of ZeroR, these models demonstrate their significance.

RQ3: Size and density for advancing the state of the art

- We can successfully reproduce previous results using Random forests.
- Extending previous models without selecting most important size attributes leads only to a marginal improvement.
- Observations suggest the interchangeability of $model_{\text{changes}}$ and $model_{\text{density}}$.

We were successful in reproducing the previous authors' results, achieving outcomes very similar. Involving size attributes when comparing model performances, a slight improvement of about $2-3\%$ in accuracy can be observed during training. Also, it seems that the relatively expensive to obtain change-features can be replaced by the density-features, without a decline in accuracy. By attempting further tuning and pruning and facilitating additional classifiers, we were then able to boost accuracy to up to 89% with a Kappa of 0.82.

PART A The third research question is three-part. Overall, we are interested in whether the additional size features can advance state of the art in commit classification. We could choose to establish our own baseline or, better, to reproduce the results from Levin and Yehudai (2017a). We chose the latter, as this validates their results and provides comparability. They use a compound model that is based on one or two sub-models (left and right), which are of either kind $model_{\text{keywords}}$, $model_{\text{changes}}$, or $model_{\text{combined}}$, where the latter is trained using all of their originally available features. If the compound model only use a single type of sub-model, that same model is used on the left and the right side. The left and right sides have a meaning for the classifier, so that the two configurations $\{model_{\text{A}}, model_{\text{B}}\}$ and $\{model_{\text{B}}, model_{\text{A}}\}$ are treated distinctly. The custom routine for classification using compound models chooses the left model whenever a commit uses any of the keywords the $model_{\text{keywords}}$ was trained with. For details about the models, refer to Levin and Yehudai (2017a).

While the previous authors have examined classification performance using J48-, GBM- and Random forest-classifiers, we

Table 6
Training performance of all nine compound models using the original dataset.

Model #	$model_{left}$	$model_{right}$	Accuracy	Kappa
1	Combined	Combined	0.730	0.579
2	Combined	Keywords	0.728	0.576
3	Combined	Changes	0.685	0.501
4	Keywords	Changes	0.673	0.478
5	Keywords	Combined	0.728	0.576
6	Keywords	Keywords	0.716	0.559
7	Changes	Combined	0.685	0.501
8	Changes	Keywords	0.673	0.478
9	Changes	Changes	0.527	0.248

chose to only reproduce these results using the latter for Research Question 3A, as their best performing model used that. We obtain the original dataset and perform an 85/15 percentage split, thereby withholding the smaller partition entirely from training. We then vertically split the dataset into ds_{keywords} and ds_{changes} (the entire width of the dataset is needed for the $model_{\text{combined}}$). Then, using five times repeated 10-fold cross-validation, we train the three aforementioned models separately.

First, we are interested in training performance. To assess it, we combine the numeric votes for each class by either model and select the highest (i.e., the most probable predicted label). The training results are reported in Table 6. We get similar results with regard to training performance, except for models #7 and #8, which perform significantly better (about +15% improved accuracy and additional Kappa of 0.25). As for the performance using the validation samples, our results are again similar (cf. Table 7). Surprisingly, we obtain the best results with a keywords-only compound model.

According to these results, our champion compound model is #1; it performs slightly better than model #5, which was the best model for Levin and Yehudai. When passing the 15% of previously unseen validation samples through those trained models, we get similar results. Model #5 is only insignificantly worse than model #4, and performs almost as well as that from Levin and Yehudai (76.7% with Kappa of 0.635). Again, our results may be within the margin of error.

PART B For the second part of this question, we are adding one model trained on size data. Also, $model_{\text{combined}}$ is extended with those features. The list of compound models is extended by seven additional models in the following way:

- Add a compound model for each of the other model types $\{keywords, changes, combined\}$ with the $model_{\text{density}}$ as the left model.
- Create three additional compound models, with $model_{\text{density}}$ as the right model.
- Add a purely density compound model, where the left and the right models are both of type $model_{\text{density}}$.

As for the baseline, the models need to beat 43.45% accuracy during training and 43.86% during validation, to be significant. As expected, the density-only compound model performs exactly as in the previous research question. The best such compound model consists of $model_{\text{density}}$ and $model_{\text{combined}}$, achieving 64.62% accuracy with a Kappa of 0.427 during training. The performance of the other compound models that include the $model_{\text{combined}}$ (which now spans size features) declines on average by 2–3% accuracy during training.

Using the validation samples, the best-performing compound model now is $model_{\text{keywords}}$, $model_{\text{combined}}$ with an accuracy of 75.44% and Kappa of 0.619, which denotes a slight improvement over the previous authors' results.

We seem to be able to swap out code-changes for density, when the respective other model is of type $model_{\text{keywords}}$, which might be worthwhile due to the lower cost of obtaining it. When further comparing models #9 through #12 and #13 through #16, we observe similar performance with either changes- resp. density-based models on the left, which is another hint at the interchangeability of these kinds of models. Overall, we observe a drop in performance in models #9 through #16, all of which use either changes or density as their left model (cf. Table 8 and Fig. 13).

Table 7
Validation performance of some selected compound models using the original dataset.

Model #	$model_{left}$	$model_{right}$	Accuracy	Kappa
4	Keywords	Changes	0.731	0.573
5	Keywords	Combined	0.766	0.631
6	Keywords	Keywords	0.784	0.660

Table 8
Validation performance of density- and change based models.

