Using Machine Learning to Design a Flexible LOC Counter

Miroslaw Ochodek*, Miroslaw Staron†, Dominik Bargowski‡, Wilhelm Meding§, Regina Hebig¶
*Poznan University of Technology, Poland
miroslaw.ochodek@cs.put.poznan.pl
†Chalmers — University of Gothenburg, Sweden
miroslaw.staron@gu.se
‡ Poznan University of Technology, Poland
dominik.bargowski@student.put.poznan.pl
§Ericsson AB, Sweden
wilhelm.meding@ericsson.com
¶Chalmers — University of Gothenburg, Sweden
regina.hebig@cse.gu.se

***Abstract*—Background: The results of counting the size of programs in terms of Lines-of-Code (LOC) depends on the rules used for counting (i.e. definition of which lines should be counted). In the majority of the measurement tools, the rules are statically coded in the tool and the users of the measurement tools do not know which lines were counted and which were not.**

Goal: The goal of our research is to investigate how to use machine learning to teach a measurement tool which lines should be counted and which should not. Our interest is to identify which parameters of the learning algorithm can be used to classify lines to be counted.

Method: Our research is based on the design science research methodology where we construct a measurement tool based on machine learning and evaluate it based on open source programs. As a training set, we use industry professionals to classify which lines should be counted.

Results: The results show that classifying the lines as to be counted or not has an average accuracy varying between 0.90 and 0.99 measured as Matthew's Correlation Coefficient and between 95% and nearly 100% measured as the percentage of correctly classified lines.

Conclusions: Based on the results we conclude that using machine learning algorithms as the core of modern measurement instruments has a large potential and should be explored further.

I. Introduction

Automated measuring instruments and measuring systems in software engineering are often based on predefined algorithms which count entities of specific kinds (or with specific properties). If they are not based on predefined algorithms then they provide a limited set of configuration parameters. As the properties of the entities evolve and the measurement needs evolve so do the measuring instruments. In practice, this evolution causes maintenance effort for the companies. In this paper, we explore the idea of using machine learning as a means to cope with the problem of the evolution of the measuring instruments. Instead of rewriting the code of the measuring instruments, we can re-teach the machine learning algorithm to change the way which the entities are counted.

In this paper, we address the problem of how to accurately count lines of code without the need to define a complete grammar for the programming language used. In particular, our aim is to explore how to use machine learning algorithms to find the classification rules for which lines should be included. The first step of our research is to list a set of potential features (characteristics) which can be used to train the machine learning algorithms on how to classify lines of code, thus our research question is:

Which features of lines of code can be used to train machine learning algorithms when classifying lines of code as included in counting?

Our research method is a design science research to explore a set of potential features, design the prototype and evaluate its applicability. The results from the evaluation show that the accuracy of the method is very promising, which means that there is a large potential in using machine learning algorithms as the core of measurement instruments. However, further experiments with "worst-case" scenarios are needed in order to understand the limitations of our approach.

II. Lines of code counters

In this paper we use the measure of Lines of Code (LOC) as an object of the study as this measure is simple and easy to conceptually link to the problems of teaching an algorithm how to recognize the objects (lines) which should be counted and which should not. The measure has been used in practice since the 1950s and there is substantial body of research on it, [2]. It is also used as an input variable to many prediction models – e.g. the Constructive Cost Model (COCOMO) and its newer versions [3].

LOC measure is often also called SLOC (Source Lines of Code) as an acronym and has multiple variations, for example:

1) Physical (Source) Lines of Code – measure of all lines, including comments, but excluding blanks

2) Effective Lines of Code – measure of all lines, excluding: comments, blanks, standalone braces, parentheses.
3) Logical Lines of Code – measure of those lines which form code statements.

III. Machine-Learning-based LOC counter

We assume that a flexible LOC counter should be programming-language agnostic. Therefore, if it is possible, we do not want to rely upon any specific language-specific parsers. We would like to treat the code as it was a plain text and perform classification at the level of a single line of code instead of block constructs. Such approach limits the necessity to modify the code of the counter to apply it to a new programming language.

A. Features

We consider two approaches to deriving features describing lines of code. In the first approach, the features are defined a priori and extracted from the text either by measuring the quantitative aspects of the text or checking the occurrence of certain patterns. The currently supported list of such features is presented in Table I. Although we use regular expressions instead of language-specific parsers, it seems clear that the features F05–F19 refer to constructs known from programming languages. Consequently, such approach is not fully programming-language agnostic. However, it is still based on the constructs (and keywords) that are present in most of the modern programming languages. As an alternative, we propose a method of automatic acquisition of features based on the analysis of a code base.

