Can Software Project Maturity Be Accurately Predicted Using Internal Source Code Metrics?

Mark Grechanik[1], Nitin Prabhu[1], Daniel Graham[2], Denys Poshyvanyk[2], and Mohak Shah[3]

[1] University of Illinois at Chicago
Chicago, IL 60607
USA
{drmark,nprabh3}@uic.edu
[2] College of William and Mary
Williamsburg, VA 23185
{dggraham,denys}@cs.wm.edu
[3] Bosch Research
Palo Alto, CA 94568
Mohak.Shah@us.bosch.com

Abstract. Predicting a *level of maturity (LoM)* of a software project is important for multiple reasons including planning resource allocation, evaluating the cost, and suggesting delivery dates for software applications. It is not clear how well LoM can be actually predicted – mixed results are reported that are based on studying small numbers of subject software applications and internal software metrics. Thus, a fundamental problem and question of software engineering is if LoM can be accurately predicted using internal software metrics alone?
We reformulated this problem as a supervised machine learning problem to verify if internal software metrics, collectively, are good predictors of software quality. To answer this question, we conducted a large-scale empirical study with 3,392 open-source projects using six different classifiers. Further, our contribution is that it is the first use of feature selection algorithms to determine a subset of these metrics from the exponential number of their combinations that are likely to indicate the LoM for software projects. Our results demonstrate that the accuracy of LoM prediction using the metrics is below 61% with Cohen's and Shah's $\kappa << 0.1$ leading us to suggest that comprehensive sets of internal software metrics alone cannot be used to predict LoM in general. In addition, using a backward elimination algorithm for feature location, we compute top ten most representative software metrics for predicting LoM from a total of 90 software metrics.

1 Introduction

Predicting various aspects of software quality and the *levels of maturity (LoM)* of software applications is important for multiple reasons including planning resource allocation for software development and maintenance, evaluating the cost, and suggesting delivery dates for software applications [8, 12, 13, 18, 27, 33, 45]. The essence of many machine learning approaches to predict certain characteristics of software is to obtain a

model that maps software applications to classes that represent levels of software quality or maturity. In these approaches, software applications are represented as vectors of features, where features are measurements of some attributes of these applications. For example, LoM classes can be represented by alpha, beta, and other testing phases of software release life cycle and features may include the frequencies of language keyword usages for software applications that are assigned to these LoM classes. A classifier can be trained on a subset of feature vectors obtained from some software applications, where these applications are assigned to the LoM classes using information from a software repository. Once a model is learned as a result of training the classifier, it can be used to predict the LoM class for a software application.

Software metrics is a term that specifies a collection of measurements that describe software-related activities [15]. Many studies focused on predicting the level of software quality by using aggregated internal software metrics that represent different measurements of the source code of software applications [14, 17, 36, 42, 44]. However, to the best of our knowledge there is no research into a fundamental question of software engineering – *can LoM be accurately predicted using internal software metrics that are obtained from the source code of software applications?*

In this paper, we reformulated this problem as a as a supervised machine learning problem to verify if collectively 90 software metrics can predict software LoM level with a high degree of accuracy. Further, our goal is to determine if a subset of 90 software metrics can predict LoM more effectively, since it is likely that not all software metrics contribute equally to LoM. Unfortunately, to investigate all subsets of $\approx 2^{90}$ elements of the superset of all software metrics is infeasible. One of our main contributions is to use feature selection algorithms to determine if a subset of these metrics can do so to guard against noise and irrelevant attributes [20].

We make the following contributions. First, we conducted a large-scale empirical study with 3,392 open-source projects using six different classifiers. Next, we used different algorithms for selecting subsets of features to eliminate the noise that non-representative software metrics add and to determine what subset of these metrics best represent the LoM for software projects. We believe that this is the first empirical study that addressed a fundamental question of determining LoM for software projected using internal source code metrics alone. To the best of our knowledge there is no other work that addressed this question on a large scale and applied feature selection algorithms to determine a manageable subset of internal source code metrics that can summarily indicate a LoM. Our results show that the accuracy of software LoM level prediction reaches $\approx 61\%$ with Cohen's and Shah's $\kappa << 0.1$ leading us to suggest that internal, source code software metrics alone cannot serve as accurate predictors of LoM in general. All workflows and data files are available from the project's website `http://www.cs.uic.edu/~drmark/prime.htm`.

2 Problem Statement

Predicting different aspects of software quality and maturity with high precision using some attributes is a problem of great importance. Stakeholders can take timely corrective actions if an accurate prediction is issued that some quality aspects of a software

application will worsen using a set of software metrics that are collected during development of this application. Taking corrective actions when software applications are still under development results in significantly lower costs and faster times to market, enabling stakeholders to reap huge economic benefits. Predicting LoM for software enables stakeholders to plan software release events, which is very important for the economic health of software projects. It is no wonder that predicting LoM draws significant attention of both academia and industry – companies that claim that they can predict different aspects of software quality can gain competitive market advantage, since they can systematically control production of software [25].

Selecting attributes from which the LoM of software applications can be predicted with a high degree of accuracy is a fundamental problem of software engineering. Various software metrics are used as attributes for predicting different aspects of the quality of software applications. As we show in our analysis of the related work in Section 5, a sheer majority of approaches follow the same methodology that consists of three steps: 1)some software or process metrics are selected ad-hoc as attributes, 2)a small set of software applications is selected with assigned values from some class attribute that describes the level of quality, and 3)a classifier is trained using these attributes to learn a prediction model. Different results are reported showing good accuracy of learned models where it reaches as high as over 90% for selected applications.

Unfortunately, many unanswered questions surround this methodology. Can internal, source code software metrics *alone* be used to build prediction models with a high degree of precision for the LoM of software applications? Do all software metrics contribute equally to building accurate predictors of software LoM? Do the learned models have good precision in predicting LoM of software applications that are developed by different programmers across many different domains? How sensitive these models are to the parameters of the learning algorithms? What is a minimal subset of attributes that can characterize LoM with a good level of precision? Answering these questions is the main goal of this paper.

