EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

Seeing the forest for the trees with new econometric aggregation tchniques

Citation for published version (APA):
Serebrenik, A., Brand, van den, M. G. J., & Vasilescu, B. N. (2012). Seeing the forest for the trees with new econometric aggregation tchniques. *ERCIM News*, *88*, 21-22.

Document status and date:
Published: 01/01/2012

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 17. Feb. 2025

- *What to (re)test?* To minimize bug resolution time, it is good practice to have a fine-grained suite of unit tests that is run frequently. If a small change to a program causes a test to fail, then the location of the defect is immediately clear and it can be resolved right away. Unfortunately, for large systems the complete suite of unit tests runs for several hours, hence software engineers tend to delay the testing to a batch mode. In that case if one unit test fails the cause of the defect is unclear and valuable time is lost trying to identify the precise location of a defect. With our research, we are able to identify which unit tests are affected by a given change, which enables us to run the respective unit tests in the background while the software engineer is programming.

"In Vitro" Vs. "In Vivo" Research

In this kind of research, industrial collaboration is a great asset. In the same way that a biologist must observe species under realistic circumstances, so must computer scientists observe software engineers in the process of building software systems. Industrial collaboration therefore shouldn't be seen as a symptom of "applied" research but rather be viewed as fundamental research in software engineering practices.

To stress the fundamental research angle, we adopt the terms "in vitro" and "in vivo". Indeed, just like in life-sciences, we take methods and tools that have proven their virtue in artificial lab conditions ("in-vitro research") and apply them in uncontrolled, realistic circumstances ("in-vivo research"). This can sometimes make a big difference. Referring back to the misclassified bug reports mentioned earlier, we originally designed the experiment to verify whether text-mining algorithms could predict the severity of new bug reports. However, from discussions with software engineers in bug triage teams, we learnt that this wouldn't be the best possible use case for such an algorithm because they rarely face problems with incoming bug reports. Also, close to the release date, pressure builds up and it is then they want to verify whether the remaining bug reports are classified correctly. Such observations can only be made by talking to real software development teams; mailing lists and software repositories are a poor substitute.

Luckily our research group has a strong reputation in this regard. Indeed, our PhD students actively collaborate with software engineers in both large and small organisations producing software intensive systems. As such we remain in close contact with the current state of the practice and match it against the upcoming state of the art.

Link:
http://ansymo.ua.ac.be/

Please contact:
Serge Demeyer, Ahmed Lamkanfi and Quinten Soetens
ANSYMO research group
University of Antwerp, Belgium
E-mail: serge.demeyer@ua.ac.be

Seeing the Forest for the Trees with New Econometric Aggregation Techniques

Alexander Serebrenik, Mark van den Brand and Bogdan Vasilescu

Understanding software maintainability often involves calculation of metric values at a micro-level of methods and classes. Building on these micro-level measurements and complementary to them, econometric techniques help to provide a more precise understanding of the maintainability of software systems in general.

Maintaining a software system is like renovating a house: it usually takes longer and costs more than planned. Like a house owner preparing a condition report identifying potential problems before renovation, a software owner should assess maintainability of software before renovating or extending it. To measure software maintainability one often applies software metrics, associating software artifacts with numerical values. Unfortunately, advanced software metrics are commonly measured at a level of small artifacts, eg methods and classes, and fail to provide an adequate picture of the system maintainability. Continuing the analogy, the state-of-the-art in software metrics corresponds to a condition report detailing the state of every brick and obscuring the general assessment in the multitude of details. Metrics visualization techniques provide a general overview of the measurements but have difficulties with presenting evolution of these measurements in time.

To see the forest of a software system for the trees of individual measurements, aggregation techniques are used. Current aggregation techniques are, however, usually unreliable or involve human judgment. For instance, the mean is known to become unreliable in the presence of highly skewed distributions, which are typical for software metrics. Another approach, distribution fitting, consists of manually selecting a known family of distributions and fitting its parameters to approximate the observed metric values. The fitted parameters can then be seen as the aggregation results. However, the fitting process should be repeated with each new metric considered. Moreover, it is still a matter of controversy whether, for instance, software size is distributed log-normally or double Pareto. Finally, threshold-based approaches assume the availability of a set of metric thresholds: depending on which thresholds the metric value of an individual element (eg method or class) exceeds, the value is classified as being poor, mediocre or good. If this is the case, the entire system is labelled as being poor, mediocre or good depending on the percentage of poor, mediocre and good values. By definition, threshold-based approaches require the presence of either commonly accepted thresholds