The proposed approach is based on the frequency analysis of tokens appearing in the code (so called bag-of-words approach). We tokenize each line in a file using white and special characters: ()[]{}!@#$%^&*-=;:'"\|‘~,.<>/?. We also preserve the split strings as tokens. We count the frequency of occurrence of the tokens in each file and the whole code base. We define two thresholds for accepting a token as a candidate feature:

- min. frequency — the minimal number of occurrences of the token in the code base;
- min. files covered — the minimal percentage of files that have at least one occurrence of the token.

The thresholds allow us to filter out tokens that appear frequently, but only in a small number of files (e.g., local variables or API calls). We set the default values of the thresholds based on our experience and trail and error to 25% for the minimum frequency and to 5 for the minimum files covered. We observed that such settings allow us to sufficiently eliminate tokens appearing only in a single module or in a small number of files. However, we believe that using a more systematic approach to tune these parameters could still improve the results. Finally, we define a feature for each candidate token meeting the thresholds as F_i: the number of times the token_i occurs in a line of code (Numeric).

Table I
The list of predefined features describing a line of code.

ID	Name	Type	Description
F01	File extension	Nominal	The extension of the file (e.g., java, cpp, etc.)
F02	Full length	Numeric	The number of characters in the line.
F03	Length	Numeric	The number of characters in the line after removing all leading and trailing white characters.
F04	Tokens	Numeric	The number of tokens in the line (the line is split based on white characters).
F05	Semicolons	Numeric	The number of semicolons in the line.
F06	Comments	Boolean	The line includes any of //, /*, */ or after trimming starts with *.
F07	Assignments	Numeric	the number of single assignment signs in the line (=).
F08	Brackets	Numeric	The number of brackets: (,)in the line.
F09	Square brackets	Numeric	The number of square brackets: [,] in the line.
F10	Curly brackets	Numeric	The number of curly brackets: {, } in the line.
F11	Class	Boolean	The word "class" appears in the line.
F12	For	Boolean	The word "for" appears in the line.
F13	If	Boolean	The word "if" appears in the line.
F14	While	Boolean	The word "while" appears in the line.
F15	Case	Boolean	The word "case" appears in the line.
F16	Try	Boolean	The word "try" appears in the line.
F17	Catch	Boolean	The word "catch" appears in the line.
F18	Expect	Boolean	The word "expect" appears in the line.
F19	Member access	Numeric	Counts members accessors: . or ->

B. Classification algorithm

We consider a standard, binary classification problem (decision classes are *Count* and *Ignore*) with numerical and categorical variables. Therefore, it could be handled by most of the available classification algorithms. However, taking into account the practical application of the proposed approach, we require that the model constructed by a classification algorithm has a form of a white-box, and allows the user to analyze how a given decision was made.

Currently, we experiment with rules- and tree-based classifiers. In particular, we tested the approach using tree algorithms available in the WEKA package [13]:

- J48 — an implementation of the C4.5 decision tree-based classifier [21];

Figure 1. An example of a report generated by the CCFlex tool.

- PART — a rule-based classifier that derives rules from C4.5 decision trees [10];
- JRip — a rule-based classifier, which is an implementation of Repeated Incremental Pruning to Produce Error Reduction (RIPPER) [7].

C. Flexible Lines of Code Counter (CCFlex)

To validate the proposed approach, we implemented a prototype tool called Flexible Lines of Code Counter (CCFlex). It is a Java-based console application that allows users to mark the lines of code to count on training sample, train a classifier, and perform LOC size estimation for the files in the code base (LOC = number of lines classified as Count). The tool supports all the predefined features F1–F19 and the automatic acquisition of features based on the frequency analysis. An exemplary report generated by the tool is presented in Figure 1.

IV. Preliminary validation

We decided to preliminary validate the proposed new way of counting LOC by answering the following research questions:

- RQ1: What level of prediction quality can be achieved by the proposed approach?
- RQ2: How the automatic features acquisition affects the classification quality?
- RQ3: How the choice of classification algorithm affects the classification quality?

A. Datasets

We performed validation on a dataset containing samples of code from three Open Source projects developed in Java: Eclipse, Jasper Reports, and Spring MVC. The dataset consisted of 9 files containing 2402 physical lines of code in total (Eclipse: 475 LOC, Jasper Reports 757 LOC, and Spring MVC: 1170 LOC).