Model #	$model_{left}$	$model_{right}$	Accuracy	Kappa
9	Changes	Combined	0.626	0.417
10	Changes	Keywords	0.608	0.388
11	Changes	Changes	0.561	0.307
12	Changes	Density	0.532	0.245
13	Density	Combined	0.626	0.403
14	Density	Keywords	0.608	0.374
15	Density	Changes	0.561	0.290
16	Density	Density	0.532	0.225

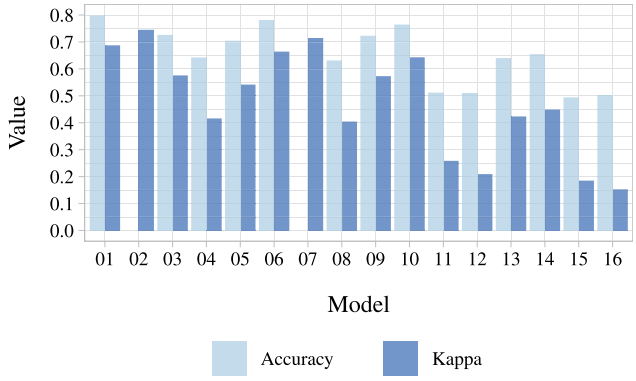

Fig. 13. Performance of all 16 compound models on the validation samples.

PART C The last part of this question is concerned with attempting to improve the performance of a model that includes all types of original and extended attributes (keywords, changes, density). We are pruning the underlying dataset in the first place, eliminating all gross-size, zero- and near-zero-variance attributes. This step reduced the dataset to less than 35 attributes. We then ran an RFE, which yielded the best model using 26 attributes that achieves an accuracy of 70.87% with a Kappa of 0.544 during training. Among the ten most important variables, we find three density attributes, four related to keywords, and three to changes.

Levin and Yehudai achieved significant results with Random forests, so we attempt to manually tune such a model with respect to its m_{try}-parameter. However, the performance did not improve compared to the best model we found using RFE. We tried a mix of classifiers on the pruned dataset and trained each with five times repeated 10-fold cross-validation. The results are reported in Table 9. The baseline set by ZeroR is an accuracy of 43.86%.

The *LogitBoost* classifier outperforms all other methods significantly, so we selected it for further tuning. The performance

Table 9
Overview of attempted methods for classification on the tuned dataset, compared to the state of the art (5 times repeated 10-fold cross-validation, results on validation samples).

Method	*R* package	Accuracy	Kappa
ZeroR (baseline)	*n/a*[a]	0.439	0.0
Levin and Yehudai (2017a) (random forest)	*n/a*[b]	0.760	0.630
LogitBoost (typical)		**0.805**	**0.690**
LogitBoost ($model_{keywords}$)	*caTools*[c]	**0.850**	**0.780**
LogitBoost ($model_{combined}$, $model_{keywords}$)		**0.891**	**0.826**
Least Squares SVM (lssvmRadial)		0.673	0.482
SVM (radial kernel)	*kernlab* (Karatzoglou et al., 2004)	0.632	0.413
SVM (linear kernel)		0.713	0.554
Neural Network	*nnet* (Venables and Ripley, 2002)	0.696	0.523
Model Averaged Neural Network (avNNet)		0.708	0.540
Gradient Boosting Machine	*gbm*[d]	0.725	0.570
eXtreme Gradient Boosting (xgbTree)	*xgboost*[e]	0.708	0.543
Linear Discriminant Analysis (lda)	*MASS* (Venables and Ripley, 2002)	0.673	0.491
Mixture Discriminant Analysis	*mda*[f]	0.708	0.540
C5.0	*C50*[g]	0.702	0.535
Naive Bayes	*naivebayes*[h]	0.544	0.253

[a]Using an own implementation done in R to predict the most common label. This results in a Kappa of 0.
[b]The authors did not disclose which package they were using, however, they used R as well.
[c]https://cran.r-project.org/web/packages/caTools/.
[d]https://cran.r-project.org/web/packages/gbm/.
[e]https://cran.r-project.org/web/packages/xgboost/.
[f]https://cran.r-project.org/web/packages/mda/.
[g]https://cran.r-project.org/web/packages/C50/.
[h]https://cran.r-project.org/web/packages/naivebayes/.

improves if we use the full combined-dataset instead of the one we just pruned. We have repeatedly run the training and report improved classification results. The most solid performing model uses $model_{keywords}$ only, achieving a stable accuracy of 85% with a typical 0.78 Kappa. The best results, however, were obtained using the compound model $model_{combined}$, $model_{keywords}$, peaking at 89.13% accuracy with a Kappa of 0.826. During subsequent runs, however, that model typically dropped to 80% with a Kappa of 0.69. This is likely explained by how the validation samples are selected between runs. Note that Kappa values between 0.61 and 0.8 are considered *substantial*, and values between 0.81 and 1.0 are considered *almost perfect* (Landis and Koch, 1977).

RQ4: Using the size and density of previous generations

- It is best to pick a principal commit that uses all available features.
- Looking back up to three commits in time improves prediction accuracy.
- Models trained for single projects profit significantly from considering preceding commits and achieve an accuracy beyond 93% with an almost perfect Kappa of 0.88.

For the fourth research question, we are examining three aspects in particular. First, we attempt to determine which features in a principal commit with appended parent generations are the most important. We find that using size features only performs worst. We then confirm that size features can replace the comparatively expensive code-changes features. Lastly, we remove keyword-features from the principal commit and report only an insignificant decline in accuracy. Second, we are interested in the amount of retained variables in previous generations of the principal commit. It appears that some datasets tend to retain more features than others and that a fair amount of these retained features are size-based net-features. Third, we examine the differences in models' performance trained across all projects and for individual projects. These results are contrasted to outcomes obtained earlier in this study, in Research Question 2. We find significant improvements for either, with new absolute champion models trained for individual projects.

Table 10
Best/worst Accuracy, Kappa and number of variables cross- and per project, involving multiple generations, for each dataset.