In this paper we address the following question: can software LoM be accurately predicted using source code software metrics alone? Collecting various software metrics from the source code of applications is easy; determining subsets of these metrics that are useful to build a good predictor is very difficult. This problem is an instance of a bigger problem in ML, namely a supervised ML problem to verify if collectively these software metrics are predictors of software LoM. This problem focuses on constructing and selecting subsets of attributes that are important for creating high-quality prediction models [20]. The root of this problem is in difficulty of determining individual predictive powers for attributes using a specific ML algorithm. For a small number of attributes it is possible to train classifiers with all the subsets from the powerset of these attributes, however, this brute-force solution quickly becomes computationally prohibitive even when the number of attributes increases to couple of dozens. In this paper, we deal with over 90 different software metrics, and trying all their subsets is not feasible. Further, we perform feature selection to determine if a subset of these metrics can do so to guard against noise and irrelevant attributes.

The problem is not only in training predictors using all subsets of attributes – different ML algorithms may perform differently, so these ML algorithms should be used

to build predictors using subsets of attributes. Parameter spaces of these ML algorithms should be explored as part of sensitivity analysis to check to see if varying values of these parameters significantly changes the precision of the predictor. These and many other variables will be explored as part of addressing our problem.

3 Experiments

In this section, we describe variables, explain our methodology, provide background on ML algorithms for learning models and on algorithms for computing an optimal set of software metrics for learning predictors, describe our experimental design, and discuss threats to validity.

3.1 Measuring LoM of Software Projects

The quality of a software application is a set of measures that describe how this application is expected to fulfill some need and meet some standards [19, pages 15-37]. There are many different measures including but not limited to correctness, reliability, performance, robustness, maintainability, and usability. *Phases of testing* (e.g., alpha, beta, release candidate) are widely used in software release life cycle to indicate levels of maturity of software. The concept of phase of testing was introduced in 1950s and widely used in software development to indicate certain aspects of the level of maturity and quality of software applications [6, pages 515-516] [19, page 400] [22]. Since it is difficult to obtain other indicators of the LoM for thousands of open-source software project, we use the phase of testing that is specified for each project as an approximation for its LoM.

There are three main phases of testing. When a software application is built and tested within a development organization, this application is usually assigned the *alpha phase*, where testing is performed by dedicated teams within this organization. Alpha phase indicates the lowest level of maturity for software applications. During the work on the alpha version of the application, software engineers fix bugs and add and change features (i.e., units or functionality) among many other things. When the collective quality of the application improves to a certain level, stakeholders assign this application to the *beta phase*, where the application is shipped to selected customers who will use it and give detailed feedback to the development organization. At some point, software engineers further improve the quality of the application, and stakeholders assign it the *production phase*. Depending on the organization, LoM may include other intermediate phases besides these three major ones. Of course, no single external software metric can serve as a strong measure of software quality, but different metrics can reflect on certain integral aspects of software quality. A testing phase is the quality aspect of software applications that is reflected in their LoM.

3.2 Methodology

A main purpose of our methodology is to increase the confidence in the results of experiments with which we attempt to answer the main research question in this paper

– can the LoM level of a software application be accurately predicted using internal software metrics alone? We answer this question through experimentation by aligning our methodology with the guidelines for statistical tests in software engineering and machine learning [3, 24]. Our goal is to collect large and highly representative samples of data when applying different approaches, perform statistical tests on these samples, and draw conclusions from these tests. Our main objective is to reduce a bias in our experimental design to minimum.

In most existing studies [21], experimental bias comes from different sources: a small number of subject applications, stakeholders who develop and maintain applications in some peculiar ways, ML algorithms that use a specific representation of the model that benefits from certain metrics, limited software metrics that reflect a certain aspect of software, and skewed representations of software LoM levels. We design our methodology to address these biases.

Selection of ML algorithms is critical for generalization of the results of this experiment. Different ML algorithms employ different function spaces for mapping inputs to the (expected) outputs. We explore various learning paradigms, and hence, a variety of function or classifier spaces so as to not introduce dependency of a choice of space on the results. We choose a representative set of ML algorithms ranging from simple (naive) bayesian formulation to more complex kernel spaces for the majority of different representations of learners [11].

Since one of our goal is to determine best performances of learned models for subsets of data, we experiment using ML algorithms with multiclass data (i.e., all applications in a single dataset with Alpha, Beta, Production LoM phases) and with binary classification (i.e., application from any two classes). Often, binary classification yields better classifier performance results when compared with multiclass performance [1, 43]. Therefore, we experiment with four subsets of the data: Alpha/Beta, Alpha/Production, Beta/Production, Alpha/Beta/Production.

With four subsets of application's software metrics, we carry out two main experiments: the first with six ML algorithms using three subsets of software metrics based on their weights and the second one with six ML algorithms using backward elimination. In total, we perform 96 experiments $= 4 \times 6 \times 3 + 4 \times 6$. We carry out experiments using RapidMiner, an open-source tool that implements various ML and data mining techniques such as data transformation, feature selection, classification and model evaluation [32]. We discarded applications that were categorized ambiguously (e.g., an application that is assigned both Alpha and Beta phases). All experiments were carried out using five-fold cross validation and stratified sampling, a standard methodology for ML algorithms. We do not report execution times for learning models, since performance measures of RapidMiner do not address research questions that we formulate here.

3.3 Why We Choose ML Algorithms

Since our goal is to provide evidence that may show that the LoM level can be accurately predicted, a natural question is why we did not use the entire universe of all statistical methods instead of concentrating on selected few. More specifically, are the ML methods that we use in some way superior to some other statistical methods to answer our research question?

An answer is that the techniques that we chose are superset of a lot of statistical approaches and being data-driven scale better. For instance, decision based classifiers are supersets of the classes of conjunctions and disjunctions and decision trees that approximate disjunctions of conjunctions, and hence can approximate quite a few linear combinations. *Support Vector Machines (SVM)* are of course generalizations of linear combinations to reproducing kernel Hilbert space (i.e., generalizations of Euclidean spaces). Many a data-driven approaches are generalizations of univariate/multi-variate statistical approaches and have shown both to scale and give state-of-the-art performances. By selecting a set of ML algorithms, we intend to cover a variety of these classifier spaces.

