Computing Contextual Metric Thresholds

Matthieu Foucault
Bordeaux University
matthieu.foucault@labri.fr

Marc Palyart
Bordeaux University
marc.palyart@labri.fr

Jean-Rémy Falleri
Bordeaux University
falleri@labri.fr

Xavier Blanc
Bordeaux University
xavier.blanc@labri.fr

ABSTRACT

Software metrics have been developed to measure the quality of software systems. A proper use of metrics requires thresholds to determine whether the value of a metric is normal or abnormal. Many approaches propose to define thresholds based on large analysis of software systems. However, as thresholds depend a lot on the context, such as the programming language of the project or its applicative domain, there is then a need for an approach that takes the context as an input to compute thresholds. In this paper we propose such an approach with the objective to reach a trade-off between representativeness of the thresholds and efficiency of its computation cost. Our approach is based on an unbiased selection of software entities and makes no assumptions on the statistical properties of software metrics. It can therefore be used by any one who wants to quickly compute a representative threshold for a given metrics and a given context.

1. INTRODUCTION

Software metrics are measures that can be used as guide in the decision-making process for improving software quality [13]. They can be roughly defined as mappings from the empirical world (i.e. classes, methods, commits, developers) to the formal relational world (a value in $\mathbb{R}$). Software metrics need thresholds which give them semantics [15]. A threshold makes a partition of software entities by creating two distinct groups: the entities that have a good value and the other ones that are considered to be risky[11].

For example, NOA and NOM are two metrics that respectively measure the number of attributes and methods of a class. These two metrics can be used to identify god classes, which are classes that contain too many attributes and methods and are therefore difficult to maintain [19]. The identification of god classes by using the NOA and NOM metrics requires the definition of a threshold. This threshold must state how many attributes and methods at least have to be declared in a class to consider it as a god class.

SAC'14 March 24-28, 2014, Gyeongju, Korea.

Recently, Zhang et al. have shown that thresholds depend on context and therefore cannot be generalized to all kinds of software systems [25]. In particular, the programming language or the domain of application are contexts that have a strong impact on the thresholds. This further strengthens the results of Nagappan et al. who have shown that thresholds obtained by performing a correlation analysis are only true for a limited set of similar software systems[20].

As there are too many contexts, thresholds cannot be computed for all of them. Thus there is a need for an approach that can, for a given context, compute automatically the corresponding threshold. With such approach a developer or a manager could identify risky software elements by computing thresholds for metrics assessing these software elements. For example, if the manager of a 3 years old Java project involving more than 10 developers wants to use the NOA and NOM metrics, she will be able to know what are the NOA and NOM thresholds for her context.

Computing thresholds for any given context raises two main issues that are representativeness and efficiency. By representativeness we mean that the threshold must truly partition the software elements that fit to the context. By efficiency we mean that the threshold must be computed quite quickly as the intent is to use it as soon as possible in order to identify risky software elements. These two issues are antagonist. Increasing the representativeness of a thresholds requires to analyze more software entities and therefore requires more time. In the opposite, decreasing the time needed to compute a threshold irremediably implies that less software entities have to be analyzed and therefore decreases the representativeness of the threshold.

In this paper we propose an approach for computing threshold for any given context. Our approach aims to reach a trade-off between representativeness and efficiency. It is based on two main principles. The first one is a random selection of both software projects and software entities. This random selection, called double sampling, offers guarantees of representativeness as it is unbiased and has good performance as few software elements have to be analysed [**?**]. The second principle follows Chidamber et al. principle[5] and considers that a metric threshold can be derived from a quantile of the distribution of metric values. Our process then inputs the quantile to reach (80%, 90% or 95%) and returns an estimation of the corresponding threshold. To compute this estimation our process relies on Bootstrap, which is a statistical approach proposed in the late 70's [9]. Bootstrap has the advantage of being adaptable to any statistic and is independent of the distribution of metric values.

We have prototyped and validated our approach to generate different thresholds of metrics from software projects hosted in GitHub, which is an open-source hosting platform. Rather than the values we obtained for the thresholds, the major results of our validation comes from the feedback we obtained regarding the number of software entities and projects that have to be analyzed and the time that is needed by our process.

The rest of this paper is organized as follows. Section 2 starts by presenting the issues related to metrics thresholds and then presents our process to compute threshold from a statistical analysis. Section 3 presents a validation of our process. Section 4 then presents a general discussion of the advantages and the limits of our process. Section 5 then presents the related work and the Section 6 presents our conclusion.

2. THRESHOLDS PROCESS

This section starts by giving definitions for metrics and thresholds with the intent to highlight the two major issues that have to be faced by any process which goal is to compute metrics thresholds for any given context. It then presents how we propose to face these two issues and describes our process proposal.

2.1 Issues related to the definition of metrics thresholds

A metric is a quantitative measure that is done on a software entity (class, method, developer, commit, etc.)[13, 6]. For the sake of simplicity, we consider that a metric is a function μ that is defined for a specific kind k of software entity and that returns a real (see definition 1). For instance, the NOA metric is a function which inputs a class and measures its number of attributes ($NOA : U_{Class} \rightarrow \mathbb{R}$) and the NOM metric inputs a class and measures the number of its methods ($NOM : U_{Class} \rightarrow \mathbb{R}$).

Definition 1 (Metric). *Let k be a kind of software entity such as Class, Method or Developer. Let U_k be the set of all software entities of that kind. For instance, U_{class} is the set of all classes (such a set is conceptual and cannot be computed). A metric μ_k is a function that measures entities of a given kind by returning a real value. Therefore $\mu_k : U_k \rightarrow \mathbb{R}$*

A threshold of a given metric is a value that splits the software entities into two groups, the ones that have a metric value that is lower or equal than the threshold and the ones that have a metric value higher than the threshold[11] (see definition 2).