Measurement	Cross-project	Individual project
Accuracy, ZeroR	0.432–0.440	0.371–0.588
Accuracy, best	0.707 (C)	0.932 (D)
Accuracy, worst	0.525 (B)	0.300 (B)
Kappa, best	0.540 (C)	0.882 (D)
Kappa, worst	0.235 (B)	0.04 (C)

PART A In Research Question 4A, we attempt to find out which type of principal commit (cf. Table 1) is most suitable when attaching features of commits of previous generations to evaluate prediction performance. We built the four datasets A, B, C, and D to evaluate this (cf. Fig. 14). Refer to Section 3.3 for how these datasets were constructed. The worst-performing models are all based on dataset B, which features only size features. We confirm our previous findings that replacing out code-change-features with size features does not result in a decline in model performance. This is an important finding when comparing the engineering-cost for either set of features. Models based on the D dataset perform reliably, even though we have eliminated the keyword-features. With a slight margin over models based on the A dataset, models based on the C datasets are our champion models, for cross-project classification. Models based on the D datasets perform best for single-project classification, by a slight margin over C. C-based models contain all the features, those from Levin and Yehudai (2017b), and those from our extended dataset (Hönel, 2019).

PART B Research Question 4B is concerned with the importance of features retained from preceding generations. While we can observe a positive trend for both accuracy and Kappa, up to and including three generations, this trend seemingly becomes negative beyond that or at least stagnates. In other words, looking

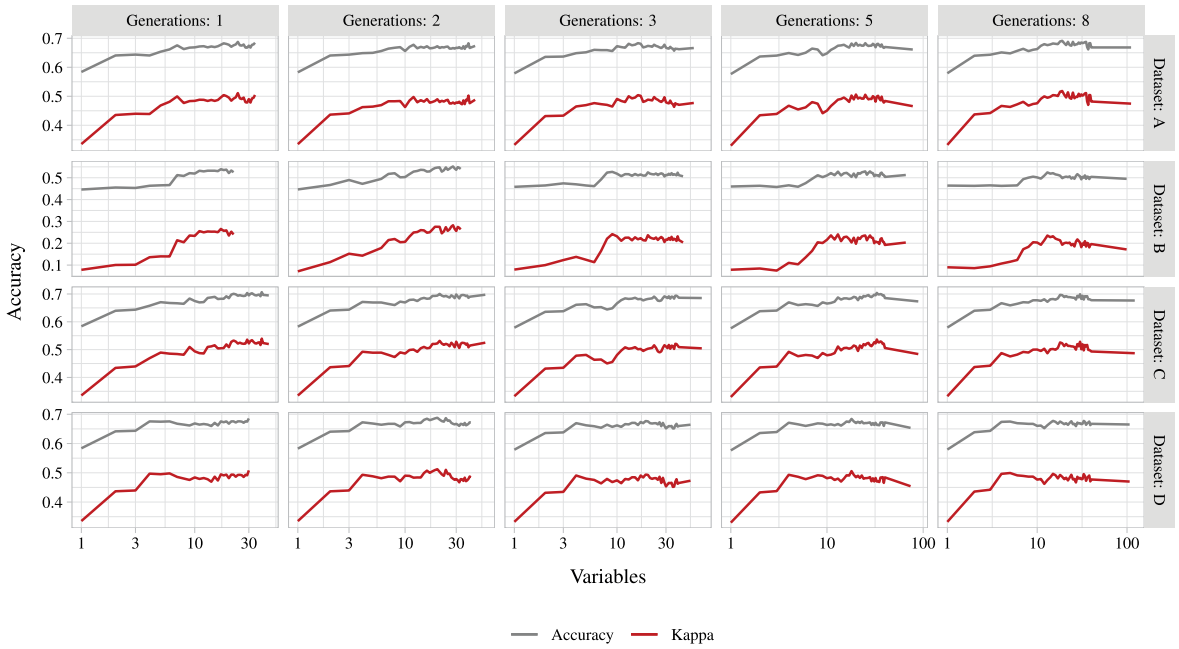

Fig. 14. Cross-project model performance including up to one, two, three, five and eight (left to right) previous generations (directly preceding commits). The four rows correspond to the four datasets A, B, C and D built for research question four.

back more than three commits to confidently predict the label of the principal commit is not of value (cf. Fig. 14).

This figure is accompanied by Fig. 15. In it, for each sub-dataset, the size (in terms of variables) of the champion-model is shown. Each model was computed using RFE. Recall that, while the principal commit may exhibit various features, the commits from preceding generations only contain size features from our extended dataset. For the B dataset, which is based on extended features only, the most size features are retained across many-generation models. The amount of retained net-variables in previous generations is fair and is approximately one-third of the total amount of variables. Models based on datasets A and C appear to retain the second-most features across generations, whereas models based on D datasets do not appear to incorporate features of previous generations well.

PART C In the last part, we contrast these results to those of Research Question 2 (cf. Table 3). Due to the vast amount of results, when introducing datasets specific to a type of principal commit and generations, we decided to aggregate them (cf. Table 10). While the ZeroR accuracy differs only slightly when comparing to the results of Table 3 to those of Table 10, we are reporting significant improvements for accuracy (+15.98% resp. +27.97%) and Kappa (+0.273 resp. +0.486), both for cross- and single-project classification for the trained champion models, respectively. We observe a decline for the worst values in single-project classification (worst accuracy −16.19% and worst Kappa −0.01). However, models trained for single projects with an accuracy beyond 93% with an *almost perfect* Kappa of 0.88 allows for commit classification with great confidence. The obtained results for accuracy and Kappa for cross-project evaluations are shown in Table 10, and those for single-projects in Fig. 16.

5. Threats to validity

Our study is threatened by concerns of both internal and external validity. In the following, we outline these threats and how we alleviate or thwart them.