3.4 Background

In this section, we provide background on a feature selection method called backward elimination that we use in our experiments to determine a minimal subset of software metrics with which a maximum performance can be achieved, and on Cohen's and Shah's κ-measures that we use in this paper as a dependent variable to measure the performance of ML algorithms.

Backward Elimination Backward elimination is a popular approach as part of the wrapper methodology that considers an ML algorithm as a black-box and uses it to assess the usefulness of subsets of features [29,30]. The essence of backward elimination, which is considered the best-performing approach for automatically selecting subsets of features with a high predictive power, is in a search algorithm that performs a direct objective optimization by maximizing the goodness-of-fit (i.e., the performance) and minimzing the number of features that are used as the input to this ML algorithm. As the name of this approach suggests, it starts with the full set of features (i.e., software metrics in our case) and uses a measure of the goodness-of-fit to eliminate some features from the input that make the lowest contribution to this goodness-of-fit. The process repeats until a minimum subset of features is found. Backward elimination approach is computationally intensive; for example, one of our experiments using this approach took 43 hours to complete using Intel Xeon Dual Core CPU X5680 at 3.33GHz and RAM 24Gb. However, using backward elimination gives us a high level of confidence that we did not overlook certain subsets of software metrics that may lead to high accuracy of prediction of LoM.

Cohen's κ-measure Often, precision is not enough to determine if a predictor is good enough. While 50% precision is essentially a random choice, does 60% or 70% precision makes it a good predictive model? To answer this question, Cohen's κ measures the agreement obtained that two raters classifying items in two mutually exclusive classes would obtain above and beyond chance (coincidental concordance). Raters in complete agreement would yield $\kappa = 1$ while raters achieving the same agreement as can be obtained by chance (owing to rater biases) would yield $\kappa = 0$. Raters in complete disagreement (worse than chance) can results in κ values as low as -1 [7,40]. In our case,

we compare the predictive model as a rater with a random rater. This measure is computed as $\kappa = \frac{P_r - P_c}{1 - P_c}$, where P_r is the relative observed agreement among raters, and P_c is the hypothetical probability of chance agreement, using the observed data to calculate the probabilities of observers randomly selecting categories. Cohen's κ in its classical version applies to two-rater two-class scenario. While various generalizations have been proposed to extend it to multi-class multi-rater case, these rely on the marginalization argument. Since our experiments are an instance of fixed rater case, we employ the Shah's κ_S [40].

3.5 Variables

Main independent variables are the subject applications, their software metrics, the values of LoM levels, ML algorithms with which we experiment, and approaches for selecting subsets of software metrics. A dependent variable is the learned model (and its performance measures) and the subset of software metrics with which this model was learned. We measure the performance of the learned model both in terms of precision, *P*, as the Cohen's κ-measure that specifies the agreement between two raters who each classify items into mutually exclusive categories, and the number of software metrics used as features to build the classifier. The effects of other variables (the structure of software applications and the types and semantics of data they process) are minimized by the design of this experiment.

Subject Applications A small number of subject applications is a threat to external validity, since it is unclear how results can be generalized. Our goal is to experiment with a large representative sample of software applications that are written and maintained by different stakeholders who assigned LoM levels to these applications independently. Even though some stakeholders may have personal biases when assigning LoM levels to some applications, we assume that the general trend is dictated by definitions of the LoM levels that we described in Section 3.1. We use 3,392 open-source projects, comprised of 958 alpha-phase projects, 1,292 beta-phase projects, and 1,142 production-phase projects. The source code for these open source projects was downloaded from Metrobase, an online database that contains the source code for open source Java projects collected from ten repositories: Sourceforge, Tigris, dev.java.net, Netbeans, nongnu.org, gnu.org, gna.org, apache.org, Javaforge.com, and googlecode.com. For each application, its creators and maintainers assigned one of the three LoM levels: alpha, beta, or production. These labels were extracted from the project repository FLOSSMole (`http://flossmole.org`).

We chose this dataset because it is the largest dataset that we found that contains LoM information. We accomplished the following steps to arrive to this dataset.

- We took all the Java projects from Merobase repository. We computed the metrics for all the classes from these projects. We cleaned data e.g.,, we excluded projects that did not contain source code or contained only a few classes (toy projects).
- We took all the metadata (including LoM info) from FLOSSMole repository, however, this dataset does not include the source code.

– We intersected these two sets (using the name of the projects) and arrived to the set of projects with metrics and LoM information.

Sourceforge has seven LoM levels for the project status: planning, pre-alpha, alpha, beta, production/stable, mature, and inactive. Some of these levels, however, have nothing to do with LoM. For example, the level *inactive* is used to mark the projects that are essentially dead meat (which could be as well mature or beta). Also, projects in the *planning* phase often do not even contain the source code, which means that we can not compute the metrics for those. Moreover, we did not find that many projects (less than ten) in certain LoM levels, so we discarded those. The only aggregation we did was to merge *pre-alpha* with *alpha* and we merged *production/stable* with *mature*.

ML Algorithms Since selection of a machine learning algorithm affects the accuracy of a learned model, we experiment with six different ML classification algorithms: Decision Tree, Naïve Bayes, Random Forest, NBTree, JRIP, and Support Vector Machine (SVM). The rationale for choosing this set of ML algorithms is that they are representative of the various classifier spaces explored and have shown to be very effective on a variety of domains [11].

Software Metrics For software metrics, we chose a large number of fundamentally different software metrics that are computed from the source code of these applications, thus effectively diversifying different criteria that may be used in the decision-making process when stakeholders assigned LoM levels to applications. The Software metrics were extracted using a tool, named `Columbus` that analyzes software systems and extracts 90 software metrics, errors, and warnings from project, package, class, attribute and method-level artifacts [16]. Software metrics were computed using the class granularity and aggregated for the application using their averages.