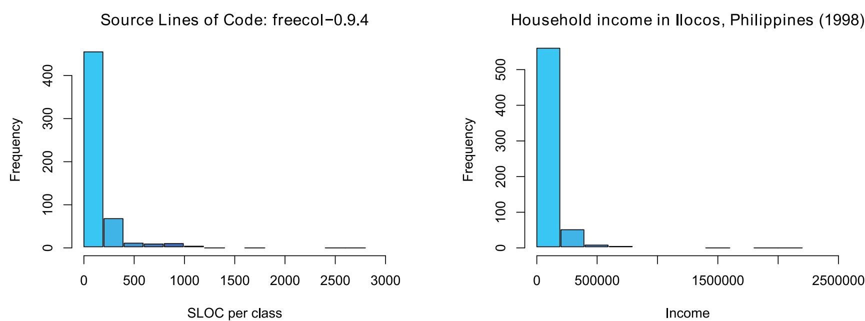

Figure 1: Software metrics (SLOC) and econometric variables (household income in the Ilocos region, the Philippines) have distribution with similar shapes.

or a large benchmark collection to derive such thresholds. Unfortunately, neither the public thresholds nor the composition of the benchmark collections are commonly accepted or can be scientifically validated. Furthermore, the threshold approaches assume that if all individual values are good, then so is the quality of the entire system. This, however, is not the case for such metrics as the depth of inheritance tree.

In an on-going project that commenced in 2010 a research team from Eindhoven University of Technology, The Netherlands (Alexander Serebrenik, Mark van den Brand and Bogdan Vasilescu) is investigating application of econometric inequality indices as aggregation techniques. Inequality indices are econometric techniques designed to measure and explain inequality of income or welfare distributions. Their application to aggregation of software metrics is based on the observation that numerous countries have few rich and many poor, and similarly, numerous software systems have few very big or complex components and many small or simple ones. Inequality indices combine the advantages of the preceding aggregation techniques and avoid their disadvantages. Similarly to distribution fitting, they provide reliable results for highly-skewed distributions. Similarly to the mean they do not require complex application procedures. Moreover, application of inequality indices does not require the availability of well-established thresholds or extensive benchmarking.

In 2010 we advocated the use of the Theil index, one of the econometric inequality indices, to measure and explain inequality among software metric values. We observed that inequality in file sizes of the Linux Debian distribution lenny can be better explained by the package these files belong to, rather than the implementation language. This suggests that if one would like to reduce this inequality, ie distribute functionality across the units in a more egalitarian way, one should focus on establishing cross-package size guidelines. In 2011 we conducted an extensive empirical comparison of different inequality indices and determined guidelines when different inequality indices should be used for software metrics aggregation. In another recent work we have applied inequality indices to studying the impact of different software project attributes on the development effort. We found that such project attributes as the primary programming language, the organization type, and the year of the project have a higher impact than the development platform or the intended market.

Our current research, carried out in cooperation with Inria, Université de Bordeaux and University of Paris 8 (France), compares inequality indices with a threshold-based technique called Squale. We have found a close theoretical relationship between the two, and we have further conducted an empirical evaluation of the approaches proposed. Even more recently, together with the colleagues from Software Engineering Lab of University of Mons (Belgium), we have been investigating the application of inequality indices to studying developer specialization in open-source projects.

As the most important direction of the future work we consider integration of the inequality indices proposed in predictive models. We believe that the ability of inequality indices to explain inequality of the metric values observed will provide for more precise prediction of the metric values.

Please contact:
Alexander Serebrenik
Eindhoven University of Technology, Eindhoven, The Netherlands
Tel: +31402473595
E-mail: a.serebrenik@tue.nl

Continuous Architecture Evaluation

by Eric Bouwers and Arie van Deursen

Keeping the implementation of a software system in-line with the original design of the system is major challenge in developing and maintaining a system. One way of dealing with this issue is to use metrics to quantify important aspects of the architecture of a system and track the evolution of these metrics over time. We are working towards extending the set of available architecture metrics by developing and validating new metrics aimed at quantifying specific quality characteristics of implemented software architectures.

Most software systems start out with a designed architecture, which documents important aspects of the design such as the responsibilities of each of the main components and the way these components interact. Ideally, the implemented architecture of the system corresponds exactly with the design. Unfortunately, in practice we often see that the original design is not reflected in the implementation. Common deviations are more (or fewer) components, undefined dependencies between components and components implementing unexpected functionality.