INTERNAL VALIDITY We use a fundamental dataset in our study, that was previously published by Levin and Yehudai (2017b). It contains more than 1 150 manually labeled commits. The authors of said dataset took numerous actions to mitigate threats to their labeling process, such as preventing class starvation (i.e., preventing that any of the classes is underrepresented), dropping commits with low confidence, and splitting the labeling work. They report having achieved an agreement level of 94.5% with an estimated asymptotic confidence interval of [90.3%, 98.7%].

The data that we add ourselves was gathered systematically, using our tool suite *Git-Density* (Hönel, 2020). It facilitates an industry-grade component for the detection of software clones and dead code, that was extensively applied and tested in that realm for more than ten years before this study. It continues to be under active development. Detecting whitespace and comments was reliably implemented using non-greedy regular expressions, inspecting line by line, then hunk by hunk. We have added an extensive suite of unit-tests to ensure that our tool behaves correctly. Therefore, we are confident in trusting that our mined size- and density-data is correct.

As other researchers have already pointed out (Hattori and Lanza, 2008; Kirinuki et al., 2014), a commit's nature may not always be pure (*tangled* changes). We follow the classification into maintenance activities, as suggested by Mockus and Votta (2000). Those allow only to describe the nature for an entire commit. Often, changes in a commit are ambiguous, meaning that they could be seen as belonging to either of the three activities. Therefore, when labeling commits manually, some residual errors cannot be avoided. That error, to some degree, is also reflected in the models that we train.

Furthermore, having tangled changes is highly likely for merge-commits, as those are the result of merging the changes from two or more commits, as their name implies. We have rigorously excluded such commits in all of our analyses. Although, the models we built were likely to behave unpredictably for regular git-workflows that feature such commits. Merge-commits

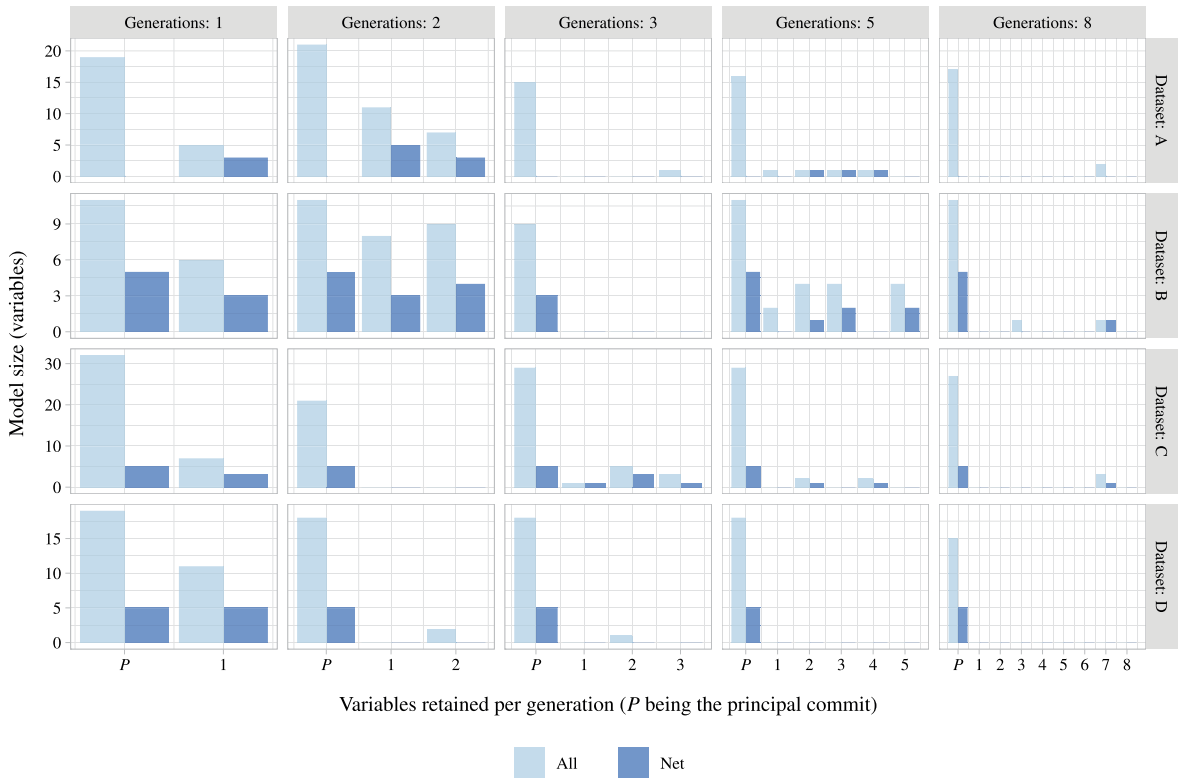

Fig. 15. Amount of retained (net-)variables in each generation, including up to eight generations. The four rows correspond to the four datasets A, B, C and D built for research question four.

need to be investigated further before they can be included in our models.

We have also found strong multi-collinearities between the many features used for Research Question 1A. Predominantly the size features come in pairs of net/gross. Those have an expected strong positive correlation. Eleven attributes can be eliminated when using a cutoff of 0.9 as the correlation coefficient. Models built using RFE perform only marginally worse with respect to accuracy and Kappa, for the benefit of reduced model complexity and a decrease in the variance of the regression coefficients.

EXTERNAL VALIDITY When training models using machine learning techniques, under- and over-fitting of such models is a concern. While the former means that a trained model cannot adequately represent the structure and patterns found in the data (and therefore performs poorly), the latter happens when the data is too small or the model too large, and the model contains more parameters than are justified by the data. Our attempt at mitigating either case was to apply RFE. In our results, we encountered both cases, under- and over-fitting. In RFE, the impact and importance of each variable for a model are measured. In our analyses, we ran the full extent of RFE, using between one and all available variables. Then, a many times repeated cross-validation was performed, always withholding a certain amount of data entirely from training. This resampling mechanism can assure external validity to a high degree. Under- and over-fitting was always observed in the context of models trained on individual projects. This is not surprising, as the amount of data available per project is significantly less. Also, due to that shortage, some features only exhibited a very low or no variance any longer, so that they had to be eliminated. The achieved results concerning individual projects should, therefore, be regarded as less generalizable compared to those for cross-projects. The results for single projects, however, demonstrate that models trained on them can approximate the nature of their commits with higher absolute accuracy and Kappa.