Definition 2 (Threshold). *A threshold of a given metric μ_k is a value ($t_{\mu_k} \in \mathbb{R}$) for that metric that splits the set of software entities U_k in two groups ($Low \cup High = U_k$ and $Low \cap High = \emptyset$). All entities of Low (resp. High) have a metric value that is lower or equal (resp. upper) than the threshold ($\forall e \in Low, \mu_k(e) \leq t_{\mu_k}$ and $\forall e \in High, \mu_k(e) > t_{\mu_k}$).*

As thresholds depend on context, any approach that aims to compute a threshold must take into account the context. A context defines the environment of the software entities of interest. As all software entities belong to software projects, we propose to define a context by a logic predicate that applies to software projects to state whether or not they fit to the context (see definition 3). For instance, the context that identifies *Java projects* can be defined by a predicate that checks whether the project contains at least one Java file.

Definition 3 (Context). *Let P be the set of all software projects. A context c is a logic predicate that applies to any software project and that states whether or not the project fits to the context ($c : P \rightarrow \mathbb{B}$).*

Any approach that aims to compute a threshold defines a process that inputs a metric and a context and that return the corresponding threshold (see definition 4).

Definition 4 (Computing thresholds). *An approach that computes thresholds for any given context defines process that inputs a metric μ_k and a context c and that returns the threshold $t^c_{\mu_k}$.*

The main motivation of computing a threshold is to identify outliers of the context with the intent to consider them as risky entities[11, 3]. An approach that computes thresholds for any given context is representative if the thresholds it generates truly identify the outliers of the given context. In other words, the thresholds must at least identify outliers of the context.

Further, we argue that the ones that will want to generate thresholds will probably want to quickly identify outliers of their project. An approach that generates thresholds is efficient if it takes few time to compute a threshold for a given metric and a given context. Having no strong requirement on time, we propose to state that it should generate thresholds in few minutes or few hours.

Nevertheless, the more entities of a context are analysed for computing a threshold, the more representative the threshold should be. This however takes many time. An approach that aims to compute threshold for a given context must then reach a trade-off between representativeness and efficiency.

2.2 Quantile Based Process

Many approaches have been proposed to compute thresholds but none of them was defined to compute thresholds for any given context [15, 11, 5, 24, 1, 22, 2, 14].

We propose to follow the approach of Chidamber et al. who define that a threshold can be simply computed by estimating a quantile with a lower bound at the 80th percentile [5]. For instance, if one wants to compute a threshold to identify god classes of Java projects, she can consider that outliers are classes that fall above the 80% of small classes.

We argue that a percentile is adequate for computing thresholds for any given context. First it is independent of the distribution of the metrics and second it can be estimated on a small sample of software entities. It is therefore adequate to reach a trade-off between representativeness and efficiency.

Computing a percentile is a classic estimation of a value based on measured data [**?**]. This requires at least three steps. First, a sample of entities that fit the context ($\Omega \subseteq U^c_k$) has to be built, second the metrics of the entities of the sample have to be computed, and third a statistical approach has to be executed to compute the estimation of the threshold. As the second step of this process is quite obvious, the main difficulties are then (A) the construction of

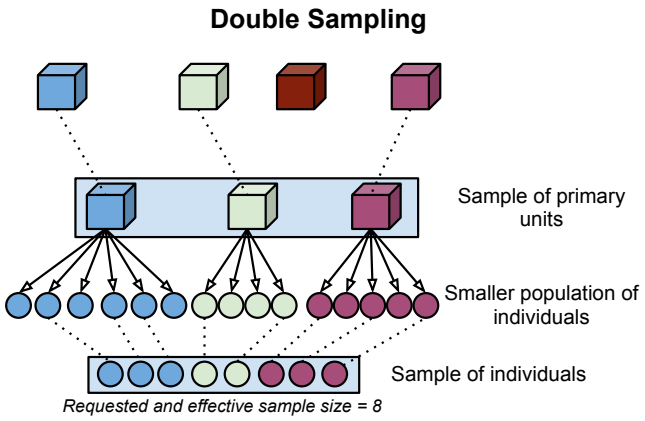

Figure 1: Double sampling. The arrows indicate the *contains* relationship between primary units and individuals, the dotted lines mean *selected at random*. The rectangles indicate the selected samples.

the sample Ω and (B) the computation of the estimation for the threshold thanks to a statistical approach. The following two sections explain how our approach tackles these two issues.

2.2.1 Building Ω

To build a precise estimation from a sample Ω, the sample must be representative of the whole set of software entities that fit a context. Representativeness means that each entity of the population U_k^c should have the same probability to be part of the sample. In other words, all entities of a sample should be selected randomly with the same probability. For example, if the population is finite and contains N entities, the probability to select one entity in the sample must be $P(e) = 1/N$.

In software engineering, software entities that fit a context, such as Java classes of large project for example, cannot be directly randomly selected. However, the software projects in which they are defined can be selected randomly, at least if the population of projects is known.

This is quite similar to an imaginary situation where one wants people to answer a questionnaire but he/she only can drive a car to reach the people and to get their answers. He/she therefore has to go from city to city to get the answers. In such a case, the representativeness of the selected people but also the cost of the selecting process are the key factors. Needless to say that if he/she goes to one of the big cities and interview all of its inhabitants the cost will be low but the answers won't be representative. In the contrary, if he/she plan to visit all the cities to interview very few of their inhabitants, this will definitively be too costly and maybe not so representative.