3.6 Hypotheses

We introduce the following null and alternative hypotheses to evaluate relations between LoM levels and internal software metrics.

H_0 The primary null hypothesis is that there is no difference between performances of different ML algorithms that are used to learn prediction models of LoM levels from software metrics.

H_1 An alternative hypothesis to H_0 is that there is significant difference between performances of different ML algorithms that are used to learn prediction models of LoM levels from software metrics.

To test the null hypothesis H_0, we are interested in the comparing precisions, recalls, and Cohen's *k*-measures to understand how ML algorithms perform better than random assignments. In particular, our experiments are designed to examine the following null and alternative hypotheses:

H1: Alpha and Beta. The effective null hypothesis is $\kappa_\alpha \approx k_\beta \approx 0$, while the true null hypothesis is that $p_\alpha \approx 50\%$ and $p_\beta \approx 50\%$. Conversely, the alternative hypothesis is $\kappa_\alpha \approx k_\beta >> 0.5$ and $p_\alpha >> 50\%$ and $p_\beta >> 50\%$.

H2: Production and Beta. The effective null hypothesis is $\kappa_\pi \approx k_\beta \approx 0$, while the true null hypothesis is that $p_\pi \approx 50\%$ and $p_\beta \approx 50\%$. Conversely, the alternative hypothesis is $\kappa_\pi \approx k_\beta >> 0.5$ and $p_\pi >> 50\%$ and $p_\beta >> 50\%$.

H3: Alpha and Production. The effective null hypothesis is $\kappa_\alpha \approx k_\pi \approx 0$, while the true null hypothesis is that $p_\alpha \approx 50\%$ and $p_\pi \approx 50\%$. Conversely, the alternative hypothesis is $\kappa_\alpha \approx k_\pi >> 0.5$ and $p_\alpha >> 50\%$ and $p_\pi >> 50\%$.

H4: Alpha and Beta and Production. The effective null hypothesis is $\kappa_\alpha \approx k_\beta \approx k_\pi \approx 0$, while the true null hypothesis is that $p_\alpha \approx 50\%$ and $p_\beta \approx 50\%$ and $p_\pi \approx 50\%$. Conversely, the alternative hypothesis is $\kappa_\alpha \approx k_\beta \approx k_\pi >> 0.5$ and $p_\alpha >> 50\%$ and $p_\beta >> 50\%$ and $p_\pi >> 50\%$.

The rationale behind the alternative hypotheses to `H1`, `H2`, `H3` and `H4` is that it is possible to create a model with some ML algorithm with which LoM levels of software applications can be predicted with a high precision using internal software metrics.

3.7 Threats to Validity

In this section, we discuss four kinds of threats: to statistical conclusion validity, internal validity, construct validity, and external validity [39, pages 61-65]. Our goal is to show that our experimental design gives an answer to the research question about the absence or relation between LoM levels (i.e., phases of testing) and internal software metrics with a high degree of confidence.

In the context of this paper, statistical conclusion validity refers to the question of presumed causal relationship between software metrics and LoM levels or absence thereof. Essentially, a threat to statistical conclusion validity is that we missed covarying between software metrics and LoM levels. To address this threat, we show that our experiments are sufficiently sensitive and able to detect small differences and we have evidence that software metrics and LoM levels do not covary. Specifically, this threat is addressed by using six ML algorithms instead of one with varying parameters and a feature selection approach to establish that no relation exists between software metrics and LoM levels. In addition, we experimented on thousands of different software projects to ensure that we significantly reduced the bias. Most importantly, the Cohen's κ-measure evaluates if a classifier performs better than a random choice, thus indicating that the possibility of covarying between independent and dependent variables. With a comprehensive set of measures and a thorough experimental design we address a threat to statistical conclusion validity.

Threats to internal validity refer to as confounding variables that may introduce a possible rival hypothesis to our hypotheses. In the context of our paper, confounding variables could be a threat if we uncover a strong relation between LoM levels and internal software metrics. In this case, a confounding variable could be the bias of the stakeholders who assign LoM levels to software projects based on some software metrics. In our case, the situation could be reversed, that is, a threat to internal validity could come from the bias of the stakeholders who assign LoM levels to software projects based on some software metrics which we did not take into consideration. However, it is highly unlikely, since we use many software applications whose LoM levels were assigned independently by many independent stakeholders, and the possibility that they all agreed on some obscure software metric that we did not cover in this paper is minuscule.

A threat to external validity refers to generalization of relation between LoM levels and internal software metrics beyond the confines of our experimental design. This is a main threat for our experimental design and it lies in the selection of subject applications and their software metrics. Since we operated on a very large set of diverse subject applications, we address a threat to external validity.

4 Empirical Results

To evaluate the null hypothesis H_0, we carried out experiments with six different ML algorithms using binary and multiclass classification to learn predictive models. We computed confusion matrices for six ML algorithms for hypotheses H1, H2, H3, and H4. A confusion matrix is a table that illustrates the performance of an ML algorithm. Each column of the matrix represents the instances in a predicted or modeled class, while each row represents the instances in an actual or true class, thereby illustrating how an algorithm confuses two classes (i.e. mislabels one as another) [41]. Symbols α, β, and π stand for Alpha, Beta, and Production LoM levels respectively. Results from these confusion matrices confirm that there are no significant differences between random choices and the results of classification, all Cohen's κ-measures are less than 0.2. In addition, precision is often lower than 50%, the highest is for testing H3 with the algorithm NBTree when it gets to 60.6%.

For the multiclass classification, the results are even worse, with the precision mostly in 30%, getting even to 28% for the Naïve Bayes classifier. That is, the results of prediction of LoM levels using software metrics is effectively the random choice. Based on the values of precision, P and Cohen's κ measures, we accept hypotheses H1–H4. In addition, results from experiments that use the feature selection backward elimination approach are consistent from both experiments. Hence, we accept the null hypothesis H_0 and reject the alternative hypothesis H_1, meaning that **there is no difference between performances of different ML algorithms that are used to learn prediction models of LoM levels from software metrics**. We can summarize these results as following.