Furthermore, the eleven projects in the datasets were all open source projects. While others claim that the evolutionary patterns in such projects and closed software are the same (Herraiz et al., 2006), we can neither support this claim nor generalize our results for closed software at this point. The eleven projects however represented a broader spectrum of software types: e.g., *RxJava* and *Spring-Framework* are libraries, *Hadoop* is an enterprise distributed storage solution, *Kotlin* is a programming language, *IntelliJ Idea* is a fully-featured development environment for desktop, and *ElasticSearch* and *OrientDb* are enterprise-grade search- and database-engines, respectively. With a somewhat higher level of confidence, we expect high generalizability of our results for software that falls into this spectrum.

6. Related work

The related work can be subdivided into three categories. First, studies that present or examine various attempts to quantify changes in commits. Second, earlier similar studies to this one, that we in part reproduce and build on, or use directly. Third, studies that use the size of commits to solve a concrete problem.

6.1. Quantification of change

Related studies found various ways to quantify the changes a commit induces. As an early method, function points were suggested by Albrecht (1979), in 1979. Rather than counting LOC, function points were meant to provide a way to quantify the size of a program as a functional size measurement. Lin and Gustafson (1988), develop an application that counts the number of changed, added, and deleted statements in COBOL applications,

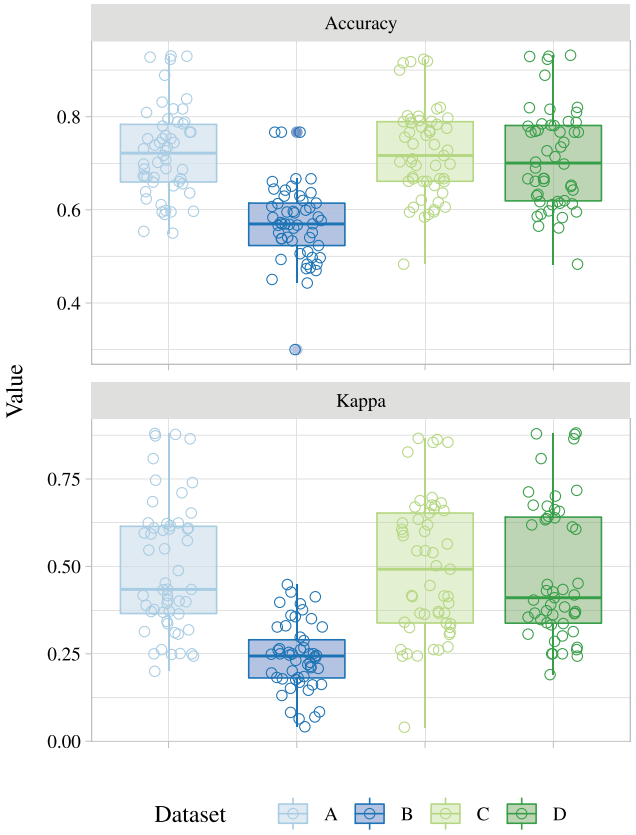

Fig. 16. Aggregated results of the best models trained on individual projects for attempting an RFE in each dataset, across many generations.

to quantify the size of a change. Jackson et al. (1994), propose a difference algorithm that carves out semantic changes. Fluri et al. (2007), apply tree-differencing to distill changes in the Abstract Syntax Trees (AST) between two versions of a software system, thus quantifying the syntactical changes in terms of statements and expressions. Several others, e.g., Fischer et al. (2003) and Hindle et al. (2009), attempted to classify commits based on their keywords or other associated messages, such as those from bug-tracking systems. Characterizing commits by reverse-engineering the stereotype of affected methods was demonstrated by Dragan et al. (2011). While only indirectly measuring size, that approach provides a valuable insight into the nature of a commit and its effects on the system's architecture.

6.2. Reproducibility and relation to previous studies

Most related work classifies the maintenance activities as *adaptive*, *corrective*, or *perfective*, as proposed by Swanson (1976), and further discussed by Mockus and Votta (2000). Others introduce additional or more distinctive categories. Our work extends and compares to the work of Levin and Yehudai (2017a). Therefore, we adopt the three originally proposed categories (labels) without alterations as they do. We apply similar validation methods, focusing on prediction accuracy and Cohen's Kappa (Cohen, 1960), for measuring the agreement of our proposed models and the true labels.

Levin and Yehudai (2017a), use a manually labeled dataset (Levin and Yehudai, 2017b), containing 1 151 commits as an underlying ground truth. They report a well-respectable classification model based on a hybrid classifier that exploits commit keywords and source code changes. The latter is obtained by the distiller from Fluri et al. (2007). We are reproducing, reusing, and extending their work.

We are particularly interested in addressing questions that were answered previously without taking into account the net size of changes. This is of interest, as some studies report strong correlations between the size and the nature of a commit. Hindle et al. (2008) are looking in particular at large commits, where they derive the size of a commit by the number of files included in it. They find that large commits tend to be perfective, while small commits are often corrective. That work is especially relevant in conjunction with another study by Herraiz et al. (2006), that found that it does not matter whether the size is defined via the number of files or the number of lines of code.