The lines of code could be counted in different ways depending on the purpose of the measurement. In this study, we wanted to investigate how the proposed approach handles two kinds of them. Therefore, we created two variants of the dataset using different approaches to mark the lines to count:

- ELOC (Count: 1492 / Ignore: 910) — we asked a senior software architect with over 15 years of experience in the telecom domain to manually classify lines in one of the files and use the same strategy to classify the remaining files in the dataset. The rules used by the architect seemed convergent with the definition of Effective Lines of Code presented in Section II.
- Subjective (Count: 1237 / Ignore: 1165) — we decided to investigate an extreme case by classifying lines according to our subjective assessment of the value of a given line of code. For instance, we ignored comment lines containing only Javadoc formatting tags or some sub-lines of multiline commands that seemed wrapped without reason.

The ELOC variant of the dataset is an example of a systematic approach to counting lines of code, strictly based on the programming language syntax. The Subjective variant corresponds to a more difficult situation when lines of code are counted based on ”subjective” feelings of the expert. According to our knowledge, none of the existing measurement tools can handle such a case.

B. Validation procedure

We performed ten runs of 10-fold cross-validation procedure on the following 18 validation schemes:

- two datasets (ELOC and Subjective);
- three feature sets (All: F01–F19 and acquired automatically; Auto: F01–F04 and acquired automatically; Predefined: F01–F19);
- three classification algorithms (PART, JRip, J48). We used the default parameters for these algorithms provided by WEKA.

C. Prediction quality mesasures

We used a set of the state-of-the-art measures to evaluate the prediction performance, such as Accuracy, Precision, Recall, F-measure, and Matthews Correlation Coefficient (MCC). The measures are calculated based on confusion matrices (see Figure 2) and according to equations 1–5.

$$Accuracy = \frac{\sum_{i=1}^{n} I(t_i = p_i)}{n} \times 100\% \quad (1)$$

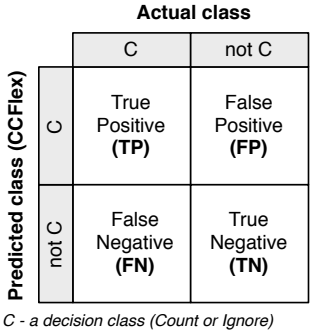

Figure 2. Confusion matrix.

where

- n is the number of lines of code,
- t is a vector of the actual classes of lines,
- p is a vector of the predicted classes of lines,
- I is a function returning 1 if its argument is true and 0 otherwise.

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

$$F\text{-}score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4}$$

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{5}$$

D. Results

The results of the cross-validation are presented in Table II. Depending on the dataset variant, features set, and classification algorithm, the average Accuracy ranged between 95.05% and 99.60%. We did not observe any visible difference between Precision and Recall, which ranged between 0.93 and 1.00. Also, the MCC coefficient, ranging between 0.90 and 1.00, indicated very high prediction quality.

We performed a series of t-tests ($\alpha = 0.05$) to investigate if any of the observed differences could be considered statistically significant.

From the perspective of the choice of the classification algorithm, we observed some statistically significant differences when comparing PART vs. JRip (two significant differences on Recall, F-measure, one on Accuracy and MCC) and J48 vs. JRip (one significant difference on all measures except Precision). None of the differences between PART and J48 seemed significant.

Looking at the same problem from the perspective of the considered features set, we observed difference depending on the variant of the dataset. For ELOC, we observed that using all or predefined features resulted in higher prediction quality than when using automatically acquired features (one statistically significant differences for all the quality criteria). However, for the Subjective variant using all or automatically acquired features provided more accurate results (three statistically significant differences for all the criteria except Precision).

E. Discussion

The observed accuracy of the proposed approach is promising (RQ1). The accuracy was similarly high for both variants of the dataset. Such high accuracy is surprising especially for the Subjective variant because the rules for counting or ignoring the lines were not as strongly related to the Java syntax as in the case of the ELOC variant of the database. It seems that the formal and well-structured nature of source code makes it easy to classify even if the rules are not directly based on the language syntax.

The classifiers using the automatically acquired features performed with nearly the same accuracy as those using the predefined features for the ELOC variant of the dataset and even better for the Subjective variant (RQ2). It is an interesting result, taking into account that the automatic features acquisition makes the approach flexible and programming-language agnostic.