- Classified Alpha-phase applications have the poorest precision and the worst recall values in general, and the multiclass classification gives the worst performance. We explain it as a result of large variances between software metrics for Alpha-phase applications, since these are the least stable versions of applications when compared to software metrics for applications that are assigned other LoM phases. Some stakeholders do a better job when releasing Alpha-phase applications, while others know that the applications should go through many changes before assigning these applications to more advanced phases (i.e., Beta and Production). Because of disparities in many software metrics, Alpha-phase applications are the most difficult ones to classify with precision.
- Performance differences are the most significant when considering binary classification between Alpha and Production-phase applications, while the differences between Alpha and Beta-phase applications are the least significant. We explain it as a result of maintenance and evolution work on applications between LoM phases. Since the amount of work on a software application is significant between Alpha

and Production phases, its software metrics differ a lot, and subsequently, it makes classifiers to learn better predictive models.
– Backward elimination rarely resulted in the improved performance of the predictive model, and when it did (i.e., for Random Forest for H1 and H2, for Decision Tree for H1 and H3, for JRIP for H1 and H3) it was not significant, on the order of three to five percent or less. The best and most consistent improvement using backward elimination was for the SVM algorithm for multiclass classification for all LoM phases, even though it was small to affect our conclusions (less than five percent).

We interpret the results of our experiments as follows.

1. *We strongly suggest that software metrics **cannot** be used alone, but only with some other indicators for estimation of LoM levels*. We think that the overall estimation of quality of software is based on some other factors, which we suggest depend on certain characteristics of the applications, finding which is a subject of future work. While this sounds like an unexpected result, the recent work in the similar direction showed that process metrics might be better predictors of software quality as compared to product metrics, which corroborates our result from a different point of view [37].
2. *We observe that the variance in precision and recall values is the largest for application in the Alpha phase*. For the algorithm NBTree, the precision and recall for projects in the Alpha-phase for H1 was zero percent with recall values for the algorithm Decision Tree as low as 5.3%. At the same time, other ML algorithms performed much better using the same data set, with Random Forest showing the precision of 42.1% and recall 23.5% for the same experiment H1. Such large variations in the learned predictive models show that it is difficult to rely on results of a single ML algorithm, since they are very sensitive to different dependencies in the input data.
3. *Our experimentation shows that the precision for multiclass experiments are in general lower than the precision for binary classification*. We explain it as a difficulty for classifiers to find discriminative attributes with more predictive power. Indeed, given that multiple software metrics have overlapping ranges of values for applications that are assigned to different LoMs, it is difficult to find a model that can discriminate among different LoMs. Adding more classes will likely exacerbate this problem, since finer granularity of LoM levels mean that there will be more overlapping among ranges of values for different software metrics, and therefore, worse precision.
4. *For the multiclass experiment, JRIP and SVM gave the worst values of recall for the alpha phase, and JRIP returned the worst value of recall, which is 2.5%*. This abnormality correlates with our binary experiments where alpha gives much lower recall when classified against beta LoM. However, in binary experiments where applications in alpha and production LoMs are classified, recall values are much closer to 50/50%. We explain it as a result of the difficulty to find strict discriminative software metrics for applications in alpha in beta phases, since they are much closer to one another. Indeed, different indirect evidence point out toward this explanation: the time intervals between releasing alpha and beta LoMs are shorter,

fewer changes are made, since the software applications are not released in production yet, and only critical and major bugs are fixed. The differences between production/beta and especially between production/alpha are much bigger, and it is reflected in the precision and recall values.

5. *Our experiments show it is possible to obtain good models by selecting a small subset of applications (less than 50) in a way that intervals are large between average values of software metrics that are collected from applications that belong to different LoMs*. In addition, when only a small subset of software metrics is chosen, it is easy to learn a model with a very high degree of accuracy. **Thus, we demonstrate the danger of making non-generalizable conclusions from a small set of data – increasing dimensions in classification problem illustrates the fallibility of small controlled experiments**. This is known as the curse of dimensionality [11], when a growing number of examples reduces the precision of classification algorithms drastically.

As part of using backward elimination algorithm for feature location, we computed top ten most representative software metrics for predicting LoM from a total of 90 software metrics. Even though using these metrics only leads to better prediction results, they still fall short of giving accurate prediction of software quality. These top ten metrics which are the following.

1. Long function is a bad smell metric that indicates that the sizes of methods grow too large.
2. Long parameter list is a bad smell metric that indicates that the number of parameters in methods is large.
3. Shotgun surgery is a bad smell metric that indicates that programmers change the source code in many locations when implementing a small feature.
4. The number of nonempty non-commented lines of code.
5. Number of incoming invocation summarizes the cardinality of the set of methods which invoke other methods.
6. Number of methods is a metric where locally defined and inherited methods and declarations and definitions are counted but if there is implementation for a declaration, the declaration is not counted.
7. Number of attributes is a metric that counts the local and inherited attributes of the class in an application.
8. Number of foreign methods accessed is a metric that summarizes the cardinality of the set of method invocations of the method where the invoked methods belong to other classes than the method itself.
9. Raw exception avoidance metric that describes the use of specialized exception classes rather than more general exception classes like Exception.
10. A metric that indicates a frequent use of method-level synchronization versus a block-level synchronization.

5 Background and Related Work

Software quality and LoM respectively are important aspects of modern software systems. Estimating and tracking quality and LoM of software systems is essential for various development and maintenance decisions. The ISO/IEC 9126 standard [23] defines

six high-level product quality characteristics which are widely accepted both by industrial experts and academic researchers: functionality, reliability, usability, efficiency, maintainability and portability. The characteristics are impacted by the low-level quality properties, that can be *internal* (measured by looking inside the product, e.g. by analyzing the source code or deriving product metrics) or *external* (measured by execution of the product, e.g. by performing testing or deriving process metrics) [4]. Many research papers have been published proposing software quality models ranging from purely theoretical to more applicable approaches [2, 4, 5, 9, 10, 26, 31, 34, 35]. While all these models are built from different combinations of product metrics, only a few of those were actually deployed and tested in industrial settings [4] [28] [38]. Unfortunately, they did not answer the question that we posed in this paper.