6.3. Studies that rely on size

Furthermore, Herraiz et al. (2006) propose size classifications of commits based on the number of files changed in the commits. While we focus on extracting size features, we also possess the means to count the affected files. Further, using density, we can reduce those counts by files that were not affected in actuality. Using this new insight, we can put some of our results in relation to the findings of Herraiz et al.. An additional study by Alali et al. (2008) that attempted to categorize commits of nine open source projects by their size quantified using the number of files, the number of lines, and the number of hunks affected, reports a weak correlation between the first two, and a strong correlation between the last two measurements. Additionally, they confirm that comparatively small changes are afflicted with correcting bugs.

Hattori and Lanza (2008), build upon the work from Herraiz et al. and use the number of affected files to determine the nature of commits. By attempting to specify size categories, such as *tiny*, *medium*, or *large*, they report that the majority of tiny commits, i.e., affecting five files or less, are not related to adaptive development activities. Rather, those changes are perfective or corrective. Their study gives us further incentive to examine the relationship between notions of size based on either affected files or lines of code.

Purushothaman and Perry (2005), address the problem of classifying small corrective changes by focusing on the properties of the changes rather than the properties of the code itself. They point out that change-size is a significant fault predictor. They raise awareness for risk assessment and risk management, as the risk associated with small changes tends to be small, too. With a more concise notion of net change size, or at least a better approximation, we are contrasting our findings against changes that were previously considered to be small.

7. Conclusions

We have demonstrated that considering additional properties of the contents of a commit can significantly improve classification performance. We have shown that the *density* of a commit is a significant predictor. By reproducing the results of others and putting our studies into relation to theirs, we have made our work comparable.

Qualitatively, the density of a commit allows a more fine-grained separation into maintenance activities than raw lines of code, as our results for RQ 1 show. The significant deviations between net- and gross-size (RQ 2) make clear, that the density has the potential to unveil changes that raw lines of code would hide, such as global renames or the incorporation of large portions of code. On the other hand, the density may hide changes not reflected in the change in functionality of the underlying source code, such as changes to the documentation. For models using

density for the classification into the chosen three maintenance categories however, the density is suitable (RQ 3). That also supports the results from the last research question (RQ 4). If the density allows assigning a commit to some maintenance activity with great confidence, then the density of preceding commits carries weight that can be exploited for further improving commit classifiers.

Earlier studies suggested that measuring the size of a commit by counting either its affected files or lines of code are equal, but we find this not to be true. There are significant differences between those and more subtle differences when differentiating net- and gross-size. We are also observing a shift in maintenance activities for high-density commits away from *corrective* and towards *perfective*.

When studied thoroughly, it becomes apparent that there is a significant difference between gross- and net notions of the size of a commit, with the latter being a more important predictor of a commit's nature. Models based on size features consistently outperform the baseline set by us. Density is a sound classifier, especially when trained on individual projects. Cross-project, the achievable accuracy is much above the baseline as well.

The third part of this study is an attempt to reproduce previous results and to extend them. We were able to comprehend previous work and extend it with our data and methods in a way that can boost classification by another 4–13% (towards 90%) with a *significant* Kappa, using models that involve the density of commits.

Going beyond the attempts made by others, we exploit relational information from our extended dataset in the last part. In it, we demonstrate that preceding generations of commits that are solely exhibiting size features are boosting commit classification rates even further, up to 93% for single projects, with an almost perfect Kappa. This confirmed a conjecture of ours that maintenance activities, especially on designated branches, follow evolutionary patterns that are typically met during software development processes.

Our results demonstrate an improvement of the state of the art in automated commit classification. Beyond that, we contribute the following:

Git-Density is an open-source suite of tools for analyzing git-repositories (Hönel, 2020). It was initially built for extracting size- and especially density properties of commits' associated source code but has been expanded ever since.

Extended Dataset The dataset used for all of our experiments is publicly available (Hönel, 2019) under the terms of open access. Refer to Section 2.3 for detailed descriptions of contained features.

R Experiments The experiments were conducted using the R statistical environment (R Core Team, 2017). We have performed extensive analyses on the published data and to strengthen collaborative scientific work and to aid the reproducibility of our results, we share the source code for all experiments on GitHub.[4]

8. Future work

We find that the size of a commit, while a significant predictor for maintenance activities, is a computationally cheap and convenient measure to use. We plan to package our classifier into a tool that can be used to automatically classify commits and use it to perform a field study. We aim to apply the classifier across a number of open source projects and use the classification information to support tasks, such as fault prediction, or to characterize the evolution process and aspects of it automatically.

Dimensionality reduction techniques could help to reduce the number of attributes, since we have few samples. Our attempts to visually analyze the data using the t-SNE (Maaten and Hinton, 2008) algorithm were not fruitful. During the experiments, we also attempted to reduce the number of dimensions using a Principal Component Analysis (PCA) (Pearson, 1901). However, this did not yield significant improvements.

We have further identified that structural and primarily relational properties of commits had not previously been considered, to the best of our knowledge. Commits are assigned to branches in a source tree, they have predecessors and successors, and there are special kinds of commits, such as merge-commits. Often, branches serve a single purpose, and adjacent commits may share a common nature, or their nature follows a logical pattern. Our tool *Git-Density* (Hönel, 2020) was extracting such properties, so we used them in research question four.

However, now that we found previous generations of commits to be useful, more potential extensions open up. The first that comes to mind are Hidden Markov Models (HMM, Baum and Petrie (1966)). Such models find the most likely path of hidden states (here: commits' labels) through a series of observable events (commits' features). But building said models would require us to label adjacent commits from our extended dataset manually. In their simplest form, HMMs are univariate models that would have a questionable application, given the vast amounts of features we were evaluating. Also, numeric data need to be discretized into events, which need to have a probability of occurrence assigned.