We have also used the learner-based feature selection algorithm[1] on the training data to determine which of the features seemed the most relevant candidates for training classifiers for each variant of the dataset and features set. The selected features are presented in Table III. The first observation was that for the ELOC variant of the dataset only predefined features were selected when all the features were considered. For the Subjective variant, an opposite observation was made.

The choice of the classification algorithm did not visibly affect the results (RQ3). The slight preference was observed towards the PART and J48 classifiers. However, for all considered algorithms the results seemed acceptable from the practical point of view. However, it would also be worth to investigate if the models created by the considered classification algorithms are equally well understandable by the potential users.

We also observed some limitations of the proposed approach. The first problem regards block comments. Currently, we analyze the code line by line. Therefore, only the starting and ending lines of a block comment are easily detectable. The second observed problem is when we have more than one meaningful line of code within a single physical line. To handle such case, we would either have to extend the number of decision classes or develop an additional layer of the algorithm that will estimate the number of lines within a line to count.

[1]WEKA WrapperSubsetEval (classifier: J48) and the BestFirst method (selection based on Accuracy and RMSE, five folds, threshold = 0.01).

Table II
THE RESULTS OF THE PREDICTION QUALITY EVALUATION (AVERAGES, AND STD. DEVIATIONS).

Dataset	Features set	Classifier	Accuracy %	Precision	Recall	F-Measure	MCC
ELOC	All	PART	99.55±0.45	1.00±0.01	1.00±0.00	1.00±0.00	0.99±0.01
ELOC	All	JRip	99.53±0.47	1.00±0.01	1.00±0.00	1.00±0.00	0.99±0.01
ELOC	All	J48	99.60±0.41	1.00±0.01	1.00±0.00	1.00±0.00	0.99±0.01
ELOC	Predefined	PART	99.53±0.46	1.00±0.01	1.00±0.00	1.00±0.00	0.99±0.01
ELOC	Predefined	JRip	99.56±0.46	1.00±0.01	1.00±0.00	1.00±0.00	0.99±0.01
ELOC	Predefined	J48	99.60±0.41	1.00±0.01	1.00±0.00	1.00±0.00	0.99±0.01
ELOC	Auto	PART	99.38±0.47	1.00±0.01	0.99±0.01	0.99±0.01	0.99±0.01
ELOC	Auto	JRip	99.28±0.47	1.00±0.01	0.99±0.01	0.99±0.01	0.98±0.01
ELOC	Auto	J48	99.18±0.54	1.00±0.01	0.99±0.01	0.99±0.01	0.98±0.01
Subjective	All	PART	97.34±1.14	0.98±0.01	0.97±0.02	0.97±0.01	0.95±0.02
Subjective	All	JRip	96.54±1.20	0.98±0.01	0.95±0.02	0.97±0.01	0.93±0.02
Subjective	All	J48	97.18±1.07	0.98±0.01	0.97±0.02	0.97±0.01	0.94±0.02
Subjective	Predefined	PART	95.05±1.45	0.97±0.02	0.93±0.02	0.95±0.01	0.90±0.03
Subjective	Predefined	JRip	95.32±1.44	0.97±0.02	0.93±0.02	0.95±0.02	0.91±0.03
Subjective	Predefined	J48	95.10±1.42	0.97±0.02	0.94±0.02	0.95±0.01	0.90±0.03
Subjective	Auto	PART	97.33±1.08	0.98±0.01	0.97±0.02	0.97±0.01	0.95±0.02
Subjective	Auto	JRip	96.38±1.14	0.98±0.01	0.95±0.02	0.96±0.01	0.93±0.02
Subjective	Auto	J48	97.08±1.09	0.98±0.01	0.96±0.02	0.97±0.01	0.94±0.02

Table III
THE MOST RELEVANT FEATURES SELECTED BY A LEARNER-BASED FEATURE SELECTION ALGORITHM ON THE TRAINING SETS.