Finally, inter- and intra- system prediction of fault-prone classes or simply bug prediction is an active research areas with a number of research publications in the last decade. While we are not classifying or discussing all these models in-depth in this paper, we refer the reader to the recent systematic literature review on fault prediction of 208 studies [21]. The results of that review motivate our study. Also, feature selection appears to be one of the major success factors for building good models in our work. *However, none of the reported 208 studies is attempted on such a scale and uses such an exhaustive combination of internal product metrics.*

6 Conclusion

We answered the question negatively whether software quality can be accurately predicted using internal source code software metrics alone by conducting a large-scale empirical study with 3,392 open-source projects using six different classifiers. Further, we performed feature selection to determine if a subset of these metrics can do so to guard against noise and irrelevant attributes. Our results show that the accuracy of software quality prediction reaches $\approx 61\%$ leading us to suggest that comprehensive sets of internal software metrics alone cannot accurately predict software quality in general. Our result affects the research and deployment of hundreds of different approaches in academia and industry, since it shows that the claim is wrong that it is possible to effectively control and predict software development activities by using internal source code metrics alone to predict the quality of software projects in general.

7 Acknowledgments

We would like to thank to Rudolf Ferenc and Tibor Gyimothy for providing academic license for Columbus. This material is based upon work supported by the National Science Foundation under Grants No. 0916139, 1017633, 1217928, 1017305, and 1547597.

References

1. E. L. Allwein, R. E. Schapire, and Y. Singer. Reducing multiclass to binary: a unifying approach for margin classifiers. *J. Mach. Learn. Res.*, 1:113–141, Sept. 2001.

2. T. Alves, C. Ypma, and J. Visser. Deriving metric thresholds from benchmark data. In *26th ICSM'10*, pages 1–10, Timisoara, Romania, September 12-18, 2010. IEEE.
3. A. Arcuri and L. C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE*, pages 1–10, 2011.
4. T. Bakota, P. Hegedus, P. Krtvlyesi, R. Ferenc, and T. Gyimthy. A probabilistic software quality model. In *27th ICSM'11*, pages 243–252, Williamsburg, Virginia, USA, September 25-30, 2011.
5. J. Bansiya and C. Davis. A hierarchical model for object-oriented design quality assessment. *TSE*, 28(1):4–17, 2002.
6. B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.
7. J. Carletta. Assessing agreement on classification tasks: the kappa statistic. *Comput. Linguist.*, 22(2):249–254, June 1996.
8. M. Cataldo and S. Nambiar. On the relationship between process maturity and geographic distribution: an empirical analysis of their impact on software quality. In *Proceedings of the the 7th ESEC/FSE '09*, pages 101–110, New York, NY, USA, 2009. ACM.
9. J. Correia, Y. Kanellopoulos, and J. Visser. A survey-based study of the mapping of system properties to iso/iec 9126 maintainability characteristics. In *25th ICSM'09*, pages 61–70, Edmonton, Alberta, Canada, September 20-26, 2009. IEEE.
10. M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Softw. Engg.*, 17(4-5):531–577, Aug. 2012.
11. P. Domingos. A few useful things to know about machine learning. *Commun. ACM*, 55(10):78–87, Oct. 2012.
12. S. K. Dubey, A. Rana, and Y. Dash. Maintainability prediction of object-oriented software system by multilayer perceptron model. *SIGSOFT Softw. Eng. Notes*, 37(5):1–4, Sept. 2012.
13. N. E. Fenton, P. Krause, and M. Neil. Probabilistic modelling for software quality control. In *Proceedings of the 6th ECSQARU '01*, pages 444–453, London, UK, UK, 2001. Springer-Verlag.
14. N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Trans. Softw. Eng.*, 25(5):675–689, Sept. 1999.
15. N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition, 1998.
16. R. Ferenc, A. Beszédes, M. Tarkiainen, and T. Gyimthy. Columbus - reverse engineering tool and schema for c++. In *Proceedings of the ICSM'02*, ICSM '02, pages 172–181, Washington, DC, USA, 2002. IEEE Computer Society.
17. J. Ferzund, S. N. Ahsan, and F. Wotawa. Empirical evaluation of hunk metrics as bug predictors. In *Proceedings of the IWSM '09 /Mensura '09*, pages 242–254, Berlin, Heidelberg, 2009. Springer-Verlag.
18. M. Genero, J. Olivas, M. Piattini, and F. Romero. Using metrics to predict oo information systems maintainability. In *Proceedings of the 13th CAiSE '01*, pages 388–401, London, UK, UK, 2001. Springer-Verlag.
19. C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2002.
20. I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, Mar. 2003.
21. T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE TSE*, 38(6), 2012.
22. J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010.
23. ISO/IEC. Iso/iec 9126. software engineering - product quality, 2001.