To face the issue of representativeness and to randomly create samples our process is based on a double sampling. It then starts by randomly selecting a set of software projects to fit a context from a given set of known projects. Then it creates a finite population that contains all software entities contained in the selected projects. Finally, it randomly selects entities in this finite population to build the sample Ω. The Figure 1 presents the three steps of the double sampling.

The double sampling has the major advantage to be purely based on random selection. Projects and entities are selected randomly. Moreover, it has the double advantage to include entities from different projects and to not include all entities that belong to a same project. As a consequence, the double sampling does not suffer from the bias of having many entities that share similar properties, which may be the case when they are contained in a same project.

The double sampling can be configured with two options: the number of projects contained in the first sample and the number of software entities contained in the returned sample. These two options have an impact on the quality of the estimation. Section 3 will present the effect of these two options on the quality of the returned threshold of different metrics.

The main drawback of the double sampling is its cost as it has to select several projects and has to build a population. Further, the number of selected projects has to be chosen carefully depending on the size of the wanted sample. A large number of projects have to be selected to be representative of the diversity of all software projects. However, the larger the set of projects, the larger the population and therefore the cost is more expansive.

2.2.2 Estimation for the threshold

The computation of the estimation requires to choose a statistical approach. Even if we choose to reduce the estimation of a threshold to the estimation of a quantile, there are still many statistic approaches that can be used. Further, it should be noted that it is better to know the distribution of a metrics to use the adequate one. Some metrics, such as NOA and NOM, follow a power law for most of the projects, but not for all [16]. In the general case, the distribution is unknown and therefore statistical approaches that are robust regardless of the distribution have to be used.

Whatever the statistical approaches, an estimation aims to reflect a real value and provides an error margin. This error margin is often represented as a confidence interval $[a; b]$ with an error probability α (or a confidence coefficient $1 - \alpha$). Our process uses intervals with a 95% confidence coefficient, which means that there is a 95% chance that the real value is between a and b (included).

Among the existing statistical approaches, we choose to base our process on *bootstrap*. Bootstrap was introduced in 1979 by Bradley Efron [9] as a method for estimating a confidence interval for a statistics. Its principal advantage is that it is available for many statistics such as quantile, which is the basis of our process. Moreover, it is a computer-based approach that can be easily used with the dedicated R library [4].

The bootstrap approach inputs a sample $\Omega = (y_1, ..., y_n)$ and returns a confidence interval for a given statistic that is computed with the following algorithm:

1. Build B independent *bootstrap samples*, noted $\Omega^{*1}, ..., \Omega^{*B}$. These samples are drawn from Ω using random sampling with replacement. In other words, these samples contain members from Ω, some appearing zero time, some appearing once and some more than once. This part of the algorithm is illustrated in Figure 2. The number of independent bootstrap samples (B) can be configured but we choose to create 10000 of them, to ensure that bootstrap will find a confidence interval as accurate as possible. Although the literature uses

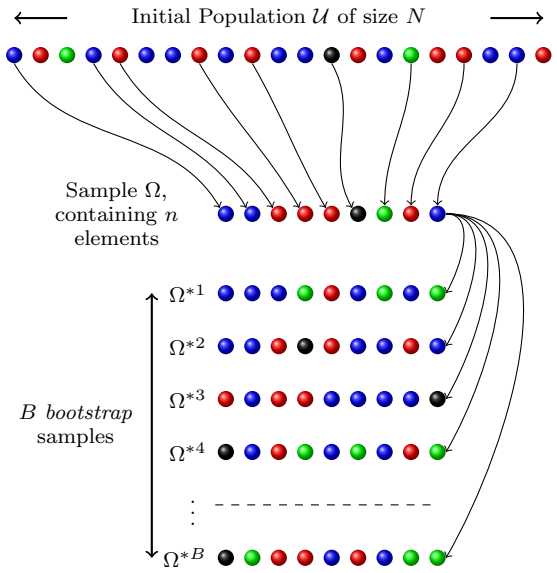

Figure 2: The bootstrap algorithm[1]

a number of bootstrap replicates around $B = 2000$, we experimented cases where bootstrap was unable to compute a confidence interval with this value. Furthermore, computers are more powerful now than when theses studies were conducted and using 10000 bootstrap samples only produces an overhead of some minutes with samples having thousands of values, which perfectly acceptable in our case.

2. For each sample, compute a replication of the statistic $\hat{\theta}^*(b) = f(\Omega^{*b})$. In our case its the computation of the quantile.

3. From the B replications of the estimated statistic, use one of bootstrap's algorithms to obtain a confidence interval. Several algorithms are defined but we choose to use the BC_A procedure [10], which offers quite accurate confidence intervals and is implemented in the R *boot* library [4].

2.2.3 *Our process*

As presented in the two last sections, our process is based on a double sampling and on the bootstrap approach. It inputs a metric μ_k and a description of a context c and returns the threshold that corresponds to the 80th percentile. An overview of our process is presented in Figure 3. It first create a sample of software entities using a double sampling. Then it computes all the metrics for the software entities contained in the sample. Then it uses Bootstrap to compute an interval for the threshold. As Bootstrap guarantees with 95% of confidence that the threshold belongs to the interval, we choose to return the mean of the boundaries of the interval.

