Information and Software Technology 56 (2014) 220–237

Contents lists available at ScienceDirect

Information and Software Technology

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

Source code size estimation approaches for object-oriented systems from UML class diagrams: A comparative study

Yuming Zhou [a,b,*], Yibiao Yang [b], Baowen Xu [a,b], Hareton Leung [c], Xiaoyu Zhou [d]

[a] State Key Laboratory for Novel Software Technology, Nanjing University, China
[b] Department of Computer Science and Technology, Nanjing University, China
[c] Department of Computing, Hong Kong Polytechnic University, China
[d] School of Computer Science and Engineering, Southeast University, China

ARTICLE INFO

Article history:
Received 16 October 2012
Received in revised form 18 June 2013
Accepted 19 September 2013
Available online 28 September 2013

Keywords:
Object-oriented
Code size
Estimation
UML
Class diagrams

ABSTRACT

Background: Source code size in terms of SLOC (source lines of code) is the input of many parametric software effort estimation models. However, it is unavailable at the early phase of software development.
Objective: We investigate the accuracy of early SLOC estimation approaches for an object-oriented system using the information collected from its UML class diagram available at the early software development phase.
Method: We use different modeling techniques to build the prediction models for investigating the accuracy of six types of metrics to estimate SLOC. The used techniques include linear models, non-linear models, rule/tree-based models, and instance-based models. The investigated metrics are class diagram metrics, predictive object points, object-oriented project size metric, fast&&serious class points, objective class points, and object-oriented function points.
Results: Based on 100 open-source Java systems, we find that the prediction model built using object-oriented project size metric and ordinary least square regression with a logarithmic transformation achieves the highest accuracy (mean MMRE = 0.19 and mean Pred(25) = 0.74).
Conclusion: We should use object-oriented project size metric and ordinary least square regression with a logarithmic transformation to build a simple, accurate, and comprehensible SLOC estimation model.

1. Introduction

Source code size is a key input of many effort estimation models such as SLIM and COCOMO [27,29,41,42,44–46,50,56,58,59]. Early code size estimation is very important, as it enables software managers to request the required development effort for software projects at the early development phase. This will help software managers to make an accurate quotation, efficiently allocate development resources, and effectively schedule development activities [3]. Due to this reason, early code size estimation has been an important research topic in software metrics community for many years [4–16,18,19,22–30,36,37,39,40,54–56,60,61,63,64,69,70].

In the last decade, many early code size estimation approaches for object-oriented systems based on UML (Unified Modeling Language) class diagrams have been proposed [4–16,18,19,25,69,70]. UML class diagrams describe the structure of a system by showing the classes of the system, their attributes, methods, and the relationships among the classes. In the object-oriented software development, class diagrams are available at the early development phase and are the foundation for generating the source code of the system. Accordingly, it is logical to use the information from class diagrams to estimate SLOC for an object-oriented system. Although many early size estimation approaches have been proposed, few empirical studies have so far investigated the actual accuracy of these sizing approaches when they are applied in practice.

In this paper, we empirically investigate the accuracy of early source code size estimation approaches that use the information from class diagrams to predict the final code size of object-oriented systems. More specifically, we use eight different modeling techniques in combination with/without a logarithmic transformation to investigate the accuracy of six types of inputs to the prediction models. The used modeling techniques include linear models, non-linear models, rule/tree-based models, and instance-based models [68]. The investigated inputs to the prediction models include class diagram metrics, predictive object points, object-oriented project size metric, fast&&serious class points, objective class points, and object-oriented function points [4–10]. Based on 100 open-source Java systems, we attempt to answer the following issues:

- How accurately these estimation approaches can predict the final source code size of object-oriented systems?

* Corresponding author at: State Key Laboratory for Novel Software Technology, Nanjing University, China. Tel.: +86 25 89682450.
E-mail addresses: zhouyuming@nju.edu.cn, cs.zhou.yuming@gmail.com (Y. Zhou).