24. N. Japkowicz and M. Shah. *Evaluating Learning Algorithms: A Classification Perspective*. Cambridge University Press, New York, NY, USA, 2011.
25. C. Jones. *Applied software measurement (2nd ed.): assuring productivity and quality*. McGraw-Hill, Inc., Hightstown, NJ, USA, 3rd edition, 2008.
26. H.-W. Jung, S.-G. Kim, and C.-S. Chung. Measuring software product quality: a survey of iso/iec 9126. *IEEE Software*, 21(5):88–92, 2004.
27. T. M. Khoshgoftaar, E. B. Allen, and Z. Xu. Predicting testability of program modules using a neural network. In *Proceedings of the 3rd Symposium on ASSET '00*, pages 57–, Washington, DC, USA, 2000. IEEE Computer Society.
28. S. Kim, T. Zimmermann, J. E. Whitehead, and A. Zeller. Predicting faults from cached history. In *29th ICSE'07*, pages 489–498, Minneapolis, MN, USA, May 20-26, 2007.
29. R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artif. Intell.*, 97(1-2):273–324, Dec. 1997.
30. D. Koller and M. Sahami. Toward optimal feature selection. In *In 13th International Conference on Machine Learning*, pages 284–292, 1995.
31. T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE TSE*, 33(1):2–13, 2007.
32. I. Mierswa, M. Scholz, R. Klinkenberg, M. Wurst, and T. Euler. Yale: Rapid prototyping for complex data mining tasks. In *In Proceedings of the 12th ACM SIGKDD*, pages 935–940. ACM Press, 2006.
33. J. Moses. Learning how to improve effort estimation in small software development companies. In *24th COMPSAC '00*, pages 522–527, Washington, DC, USA, 2000. IEEE Computer Society.
34. P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In *IEEE ICSM'92*, pages 337–344, Orlando, FL, USA, November, 1992. IEEE.
35. I. Ozkaya, L. Bass, R. Nord, and R. Sangwan. Making practical use of quality attribute information. *IEEE Software*, 25(2):25–33, 2008.
36. K. Raaschou and A. W. Rainer. Exposure model for prediction of number of customer reported defects. In *Proceedings of the ESEM '08*, pages 306–308, New York, NY, USA, 2008. ACM.
37. F. Rahman and P. Devanbu. How, and why process metrics are better. In *35th IEEE/ACM ICSE'13*, San Francisco, CA, 2013.
38. F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu. Bugcache for inspections: Hit or miss? In *8th ESEC/FSE'11*, pages 322–331, Szeged, Hungary, September 5-9, 2011. ACM.
39. R. Rosenthal and R. L. Rosnow. *Essentials of behavioral research : methods and data analysis*. McGraw-Hill, 2nd edition, 1991.
40. M. Shah. Generalized agreement statistics over fixed group of experts. In *Proceedings of the 2011 ECML PKDD'11*, pages 191–206, Berlin, Heidelberg, 2011. Springer-Verlag.
41. S. V. Stehman. Selecting and interpreting measures of thematic classification accuracy. *Remote Sensing of Environment*, 61(1):77–89, Oct. 1997.
42. A. Tosun, A. Bener, B. Turhan, and T. Menzies. Practical considerations in deploying statistical methods for defect prediction: A case study within the turkish telecommunications industry. *Inf. Softw. Technol.*, 52(11):1242–1257, Nov. 2010.
43. V. N. Vapnik. *The nature of statistical learning theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.
44. A. Vogelsang, A. Fehnker, R. Huuck, and W. Reif. Software metrics in static program analysis. In *Proceedings of the 12th ICFEM'10*, pages 485–500, Berlin, Heidelberg, 2010. Springer-Verlag.
45. S. A. Wake and S. M. Henry. A model based on software quality factors which predicts maintainability. Technical report, Virginia Polytechnic Institute & State University, Blacksburg, VA, USA, 1988.

8 Appendix: For Reviewing Only, It Will be Removed In the Camera-Ready Version

Table 1: Confusion matrices for binary classification experiments that address hypotheses H1, H2, and H3 using six ML algorithms. Symbols α, β, and π stand for Alpha, Beta, and Production LoM levels respectively. Rows labeled as M show the modeled numbers of samples and columns labeled as T show true numbers of samples. Variables *P* and *R* stand for precision and recall respectively. The lowest right cell in each confusion matrix holds the value of Cohen's κ-measure.

ML \ H	H1				H2				H3			
DecTree	M \ T	α	β	*P*	M \ T	π	β	*P*	M \ T	α	π	*P*
	α	68	74	47.9%	π	183	172	51.6%	α	461	502	47.9%
	β	890	1218	57.8%	β	959	1120	53.9%	π	497	640	56.3%
	R	7.1%	94.3%	0.02	*R*	16%	86.7%	0.03	*R*	48.1%	56%	0.04
Bayes	M \ T	α	β	*P*	M \ T	π	β	*P*	M \ T	α	π	*P*
	α	663	895	42.6%	π	797	910	46.7%	α	542	603	47.3%
	β	295	397	57.4%	β	345	382	52.5%	π	416	539	56.4%
	R	69.2%	30.7%	0.0	*R*	69.8%	29.6%	0.01	*R*	56.6%	47.2%	0.04
Forest	M \ T	α	β	*P*	M \ T	π	β	*P*	M \ T	α	π	*P*
	α	243	289	45.7%	π	654	689	48.7%	α	350	325	51.9%
	β	715	1003	58.4%	β	488	603	55.3%	π	608	817	51.9%
	R	25.4%	77.6%	0.03	*R*	57.3%	46.7%	0.04	*R*	36.5%	71.5%	0.08
JRIP	M \ T	α	β	*P*	M \ T	π	β	*P*	M \ T	α	π	*P*
	α	225	280	44.6%	π	451	402	52.9%	α	381	413	48%
	β	733	1012	58%	β	691	890	56.3%	π	577	729	55.8%
	R	23.5%	78.3%	0.02	*R*	39.5%	68.9%	0.09	*R*	39.8%	63.8%	0.04
NBTree	M \ T	α	β	*P*	M \ T	π	β	*P*	M \ T	α	π	*P*
	α	0	0	0.0%	π	293	307	48.8%	α	513	459	52.8%
	β	958	1292	57.4%	β	849	985	53.7%	π	445	683	60.6%
	R	0.0%	100%	0.0	*R*	25.7%	76.2%	0.02	*R*	53.6%	59.8%	0.14
SVM	M \ T	α	β	*P*	M \ T	π	β	*P*	M \ T	α	π	*P*
	α	134	136	49.6%	π	307	294	51.1%	α	383	340	53%
	β	824	1156	58.4%	β	835	998	54.5%	π	575	802	58.2%
	R	14%	89.5%	0.04	*R*	26.9%	77.2%	0.0	*R*	40%	70.2%	0.1

Table 2: Confusion matrices for binary classification experiments with *the backward elimination feature selection approach* that address hypotheses H1, H2, and H3 using six ML algorithms. Symbols α, β, and π stand for Alpha, Beta, and Production LoM levels respectively. Rows labeled as M show the modeled numbers of samples and columns labeled as T show true numbers of samples. Variables *P* and *R* stand for precision and recall respectively. The lowest right cell in each confusion matrix holds the value of Cohen's κ-measure.