Several options can be used to tune our process. The first one is the list of projects that will be used to create the sample. This list defines the whole population and has to be chosen carefully as the threshold returned by our approach will

[1] Figure under Creative Commons license, original work by Germain Salvato-Vallverdu

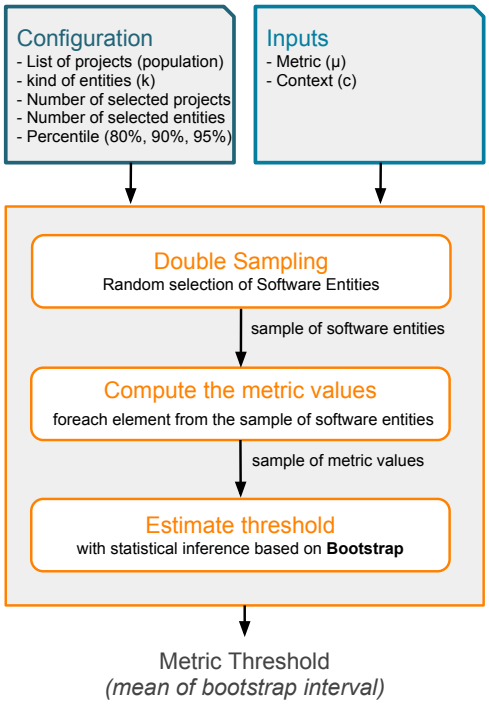

Figure 3: Overview of the proposed process for computing metrics thresholds

be representative of that population. If the context targets Open Source projects, we usually choose a hosting platform such as Sourceforge or GitHub for this option. If the context targets industrial projects, we usually choose all the project that have been realized by the company that is asking for a threshold. Our process will then choose repositories in the population and consider them as projects. We apply a filter to select only some of the repositories of the population that do fit the context. For instance, one may want to analyse only repositories that contain more than 100 Java classes, or repositories using a particular configuration management system such as Maven, or even repositories that exist for more than two years. The second option is the kind k of software entities that will be part of the sample Ω. This option has to be defined with a fetch function that can extract all software entities of the given kind that are contained in a project. This function will be used to create the sample of software entities.

Further, as presented in the previous section, the double sampling has to be configured. The two options are the number of projects and the number of software entities. These two options have an impact on the quality of the returned threshold but their values depend on the metrics and on the population. The section 3 presents an analysis of these options for the GitHub platform and for some well-known metrics.

Finally, the reached percentile can be configured too. We mainly follow the 80% principle but if one wants a stronger threshold, she can choose either 90% or even 95% as a target percentile.

It should be noted that the Bootstrap approach can be configured too but we chose to propose a fixed configura-

tion. As presented in the previous section, bootstrap takes as input an integer B, which is the number of *bootstrap samples*. The larger the value, the more accurate the result will be. However, increasing this value also increase bootstrap's computation time and memory usage. With modern computers, it is although possible to have a large number of *bootstrap samples*. We arbitrary choose to use $B = 10000$. The other value which can be configured is the confidence coefficient of bootstrap. If this value is too high (e.g. 99%), the resulting intervals will be very large. On the contrary, a lower confidence coefficient will provide narrower confidence intervals, but they may not be accurate. As stated above, we use a 95% confidence coefficient. Finally, bootstrap requires an algorithm to obtain a confidence interval. For that option, we choose to use the BC_A one.

3. VALIDATION

TBD

3.1 Prototype

The research prototype we developed to conduct this study is composed of three parts that respectively implement the three steps of our process. This tool was built on top of HARMONY [**?**] which is an open source research platform designed to ease the development of tools that mine software repositories.

The first part implements the double sampling. It uses the population of projects hosted on GitHub. This population has been obtained by querying the GitHub API to obtain its full list of stored projects. We selected a wide context: Java projects that do not come from a project fork. GitHub stores many forked projects and considers them as different projects but still knows that they are forked projects. While performing the study we found that around 40% of Java projects stored in GitHub were marked as forked projects. However some developer make manual copies of repository. In this case we are not able to detect them and thus these copies of projects are still included in our population. This layer inputs three parameters: the context (project age, language, size, ...), the size of the sample of projects and the size of the sample of software entities, here Java classes.

The second part aims to compute the metrics values. It is defined as an HARMONY analysis (see [**?**] for more details). The analysis runs on the Java classes selected during the previous step.

The third part uses the R language to compute the threshold. This layer inputs the target quantile and the metrics values that have been computed by the HARMONY analysis. It finally returns the threshold.

All these three parts are integrated together and their execution can be chained to output directly metrics thresholds.

3.2 Representativeness

To evaluate the threshold computed by our approach we selected the two following metrics as a benchmark:

- *NOA+NOM* (Number Of Attribute, Number Of Method). This metric counts the number of attributes and methods of a class. This combination of metrics is useful to identify code smells such as the God Class anti-pattern [19] which highlights classes with to many features
- *CBO* (Coupling Between Object classes). This metric represents the number of classes coupled to a given class through method calls, field accesses, inheritance, arguments, return types, and exceptions.

To measure the quality of a computed threshold, we decided to compare the quantile obtained by the threshold with the reached quantile. To make a robust comparison, we have generated 10 sets of 100 projects as a benchmark. We then measure the obtained quantile for each of this 10 sets. We finally use the root-mean-square error (RMSE) to measure the difference between the 10 obtained quantiles with the reached quantile. In our validation we arbitrary selected 90% as the quantile estimated by the thresholds.

The representativeness of DS and Bootstrap has already been validated in other domain. That is why this section is more concerned about the configuration of the process

The main goal of our validation is to measure the effect of the number of projects and classes on the computed thresholds. This analysis is a classical empirical analysis that has two independent factors (the number of projects and the number of classes) and one dependent variable (the obtained error). Other independent variables we didn't modify are the quantile, the population of projects and the configuration of bootstrap. The object of the analysis is to generate a threshold for two known metrics.