ELOC, All	ELOC, Predefined	ELOC, Auto	Subjective, All	Subjective, Predefined	Subjective, Auto
Brackets	Brackets	Freq. of "*"	Assignment	Assignment	Freq. of "*"
Comments	Comments	Freq. of "("	Freq. of "*"	Comments	Freq. of "available"
Semicolons	Full length	Freq. of ";"	Freq. of "available"	If	Freq. of ":"
Full length	Semicolons	Freq. of "/"	Freq. of ":"	While	Freq. of "="
		Full length	Freq. of "has"	Full length	Freq. of "has"
			Freq. of "implied"	Length	Freq. of "implied"
			Freq. of "license"	Semicolons	Freq. of "license"
			Freq. of "none"	Tokens	Freq. of "none"
			Freq. of "reserved"		Freq. of "reserved"
			Freq. of "return"		Freq. of "return"
			Freq. of "see"		Freq. of "see"
			Freq. of "software"		Freq. of "software"
			Full length		Full length
			Length		Length
			Tokens		Tokens

V. RELATED WORK

In the following section, we give a short introduction into related work about measurement theory and lines of code measures as well as related works on the use of machine learning for prediction and measurement in the management of software development.

Software size measures: To apply measurement theory to software engineering Briand et al. [4] redefined basic concepts, such as relational systems, mappings, and scales. As a foundation for defining a general measurement instrument model, properties can be defined for software measures, similar to the definition of relational systems. Briand et al. [5] define such properties, based on groundwork for more general measure properties by Weyuker [27], Zuse [28] and Tian and Zelkowitz [25]. The goal of the use of properties is to ensure correctness of definitions of software metrics. However, the approach cannot remove uncertainties from the measurement process in practice (i.e. lacking a mechanism to guarantee the correct instantiation of the mapping between the empirical world and the relational systems).

With the appearance of new software development paradigms over the years, also methods for size measurement evolved. For example, the introduction of component-based software development was accompanied by a new size

measure: the number of components [9], which was intended as an addition to the measurement of the components' sizes. Similarly, object-oriented software development brought the notion of measuring objects' and methods' size, e.g. [8], [16] and [23]. Armour [1] recognized the problems of using LOC measures when estimating the future size of software and pointed out the future of Function Point Measurement.

However, ambiguity has a long history for many measures, such as lines of code (e.g. as shown by Rosenberg [22]). Park et al. [20] addressed this problem by providing one of the first standardized instructions, focusing on distinct guidelines to recognize physical LOC and logical LOC. Our work is a complement and a different perspective on this problem – trying to find which rules can be applied to recognize the lines to be counted based on the human classification.

This is mainly motivated by previous works. For example, in our own previous work we have studied standard LOC counters and the discrepancy between the results obtained from them [24]. The study showed not just an inconsistency between tools' results, but also that the measurement error can be recognized using statistics. Furthermore, it could be seen that the source of the problem is the differences in the internal rules applied in the LOC counters.

Also Lincke et al. [17] found significant deviations between tools when studying the measurement of size on object-oriented designs. This way, they showed the importance of the notion of systematic and random measurement errors.

Hebig et al. [14] studied the differences in results of size measurement of multi-language programs when using different measures. Having used different estimators of size (e.g. Lines-of-Code, McCabe Complexity) they showed inconsistent between measurement tools. Their results showed discrepancies between tools for multiple estimators.

Machine learning: The main use of machine learning techniques in the management of software development is to predict aspects such as performance (e.g. for data base queries as by Ganapathi, et al. [11]), error rates (e.g. Gondra [12]), or effort.

Challagulla et al.[6] performed an experiment, where they applied different prediction techniques on 4 data sets of software defects. They could show that the combination of machine learning techniques with statistical techniques does not bring an advantage to the pure use of machine learning. Furthermore, Wen et al. [26] performed a systematic review of 84 studies that investigate the precision of effort estimation when using machine learning. The studies include 8 different types of machine learning techniques, of which Case-Base Reasoning, Artificial Neural Networks, and Decision Trees occur most often. They found that in the majority of studies (66%) non-machine learning approaches, such as the Constructive Cost Model (COCOMO) [3], are outperformed by machine learning approaches.

There are a few approaches to using machine learning to retrieve metrics. These can be found in the area of image and video processing. Narwari and Weisi [18] use machine learning to pool features for quality assessment of images. Köstinger, et al.[15] use the machine learning to develop distance metrics for images. Their results show that the learned metric can be orders of magnitudes better than comparable methods. Machine learning was also used to approximate functional size based on use-case names [19].

However, to the best of our knowledge, there is so far no approach that explores the use of machine learning for the creation of code metrics, such as LOC.

VI. Conclusions

In this paper, we explored the idea of using machine learning (decision tree algorithms) as algorithms to classify lines of code whether they should be counted or not. The goal was to study whether we could exchange the static measuring instruments with the more flexible ones, based on machine learning. As shown in the paper the accuracy was very promising between 95% and nearly 100%. This means that the approach can be further developed and we need more experiments with other entities to study the limitations of this approach.