ML \ H	H1				H2				H3			
DecTree	M \ T	α	β	*P*	M \ T	π	β	*P*	M \ T	α	π	*P*
	α	51	50	50.5%	π	183	174	51.3%	α	527	563	48.4%
	β	907	1242	57.8%	β	959	1118	53.8%	π	431	579	57.3%
	R	5.3%	96.1%	0.02	*R*	16%	86.5%	0.02	*R*	55%	50.7%	0.06
Nv Bayes	M \ T	α	β	*P*	M \ T	π	β	*P*	M \ T	α	π	*P*
	α	182	233	43.9%	π	537	579	48.1%	α	493	529	48.2%
	β	776	1059	57.7%	β	605	713	54.1%	π	465	613	56.9%
	R	19%	82%	0.01	*R*	47%	55.2%	0.02	*R*	51.5%	53.7%	0.05
Forest	M \ T	α	β	*P*	M \ T	π	β	*P*	M \ T	α	π	*P*
	α	225	310	42.1%	π	661	663	49.9%	α	367	353	51%
	β	733	982	57.3%	β	481	629	56.7%	π	591	789	57.2%
	R	23.5%	76%	0.01	*R*	57.9%	48.7%	0.07	*R*	38.3%	69.1%	0.08
JRIP	M \ T	α	β	*P*	M \ T	π	β	*P*	M \ T	α	π	*P*
	α	186	220	45.8%	π	403	384	51.2%	α	421	447	48.5%
	β	772	1072	58.1%	β	739	908	55.1%	π	537	695	56.4%
	R	19.4%	83%	0.03	*R*	35.3%	70.3%	0.06	*R*	44%	60.9%	0.05
NBTree	M \ T	α	β	*P*	M \ T	π	β	*P*	M \ T	α	π	*P*
	α	0	0	0%	π	296	310	48.8%	α	513	459	52.8%
	β	958	1292	57.4%	β	846	982	53.7%	π	445	683	60.6%
	R	0%	100%	0.0	*R*	25.9%	76%	0.02	*R*	53.6%	59.8%	0.13
SVM	M \ T	α	β	*P*	M \ T	π	β	*P*	M \ T	α	π	*P*
	α	112	120	48.3%	π	280	268	51.1%	α	362	314	53.6%
	β	846	1172	58.1%	β	862	1024	54.3%	π	596	828	58.2%
	R	11.7%	90.7%	0.04	*R*	24.5%	79.3%	0.04	*R*	37.8%	72.5%	0.12

Table 3: Confusion matrices for the multiclass experiments that address hypothesis H4 using six ML algorithms. Symbols α, β, and π stand for Alpha, Beta, and Production LoM levels respectively. Rows labeled as M show the modeled numbers of samples and columns labeled as T show true numbers of samples. Variables *P* and *R* stand for precision and recall respectively. The lowest right cell in each confusion matrix holds the value of Shah's κ-measure.

ML Algor	H4				
Dec Tree	M \ T	α	β	π	*P*
	α	216	262	234	30.2%
	β	513	677	571	38.4%
	π	229	353	334	36.5%
	R	49.3%	12.5%	41%	0.02
Naïve Bayes	M \ T	α	β	π	*P*
	α	472	658	545	28.2%
	β	115	161	129	39.8%
	π	371	473	468	35.7%
	R	22.6%	52.4%	29.3%	0.01
Rand Forest	M \ T	α	β	π	*P*
	α	220	254	216	31.9%
	β	381	576	463	40.6%
	π	357	462	463	36.1%
	R	23%	44.6%	40.5%	0.04
JRIP	M \ T	α	β	π	*P*
	α	24	24	24	43.65%
	β	817	1070	874	38.8%
	π	117	198	244	43.7%
	R	2.5%	82.8%	21.4%	0.04
NBTree	M \ T	α	β	π	*P*
	α	202	261	181	31.4%
	β	525	643	547	37.5%
	π	231	388	414	40.1%
	R	21.1%	49.8%	36.3%	0.03
SVM	M \ T	α	β	π	*P*
	α	97	85	73	38%
	β	698	946	805	38.6%
	π	163	261	264	38.4%
	R	10.3%	73.2%	23.1%	0.03

Table 4: Confusion matrices for the multiclass experiments using *the backward elimination feature selection approach* that address hypothesis H4 using six ML algorithms. Symbols α, β, and π stand for Alpha, Beta, and Production LoM levels respectively. Rows labeled as M show the modeled numbers of samples and columns labeled as T show true numbers of samples. The lowest right cell in each confusion matrix holds the value of Cohen's κ-measure.

ML Algor	H4				
Dec Tree	M \ T	α	β	π	*P*
	α	193	262	224	28.4%
	β	513	696	572	40%
	π	252	394	346	34.9%
	R	20.2%	49.2%	30.3%	0.01
Naïve Bayes	M \ T	α	β	π	*P*
	α	348	469	396	28.7%
	β	278	405	323	40.3%
	π	332	418	423	36.1%
	R	36.3%	31.4%	37%	0.02
Rand Forest	M \ T	α	β	π	*P*
	α	212	241	200	32.5%
	β	409	557	445	39.5%
	π	337	494	497	37.4%
	R	22.1%	43.1%	43.5%	0.04
JRIP	M \ T	α	β	π	*P*
	α	30	35	23	34.1%
	β	838	1135	968	38.6%
	π	90	122	151	41.6
	R	3.1%	87.9%	13.2%	0.02
NBTree	M \ T	α	β	π	*P*
	α	166	177	120	35.9%
	β	600	812	713	38.2%
	π	192	303	309	38.4%
	R	17.3%	62.9%	27.1%	0.03
SVM	M \ T	α	β	π	*P*
	α	88	77	55	40%
	β	722	983	809	39.1%
	π	148	232	278	42.3%
	R	9.2%	76.1%	24.3%	0.05