For the treatments of the number of projects, we decided to use a logarithmic scale as we thought that the effect was more important for small numbers of projects. As the number of projects is a natural, we decided to use Fibonacci numbers for treatments and deliberately decided to start with 3 projects and to stop with 377.

For the treatments of the number of projects, we decided to use a linear scale with a step of 200 classes. We deliberately decided to start with 200 classes and to stop with 2,000 classes.

The tables 1, 2 presents the results we obtained respectively for the NOA+NOM and CBO metrics. ∅ means that no sample of classes could be constructed because the selected projects contain too few classes.

Two main results have been obtained by our validation. First of all, a minimum of projects have to be selected to be able to generate samples of classes. For instance, with less than 89 projects the double sampling may not be able to generate samples, depending of their size. This is due to the fact that we have chosen GitHub as a population of projects and that GitHub mainly contains small projects, which contain few classes. To verify this point, we have measured the distribution of Java classes contained in projects stored by GitHub. To that extent, we have randomly selected 400 projects and measured how many Java classes they contain. Figure 4 shows an histogram of this distribution. It clearly shows that most of the projects contain very few classes and that few projects contain lots of classes. More precisely, 90% of projects contain less than 250 classes and 2.5% of projects contain more than 1000 classes. Therefore, by choosing 55 projects, the double sampling will probably select 1 or 2 big projects, which is why it may fail to generate large samples (containing 2,000 classes). However, by choosing 144, 233 or 377 projects, the double sampling will probably select 3 to 10 big projects, which is why it will be able to generate large samples of classes.

The second main result that has been obtained by our validation is a measure of the effect of the number of selected projects and classes on the quality of the threshold.

Table 1: Root Mean Square Errors for estimated 90% thresholds of the NOA+NOM metric

	Number of Classes									
Number of Repositories	200	400	600	800	1000	1200	1400	1600	1800	2000
3	∅	∅	∅	∅	∅	2.09	∅	∅	∅	∅
5	3.57	∅	∅	∅	∅	∅	∅	∅	∅	∅
8	∅	3.57	∅	∅	2.09	∅	∅	∅	∅	∅
13	3.57	5.47	∅	∅	∅	∅	∅	∅	∅	∅
21	7,63	5.33	6.40	∅	∅	∅	12.24	2.08	∅	∅
34	2.08	3.12	3.89	2.83	6.40	∅	6.40	3.62	∅	∅
55	4.96	3.57	2.32	5.33	2.05	∅	2.32	4.96	∅	3.57
89	3.89	7.63	3.57	6.40	2.83	5.33	2.83	4.33	2.45	2.09
144	2.08	3.38	2.84	6.40	2.84	2.08	3.12	2.09	2.32	2.45
233	2.08	2.08	2.05	2.08	2.45	2.08	2.04	2.45	2.08	2.45
377	3.38	3.38	6.40	2.45	4.12	2.56	2.56	2.08	2.45	2.83

Table 2: Root Mean Square Errors for estimated 90% thresholds of the CBO metric

	Number of Classes									
Number of Repositories	200	400	600	800	1000	1200	1400	1600	1800	2000
3	∅	∅	∅	∅	∅	3.23	∅	∅	∅	∅
5	3.23	∅	∅	∅	∅	∅	∅	∅	∅	∅
8	∅	4.79	∅	∅	3.30	∅	∅	∅	∅	∅
13	7.26	5.74	∅	∅	∅	∅	∅	∅	∅	∅
21	10.55	4.42	10.55	∅	∅	∅	8.85	3.92	∅	∅
34	2.75	2.94	2.92	5.92	5.92	∅	4.79	4.71	∅	∅
55	5.73	2.72	4.79	3.92	7.26	∅	8.85	4.71	∅	5.91
89	5.37	3.31	3.92	3.31	7.26	3.92	2.75	4.79	3.87	2.94
144	3.51	3.51	2.92	8.85	3.23	3.31	4.17	3.23	3.87	2.91
233	3.23	2.72	3.87	2.72	2.94	2.72	2.92	4.79	3.92	3.31
377	3.92	3.23	2.92	2.75	4.71	3.51	3.87	2.72	2.94	2.72

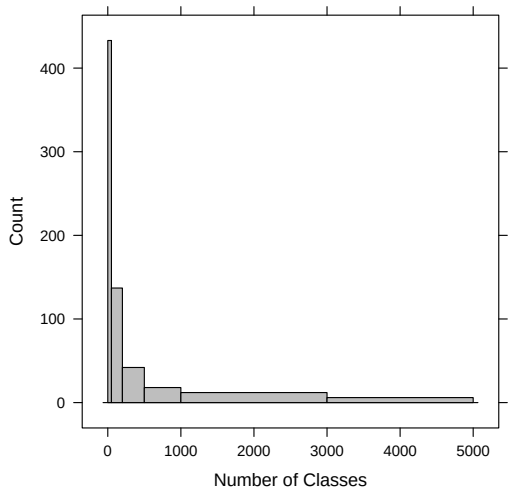

Figure 4: Number of Java classes per project stored by GitHub.

If we focus on values obtained with 89 projects or more, then the errors are under 10%. However, if we focus on values obtained for 144 projects or more and for 1,000 classes or more then the errors are very low (under 5%). We then have performed ANOVA test and regression tests for this subsets of values but no significant results were observed. This means that once the size of the sample is sufficient to be representative, there is no need to include more data in it, or at least this won't improve the quality of the estimation. This corresponds to any statistical approaches that advice to build a representative sample with a limited size. For instance, lots of samples performed for questionnaire contains around 1,000 people.