Our future work includes experimentation with more types of programs (to study the impact of the training set on the accuracy), defects and requirements (to study the impact of different entities on the accuracy of the classification) and different similarity algorithms (to study the impact of the classifiers/features on the accuracy of the classification).

Acknowledgment

This research has been partially carried out in the Software Centre, University of Gothenburg, and Ericsson AB.

References

[1] Phillip G Armour. Beware of counting loc. *Communications of the ACM*, 47(3):21–24, 2004.

[2] Barry Boehm. Managing software productivity and reuse. *Computer*, 32(9):111–113, 1999.

[3] Barry Boehm, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, and Richard Selby. Cost models for future software life cycle processes: Cocomo 2.0. *Annals of software engineering*, 1(1):57–94, 1995.

[4] Lionel Briand, Khaled El Emam, and Sandro Morasca. On the application of measurement theory in software engineering. *Empirical Software Engineering*, 1(1):61–88, 1996.

[5] Lionel C Briand, Sandro Morasca, and Victor R Basili. Property-based software engineering measurement. *Software Engineering, IEEE Transactions on*, 22(1):68–86, 1996.

[6] Venkata Udaya B Challagulla, Farokh B Bastani, I-Ling Yen, and Raymond A Paul. Empirical assessment of machine learning based software defect prediction techniques. *International Journal on Artificial Intelligence Tools*, 17(02):389–400, 2008.

[7] William W. Cohen. Fast effective rule induction. In *Twelfth International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann, 1995.

[8] Sergey Diev. Use cases modeling and software estimation: applying use case points. *ACM SIGSOFT Software Engineering Notes*, 31(6):1–4, 2006.

[9] Jose Javier Dolado. A validation of the component-based method for software size estimation. *IEEE Transactions on Software Engineering*, 26(10):1006–1021, 2000.

[10] Eibe Frank and Ian H. Witten. Generating accurate rule sets without global optimization. In J. Shavlik, editor, *Fifteenth International Conference on Machine Learning*, pages 144–151. Morgan Kaufmann, 1998.

[11] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *2009 IEEE 25th International Conference on Data Engineering*, pages 592–603. IEEE, 2009.

[12] Iker Gondra. Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software*, 81(2):186–195, 2008.

[13] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[14] Regina Hebig, Jesper Derehag, and Michel RV Chaudron. Identifying metrics' biases when measuring or approximating size in heterogeneous languages. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–4. IEEE, 2015.

[15] Martin Köstinger, Martin Hirzer, Paul Wohlhart, Peter M Roth, and Horst Bischof. Large scale metric learning from equivalence constraints. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 2288–2295. IEEE, 2012.

[16] Luiz A. Laranjeira. Software size estimation of object-oriented systems. *IEEE Transactions on software engineering*, 16(5):510–522, 1990.

[17] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. Comparing software metrics tools. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 131–142. ACM, 2008.

[18] Manish Narwaria and Weisi Lin. Svd-based quality metric for image and video using machine learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 42(2):347–364, 2012.

[19] Mirosław Ochodek. Functional size approximation based on use-case names. *Information and Software Technology*, 80:73–88, 2016.

[20] Robert E Park. Software size measurement: A framework for counting source statements. Technical report, DTIC Document, 1992.

[21] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[22] Jarrett Rosenberg. Some misconceptions about lines of code. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 137–142. IEEE, 1997.

[23] Miroslaw Staron. Measuring the size of stereotypes, profiles, and stereotyped designs in uml. In *Model Size Workshop*, 2006.

[24] Miroslaw Staron, Darko Durisic, and Rakesh Rana. Improving measurement certainty by using calibration to find systematic measurement error–a case of lines-of-code measure. In *18th KKIO Software Engineering Conference*, 2016.

[25] Jianhui Tian and Marvin V Zelkowitz. A formal program complexity model and its application. *Journal of Systems and Software*, 17(3):253–266, 1992.

[26] Jianfeng Wen, Shixian Li, Zhiyong Lin, Yong Hu, and Changqin Huang. Systematic literature review of machine learning based software development effort estimation models. *Information and Software Technology*, 54(1):41–59, 2012.

[27] Elaine J. Weyuker. Evaluating software complexity measures. *Software Engineering, IEEE Transactions on*, 14(9):1357–1365, 1988.

[28] Horst Zuse. *A framework of software measurement*. Walter de Gruyter, 1998.