0950-5849/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2013.09.003

- Which estimation approaches have the highest prediction accuracy?
- Which modeling techniques are the best performing techniques?

These issues are of highly practical value, as they determine whether we might be able to use UML class diagrams to accurately predict software source code size at the early development phase. However, little is currently known on this subject. Our study attempts to fill this gap by a comprehensive investigation.

This paper makes three particular contributions. First, we present new evidence showing an association between the information collected from UML class diagrams and the final code size of object-oriented systems, thereby providing valuable data in an important area for which otherwise there is limited experimental data available. Second, we investigate six different types of metrics collected from UML class diagrams. To the best of our knowledge, this is the first time such a comprehensive set of inputs to the SLOC prediction models have been investigated in the context of object-oriented systems. Third, on the methodological front, we build the prediction models using not just the familiar method of linear models but also non-linear models, tree/rule-based models, and instance-based models.

The rest of this paper is organized as follows. Section 2 describes six types of metrics collected from UML class diagrams, which can be used as the inputs to the SLOC prediction model. Section 3 discusses related work. Section 4 introduces the research method used in this study, including the data set used, the modeling techniques employed, the model evaluation criteria, and the data analysis procedure. Section 5 presents in detail the experimental results. Section 6 concludes the paper and outlines directions for future work.

2. Six types of metrics from UML class diagrams

In this section, we introduce six types of metrics to estimate the source code size of an object-oriented system, which can be collected from its UML class diagrams.

2.1. Class diagram metrics

CDM (class diagram metrics) have been widely used to estimate the source code size of an object-oriented system in the literature [7,11,12,16,18,25]. Table 1 summaries the definitions of the most commonly used CDM available at the analysis phase of an OO software development life cycle [1,43]. As can be seen, these metrics can be classified into two categories:

- *Size metrics*: These metrics count the total number of classes (NC), the total number of attributes (NA), and the total number of methods (NM) in a class diagram. Classes are the fundamental building blocks of an object-oriented system that define the properties and behaviors of the objects. If a class diagram contains more classes, this may increase the source code size of the system. Attributes are the properties of a class possessed by all objects. If a class diagram contains more attributes, this may increase the size of the source code for maintaining these properties. Methods are functions that define the behaviors to be exhibited by objects of the associated class at run time. If a class diagram contains more methods, this may increase the size of the source code for defining these behaviors. Intuitively, NC, NA, and NM are the main factors that influence the source code size of an object-oriented system.
- *Structural complexity metrics*: These metrics measure the complexity of the relationships among the classes in a class diagram, including the association, aggregation, dependency, and generalization relationships. Association and aggregation are object-level relationships. An association represents the static relationship shared among the objects of two classes, where an attribute of one class is an instance of another class. An aggregation is a variant of the association relationship and represents a "whole-part" relationship. Dependency and generalization are class-level relationships. There is a dependency relationship between two classes if one class is a parameter variable of a method of another class. Generalization represents an "is-a" relationship in which one class (the subclass) is a specialized form of another class (the super class). When the number of classes is fixed, for a class diagram: (1) more association/aggregation relationships mean that more code may be needed to maintain the navigation between the objects of the associated classes; (2) more dependency relationships mean that there may be more message passing between the associated classes and hence may increase the source code size of the system; and (3) more generalization relationships mean more code reuse and hence may reduce the source code size of the system. In this sense, the number of these relationships can influence the source code size of an object-oriented system.

From the above analysis, we can see that the source code size of an object-oriented system largely depends on the size and structural complexity metrics collected from its UML class diagram. Therefore, it is reasonable to expect that the prediction model built with these class diagram metrics would produce an accurate estimation of the SLOC of the object-oriented system.

2.2. Predictive object points

The POPs (predictive object points) metric is a size metric designed specifically for object-oriented software by Minkiewicz at PRICE Systems [2,8]. When sizing an OO system, the POPs metric takes into account three aspects of object-oriented software: the amount of raw functionality the software delivers, the communica-

Table 1
UML class diagram metrics.