As a minor result regarding the intent of our validation, we can provide threshold of the two metrics. In particular, for the NOA+NOM metrics, the best precision is obtained with 233 projects and 1,400 classes (error = 2.04) that means that the obtained quantile is 90% more or less 2.04%. The value of the threshold is then 25. For the CBO metrics, the best precision is obtained with 233 projects and 400 classes (error = 2.72). The value of the threshold is then 27.

3.3 Efficiency

The previous section has shown that once the sample is big enough, the quality of the threshold is good at more or less 5%. However, the more selected projects and classes, the more time it takes to compute the threshold. Thus it is important to be able to estimate the time required to compute a threshold for a given context in order to configure correctly the process. To that extent, we have measured the time needed to compute a threshold by our process. In particular, we measured the time needed by the double sampling and the time needed by Bootstrap. We have chosen not to measure the time needed to compute the metrics for the entities of the sample as it depends on the metrics and on how it has been implemented. All the measures have been recorded on a computer equipped with a 2.90GHz Intel®Core I7 processor, with 8GB of RAM and running on a 64-bits Windows 7.

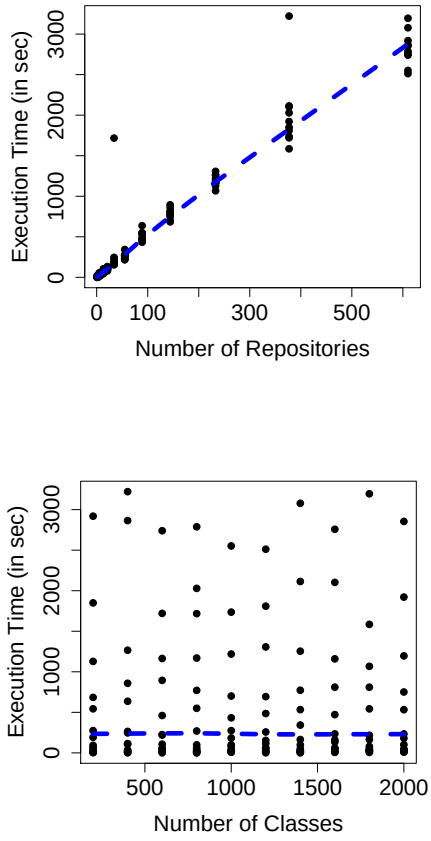

Figure 5: Double Sampling Execution Time

The Figure 5 presents the time needed by the double sampling and clearly shows that it grows linearly with the number of projects but has almost no relationship with the number of classes. The reason is because the double sampling requires to make a clone of the selected project to create the limited population. Making a clone takes some time as it requires to download the project from GitHub and to extract it on the local disk. Even if there are much more small projects than big ones in GitHub, it takes 10 minutes to clone 100 projects in average. Once the selected projects have been extracted, it takes almost the same time to select randomly some classes from their latest revision, whatever their number. This is why the time needed by the double sampling does not depend on the number of classes.

The Figure 6 presents the time needed by Bootstrap and clearly shows that it grows linearly with the number of classes but has almost no relationship with the number of projects. The reason is because Bootstrap takes much more time to create its samples when the original sample is big. In average, Bootstrap needs 25 seconds to create a threshold with a sample of 1,000 classes. The number of projects has no influence on the time needed by Bootstrap as Bootstrap just input the sample of classes, whatever how many projects have been selected.

As a indication, if we want 233 repositories and select 1,200 classes from them it takes approximately half an hour

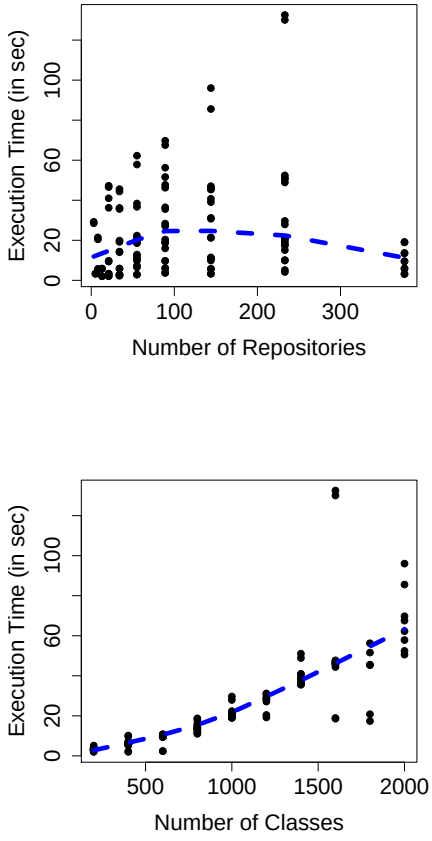

Figure 6: Bootstrap Execution Time

Table 3: Manual samples of repositories

Sample	Repository	Number of Classes
M_A	Eclipse Platform	744
	Jetty	1630
M_B	Vuze	365
	Weka	1420
	JWebMail	113

Table 4: Correctness of the thresholds based on manual samples

Sample	RMSE
M_A	5.46
M_B	6.67

to compute the two thresholds introduced the previous section.

4. DISCUSSION

Our approach aims to compute threshold of metrics thanks to a statistical approach. Its main principles are based on random selection and quantile computed by Bootstrap. Each of these two principles are subjects to discussion.

4.1 Random selection

Random selection is the cornerstone of any approach that is based on statistics. In our context, software entities must be selected randomly to ensure representativeness of the computed threshold. Nevertheless, studies frequently rely on a manual selection of projects to compute thresholds [22]. To confirm the importance of random selection we built two samples composed of projects manually selected and we compute the threshold of the metrics from them. Table 3 presents these two samples, composed of software projects used in the literature. They contain about the same number of classes.