Another potential approach for taking multiple generations and multiple features into account would be to apply Bayes' theorem. It is built around conditional probabilities and can be extended efficiently to operate on occurrences of more than just one event. Such an example, given the two conditional events $\{B, C\}$ is given in Eq. (8), the example may be extended to accommodate an arbitrary number of events. The advantage of using a Bayes approach lies in its simplicity and the low effort required to set it up. It would, however, require discretized events from numeric data, too.

$$P(A|B,C) = \frac{P(B|C,A) \times P(C|A) \times P(A)}{P(B|C) \times P(C)} \tag{8}$$

One more substantial extension of our approach could be the application of multivariate HMMs. While these models tend to be more complex, they support a probability distribution for each feature of the current observation. We conjecture that such models would likely deliver high classification rates, or that they could be used in an ensemble- or meta-learner, to further stabilize prediction accuracy.

Lastly, we have extracted the dwell times between adjacent commits in our extended dataset, but have not yet made use of them. Such information can be exploited in Hidden semi-Markov Models (HsMM, Yu (2010)). These models allow for a separate probability distribution of dwell times in each state, and we surmise that this feature carries some weight.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We would like to thank the anonymous reviewers for their invaluable comments, which helped us to further improve this work. We would also like to thank the Linnaeus University Centre for Data Intensive Sciences (DiSA) and Applications and the Swedish Research School of Management and IT (MIT).

[4] https://github.com/MrShoenel/density-paper-2019-R-experiments.

References

Abran, A., Moore, J.W., Bourque, P., Dupuis, R., Tripp, L., Software engineering body of knowledge, IEEE Computer Society, Angela Burgess.

Alali, A., Kagdi, H., Maletic, J.I., 2008. What's a typical commit? a characterization of open source software repositories. In: 2008 16th IEEE International Conference on Program Comprehension. IEEE, pp. 182–191.

Albrecht, A.J., 1979. Measuring application development productivity. In: Proc. Joint Share, Guide, and IBM Application Development Symposium. 1979.

Albrecht, A.J., Gaffney, J.E., 1983. Software function, source lines of code, and development effort prediction: a software science validation. IEEE Trans. Softw. Eng. (6), 639–648.

Alexander, C., 1977. A Pattern Language: Towns, Buildings, Construction. Oxford university press.

Ambroise, C., McLachlan, G.J., 2002. Selection bias in gene extraction on the basis of microarray gene-expression data. Proc. Natl. Acad. Sci. 99 (10), 6562–6566.

Baum, L.E., Petrie, T., 1966. Statistical inference for probabilistic functions of finite state Markov chains. Ann. Math. Stat. 37 (6), 1554–1563.

Bell, R.M., Ostrand, T.J., Weyuker, E.J., 2011. Does measuring code change improve fault prediction? In: Proceedings of the 7th International Conference on Predictive Models in Software Engineering. pp. 1–8.

Breiman, L., 2001. Random forests. Mach. Learn. 45 (1), 5–32.

Chen, T., He, T., Benesty, M., et al., 2015. Xgboost: extreme gradient boosting. pp. 1–4, R package version 0.4-2.

Cohen, J., 1960. A coefficient of agreement for nominal scales. Educ. Psychol. Meas. 20 (1), 37–46.

Dragan, N., Collard, M.L., Hammad, M., Maletic, J.I., 2011. Using stereotypes to help characterize commits. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM). IEEE, pp. 520–523.

Fernández-Delgado, M., Cernadas, E., Barro, S., Amorim, D., 2014. Do we need hundreds of classifiers to solve real world classification problems?. J. Mach. Learn. Res. 15 (1), 3133–3181.

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, pp. 23–32.

Fluri, B., Gall, H.C., 2006. Classifying change types for qualifying change couplings. In: 14th IEEE International Conference on Program Comprehension (ICPC'06). IEEE, pp. 35–45.

Fluri, B., Giger, E., Gall, H.C., 2008. Discovering patterns of change types. In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. IEEE, pp. 463–466.

Fluri, B., Wuersch, M., Pinzger, M., Gall, H., 2007. Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Trans. Softw. Eng. 33 (11), 725–743.

Friedman, J.H., 2002. Stochastic gradient boosting. Comput. Statist. Data Anal. 38 (4), 367–378.

Hanley, J.A., McNeil, B.J., 1982. The meaning and use of the area under a Receiver Operating Characteristic (ROC) curve. Radiology 143 (1), 29–36.

Hattori, L.P., Lanza, M., 2008. On the nature of commits. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, pp. III–63.

Herraiz, I., Robles, G., González-Barahona, J.M., Capiluppi, A., Ramil, J.F., 2006. Comparison between SLOCs and number of files as size metrics for software evolution analysis. In: Conference on Software Maintenance and Reengineering (CSMR'06). IEEE, p. 8.

Hindle, A., German, D.M., Godfrey, M.W., Holt, R.C., 2009. Automatic classication of large changes into maintenance categories. In: IEEE 17th International Conference on Program Comprehension 2009 (ICPC'09). IEEE, pp. 30–39.

Hindle, A., German, D.M., Holt, R., 2008. What do large commits tell us?: a taxonomical study of large commits. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories. ACM, pp. 99–108.

Hönel, S., 2019. 359, 569 commits with source code density; 1149 commits of which have software maintenance activity labels (adaptive, corrective, perfective) [Data set]. http://dx.doi.org/10.5281/zenodo.2590519.

Hönel, S., 2020. Git Density 2020.2: Analyze Git Repositories To Extract the Source Code Density and Other Commit Properties. http://dx.doi.org/10.5281/zenodo.2565238.

Hönel, S., Ericsson, M., Löwe, W., Wingkvist, A., 2018. A changeset-based approach to assess source code density and developer efficacy. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion). IEEE, pp. 220–221.

Jackson, D., Ladd, D.A., et al., 1994. Semantic diff: A tool for summarizing the effects of modifications. In: Proceedings of the International Conference on Software Maintenance, Vol. 94. pp. 243–252.