We performed the same validation used in section 3 that consists in looking on 10 sets of 100 projects on how a computed threshold identify the entities. Then we can compare the obtained quantile with the targeted one. Table 4 presents the results obtained for the NOA+NOM metrics. This result shows that the errors are higher with thresholds obtained with manual samples than with the threshold computed with our process and with less data (144 projects and 1000 classes). Furthermore, when using manual sampling no guarantee can be provided regarding the quality of the computed thresholds and no configuration can be tuned to improve the obtained results.

If random selection is central to our approach, then the population of projects from which random selection is performed is crucial. As presented in the last section, we choose to use all GitHub Java projects. As a consequence, the thresholds computed by our approach are representative of that population. Hence GitHub, our starting population could be seen as one parameter of the context or it could be considered that GitHub is a representative source for open source projects.

In our experiment we did choose to put the project size as a parameter of the context. One could argue that tak-

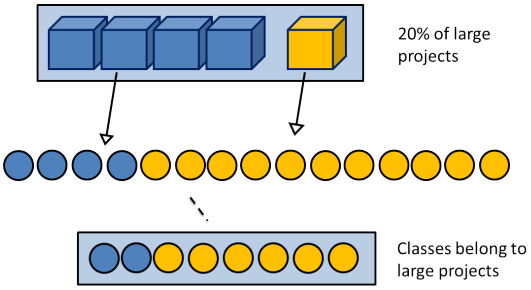

Figure 7: Example of double sampling with large and small projects. Classes of the sample belong to large projects.

ing the corresponding population should not be considered as it contains too many projects that are nearly empty. Projects nearly empty contain few classes and therefore if many of them are selected the population of classes will contain much more classes belonging to large projects than classes contained in small projects. To illustrate this, the Figure 7 presents a sample of projects that contains only one large project and four small projects. The double sampling will however return a sample of classes that will mainly belong to the large project. To ensure this behaviour, we have computed a threshold for the NOA+NOM metrics with 144 projects and 1,000 classes coming from a population composed only of Java projects containing more than 100 classes and with more than one year of history. The returned threshold was the same than one obtained with the whole population of GitHub projects.

4.2 Quantile computed by Bootstrap

We focused our validation on two metrics, NOA+NOM and CBO, which follow a power law, as many other metrics [16]. Moreover, the values of these metrics have a somehow quite large range. The Figure 8 shows that the NOA+NOM metrics follow a power low and that its values range from 0 to 50 (and even more for outliers that do not appear in the figure). However Bootstrap could also handle metrics that do not follow a power law as it is not dependent of the distribution of metrics values.

5. RELATED WORK

In [1], Alves et al. identify three families of approaches to compute threshold. The first family of approaches is based on the knowledge of experts who define arbitrarily thresholds according to their own experience. For instance, McCabe defined 10 as a threshold for its cyclomatic complexity metric [17]. The second family of approaches is based on a correlation analysis performed to identify causality between metrics and error proneness. In particular, D'Ambros et al. have done an extensive comparison for such a family of approaches [7]. The third family of approaches, is based on a statistic analysis of a large set of software entities. The main idea is to measure a large set of entities and to define metrics thresholds thanks to statistics [22, 14]. Our approach falls in the third family as it uses statistics to compute threshold.

Regarding this third family, many approaches use different kinds of statistics to compute thresholds[15, 11, 5, **?**, 24, 1, 22, 2, 14, 12]. However all of these approaches aim at finding generic thresholds that holds for all software systems. None

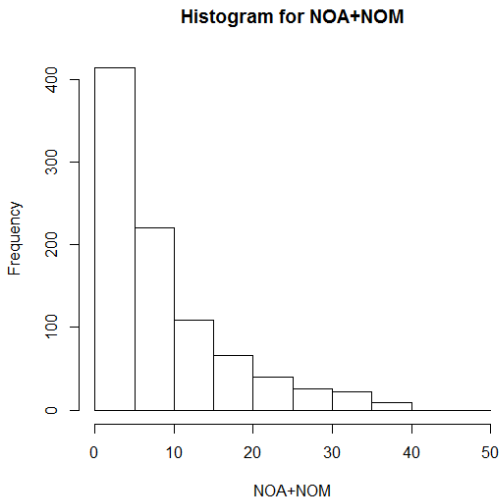

Figure 8: The NOA+NOM metric follows a power law and its values range from 0 to 50.

of them takes the context as an input and none of them proposes an deep investigation of the representativeness and efficiency issues.

Nagappan *et al.* have discussed about representativeness in MSR studies [18, **?**]. Based on a complete analysis of a population of interest, they propose metrics to measure the representativeness of a sample. Their intent is however to discuss about coverage. The process they propose to compute a sample outputs a small set of software entities where each entity represents a group of similar entities. Such a sample then covers all entities of a population but does not represent their distributivity, which is needed for our concern.

Finally, the theory of sampling in statistics has been the subject of numerous studies in the literature. For detailed analysis of the various existing techniques, we refer the reader to the excellent book of Thompson [**?**]. The importance of the size and the quality of the sample has been largely discussed, especially in the literature of health sciences [21, 23].

6. CONCLUSION AND FUTURE WORK

In this paper we propose a process to compute thresholds of metrics. Our process is based on two principles that are the use of the double sampling and the use of quantile computed by Bootstrap. The main advantages of the double sampling is first to support random selection of both projects and software entities, and second to not include all the software entities of selected projects. It thus aims to avoid the impact on manual selection and to minimize the bias of selecting all entities of selected projects. The advantage of using quantile computed by bootstrap is to be independent of the distribution law of the metrics and to provide a significant estimation. We argue that a quantile is a good requirement to generate a threshold based on a statistical approach.

We have provided a strong validation of our process by computing thresholds of two metrics (NOA+NOM and CBO). Our validation has been done on GitHub projects. The ob-

ject of our validation was to compute thresholds of metrics with a 90th percentile as a reached quantile. By analyzing at least 144 projects and 1,000 classes our process returns thresholds with a quite good quality (less than 5% of error). Our validation has also shown that using much more projects and classes provides no real benefit. Finally, our validation has shown that our process needs approximatively half an hour to compute thresholds.

Based on our validation, it clearly appears that double sampling and quantile computed by Bootstrap give interesting results but raise some discussions. In particular, the definition of the base population of projects has an impact on the returned thresholds. For GitHub, if the base population includes all stored projects then many projects have to be selected (more than 144) to be able to compute a threshold of high quality. This is due to the fact that GitHub contains a lot of small projects. The metrics themselves have a high impact on the results provided by our process. For instance, our process fails in computing threshold for DIT. This is due to the fact that DIT has a very small range of values. As a consequence, Bootstrap fails to generate a significant estimation.

As a general concern, our study can be seen a first step towards a thorough investigation of the importance of sampling for the computation of summary statistics. Regarding this general goal, our proposal pinpoints some of the benefits and limitation of the double sampling and Bootstrap. Secondly, it gives some feedbacks in terms of quality and efficiency.

In our future work, we aim at using our process not only for computing thresholds of metrics but to get general summary statistics that are needed in some MSR studies. For instance, our process can be used to measure the popularity of programming languages or the frequency of commits. Our objective is then to generalize the results that were obtained. We also plan to realize a complete comparison of other sampling designs that are defined in the literature.

7. REFERENCES

[1] T. L. Alves, C. Ypma, and J. Visser. Deriving metric thresholds from benchmark data. In *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, pages 1–10. IEEE Computer Society, 2010.

[2] T. Bakota, P. Hegedus, P. Kortvelyesi, R. Ferenc, and T. Gyimothy. A probabilistic software quality model. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, page 243âĂŞ252, Washington, DC, USA, 2011. IEEE Computer Society.

[3] S. Benlarbi, K. E. Emam, N. Goel, and S. Rai. Thresholds for object-oriented measures. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, ISSRE '00, page 24âĂŞ, Washington, DC, USA, 2000. IEEE Computer Society.

[4] A. Canty and B. D. Ripley. *boot: Bootstrap R (S-Plus) Functions*. 2013. R package version 1.3-9.

[5] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Trans. Softw. Eng.*, 24(8):629âĂŞ639, Aug. 1998.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476âĂŞ493, June 1994.

[7] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In J. Whitehead and T. Zimmermann, editors, *MSR*, pages 31–41. IEEE, 2010.

[8] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, page 339âĂŞ348, Washington, DC, USA, 2001. IEEE Computer Society.

[9] B. Efron. Bootstrap methods: another look at the jackknife. *The annals of Statistics*, page 1âĂŞ26, 1979.

[10] B. Efron. Better bootstrap confidence intervals. *Journal of the American Statistical Association*, 82(397):171âĂŞ185, 1987.

[11] K. Erni and C. Lewerentz. Applying design-metrics to object-oriented frameworks. In *Proceedings of the 3rd International Software Metrics Symposium*, pages 64 –74, Mar. 1996.

[12] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha, L. F. Mendes, and H. C. Almeida. Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, 85(2):244–257, Feb. 2012.

[13] W. Harrison. Software measurement: A decision-process approach. In Marshall C. Yovits, editor, *Advances in Computers*, volume Volume 39, pages 51–105. Elsevier, 1994.

[14] S. Herbold, J. Grabowski, and S. Waack. Calculation and optimization of thresholds for sets of software metrics. *Empirical Software Engineering*, 16(6):812âĂŞ841, 2011.

[15] M. Lorenz and J. Kidd. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[16] P. Louridas, D. Spinellis, and V. Vlachos. Power laws in software. *ACM Trans. Softw. Eng. Methodol.*, 18(1):2:1âĂŞ2:26, Oct. 2008.

[17] T. J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308âĂŞ320, 1976.

[18] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. Representativeness in software engineering research, 2012.

[19] N. Moha, Y. GueÌĄheÌĄneuc, L. Duchien, and A. Le Meur. DECOR: a method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, 36(1):20–36, 2010.

[20] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, page 452âĂŞ461, New York, NY, USA, 2006. ACM.

[21] M. X. Patel, V. Doku, and L. Tennakoon. Challenges in recruitment of research participants. *Advances in Psychiatric Treatment*, 9(3):229–238, May 2003.

[22] R. Shatnawi, W. Li, J. Swain, and T. Newman. Finding software metrics threshold values using ROC curves. *Journal of Software Maintenance and*

Evolution: Research and Practice, 22(1):1âĂŞ16, 2010.

[23] S. Straus MD, P. Paul Glasziou MB BS PhD FRACG, Scott Richardson MD, and Brian Haynes MD. *Evidence-Based Medicine: How to Practice and Teach it (Straus, Evidence-Based Medicine)*. Churchill Livingstone, Dec. 2010.

[24] K.-A. Yoon, O.-S. Kwon, and D.-H. Bae. An approach to outlier detection of software measurement data using the k-means clustering method. In *First International Symposium on Empirical Software Engineering and Measurement, 2007. ESEM 2007*, pages 443–445, 2007.

[25] F. Zhang, A. Mockus, Y. Zou, F. Khomh, and A. E. Hassan. How does context affect the distribution of software maintainability metrics? In *Proceedings of the 29th IEEE International Conference on Software Maintainability*, ICSM '13, 2013.