Karatzoglou, A., Smola, A., Hornik, K., Zeileis, A., 2004. Kernlab – An S4 Package for Kernel Methods in R. J. Stat. Softw. 11 (9), 1–20, URL http://www.jstatsoft.org/v11/i09/.

Kirinuki, H., Higo, Y., Hotta, K., Kusumoto, S., 2014. Hey! are you committing tangled changes?. In: Proceedings of the 22nd International Conference on Program Comprehension. ACM, pp. 262–265.

Kohonen, T., 1995. Learning vector quantization. In: Self-Organizing Maps. Springer, pp. 175–189.

Kuhn, M., 2008. Building Predictive Models in R using the caret Package. J. Stat. Softw. (ISSN: 1548-7660) 28 (5), 1–26. http://dx.doi.org/10.18637/jss.v028.i05, Articles.

Kuhn, A., Ducasse, S., Gírba, T., 2007. Semantic clustering: Identifying topics in source code. Inf. Softw. Technol. 49 (3), 230–243.

Landis, J.R., Koch, G.G., 1977. An application of hierarchical Kappa-type statistics in the assessment of majority agreement among multiple observers. Biometrics 363–374.

Lehman, M.M., Belady, L.A., 1985. Program Evolution: Processes of Software Change. Academic Press Professional, Inc..

Leszak, M., Perry, D.E., Stoll, D., 2002. Classification and evaluation of defects in a project retrospective. J. Syst. Softw. 61 (3), 173–187.

Levin, S., Yehudai, A., 2016. Using temporal and semantic developer-level information to predict maintenance activity profiles. In: Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 463–467.

Levin, S., Yehudai, A., 2017a. Boosting automatic commit classification into maintenance activities by utilizing source code changes. In: Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering. ACM, pp. 97–106.

Levin, S., Yehudai, A., 2017b. 1151 commits with software maintenance activity labels (corrective, perfective, adaptive) [Data set]. http://dx.doi.org/10.5281/zenodo.835534.

Lientz, B.P., Swanson, E.B., Tompkins, G.E., 1978. Characteristics of application software maintenance. Commun. ACM 21 (6), 466–471.

Lin, I.-H., Gustafson, D.A., 1988. Classifying software maintenance. In: Proceedings. Conference on Software Maintenance, 1988. IEEE, pp. 241–247.

Maaten, L.v.d., Hinton, G., 2008. Visualizing data using t-SNE. J. Mach. Learn. Res. 9 (Nov), 2579–2605.

Mockus, A., Votta, L.G., 2000. Identifying reasons for software changes using historic databases. In: Proceedings of the International Conference on Software Maintenance. pp. 120–130.

Pearson, K., 1901. LIII. On lines and planes of closest fit to systems of points in space. Lond. Edinburgh Dublin Phil. Mag. J. Sci. 2 (11), 559–572.

Pícha, P., Brada, P., Ramsauer, R., Mauerer, W., 2017. Towards architects activity detection through a common model for project pattern analysis. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW). IEEE, pp. 175–178.

Purushothaman, R., Perry, D.E., 2005. Toward understanding the rhetoric of small source code changes. IEEE Trans. Softw. Eng. 31 (6), 511–526.

R Core Team, 2017. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria, URL https://www.R-project.org/.

Rising, L., Janoff, N.S., 2000. The Scrum software development process for small teams. IEEE Softw. 17 (4), 26–32.

Svetnik, V., Liaw, A., Tong, C., Wang, T., 2004. Application of Breiman's random forest to modeling structure–activity relationships of pharmaceutical molecules. In: International Workshop on Multiple Classifier Systems. Springer, pp. 334–343.

Swanson, E.B., 1976. The dimensions of maintenance. In: Proceedings of the 2nd International Conference on Software Engineering. IEEE Computer Society Press, pp. 492–497.

Venables, W.N., Ripley, B.D., 2002. Modern Applied Statistics with S, fourth ed. Springer, New York, ISBN: 0-387-95457-0, URL http://www.stats.ox.ac.uk/pub/MASS4.

Weinberg, G.M., 1983. Kill that code. Infosystems 30 (8), 48–49.

Yeo, I.-K., Johnson, R.A., 2000. A new family of power transformations to improve normality or symmetry. Biometrika 87 (4), 954–959.

Yu, S.-Z., 2010. Hidden Semi-Markov models. Artificial Intelligence 174 (2), 215–243.

Sebastian Hönel is a doctoral student at the Department of Computer Science and Media Technology of Linnaeus University, Sweden. He is also a member of the Swedish Research School of Management and IT (MIT). His main interests lie in software repository mining, and automated software maintenance and information quality, with the goal of improving overall software quality. Prior to becoming a PhD student, he gained about three years of industry experience. He spent an additional year as working student at SAP Research in Dresden, Germany.

Dr. Morgan Ericsson is an associate professor of computer science at Linnaeus University. His main research interests include software quality and metrics, empirical software engineering, and data mining for software engineering data.

Prof. Dr. Welf Löwe holds the chair in software technology at Linnaeus University in Växjö (Sweden) since 2002. Before, he studied at TU Dresden (Germany), received a PhD from TH Karlsruhe (Germany), was postdoc at ICSE Berkeley (CA, USA), and assistant professor at TH Karlsruhe. He is interested in technology for software construction, analysis, optimization, and runtime support. He is also co-founder of Softwerk, Aimo, and DueDive.

Dr. Anna Wingkvist is an Associate Professor in Computer Science at Linnaeus University, Sweden. Her academic background is in information systems development, methodological and research methods reasoning, and project management. Since completing her PhD in 2009, her scientific interest and publications are mainly in the information quality domain. In 2011, she was awarded a Marie Curie Fellowship research grant.