Category	Metric name	Metric definition
Size	Number of classes (NC)	The total number of classes
	Number of attributes (NA)	The total number of attributes defined in all classes
	Number of methods (NM)	The total number of methods defined in all classes
Structural complexity	Number of associations (NAssoc)	The total number of association relationships
Number of aggregations (NAgg)	The total number of aggregation relationships	Number of compositions (NComp)
The total number of composition relationships	Number of dependencies (NDep)	The total number of dependency relationships
Number of generalizations (NGen)	The total number of generalization relationships	Number of generalization hierarchies (NGenH)
The total number of structures with generalization relationships	Maximum DIT (MaxDIT)	The maximum DIT value. The DIT value of a class in an inheritance hierarchy is the length of the longest path from the class to the root

tion complexity between objects, and the degree of reuse through inheritance. Minkiewicz believes that the size of object-oriented software largely depends on the first dimension and can be substantially influenced by the second and third dimensions.

Based on this rationale, Minkiewicz combines the following four object-oriented metrics to define the POPs metric [2,8,52]:

$$\mathrm{POPs} = \mathrm{AvgWMC} \times \mathrm{TLC} \times \frac{\{1 + [(1 + \mathrm{AvgNOC}) \times \mathrm{AvgDIT}]^{1.01} + (|\mathrm{AvgNOC} - \mathrm{AvgDIT}|)^{0.01}\}}{7.8}$$

where TLC (number of top level classes) is the number of classes that are roots in a class diagram, from which all other classes derived, AvgDIT (average depth of inheritance tree) is the average DIT value for all classes in a class diagram. The DIT for a class is the length of the path from the class to the root of the inheritance hierarchy. AvgNOC (average number of children per class) is the average NOC value for all classes in a class diagram. The NOC for a class is the number of classes that directly inherit from the class. AvgWMC (average number of weighted methods per class) is the average value of the number of methods per class, where each method is weighted by a complexity based on the type of the method, the number of attributes the method modifies, and the number of services this method provides to the system. AvgWMC is the basis for the POPs count, which encompasses both the amount of functionality the software delivers and the communication complexity between objects. Furthermore, the POPs count is adjusted by TLC, AvgDIT, and AvgNOC to account for the effects of reuse through inheritance and overall system size. Consequently, all three dimensions (i.e. raw functionality, the inter-object communication, and reuse through inheritance) of object-oriented software are considered in the POPs metric. In [8], Minkiewicz used a data set consisting of over 20 systems to determine the formula for POPs. These systems were programmed in C++ or Smalltalk and were from various software vendors including military, financial, and commercial. Based on this data set, they performed various regressions and found that the above formula was the best way to combine AvgWMC, TLC, AvgDIT, and AvgNOC to compute POPs. According to [8], the resulting POPs correlated meaningfully with source code size.

In computing the AvgWMC metric, the weight of a method is determined by its type and complexity (as shown in Table 2). On the one hand, the methods in a class are categorized into five method types: constructors (methods which instantiate an object), destructors (methods which destroy an object), modifiers (methods which modify the state of an object), selectors (methods which read the state of an object), and iterators (methods which access all parts of an object in a well-defined order). In Table 2, constructors and destructors are combined into a single type, as they have similar complexity [2]. On the other hand, the complexities of the methods in a class are categorized into three types: low, average, and high. Table 3 summarizes the rules to determine the complexity of a method based on the number of messages it responds to and the number of attributes it modifies. In this context, the number of message responses indicates the quantity of the services that the method provides to the rest of the system. The number of attributes modified indicates the extent of influence that the method has on the system and how it impacts complexities due to inter-object communication [8]. According to [8], the method weightings in Table 2 and the complexity assignment rules in Table 3 were determined by examining the actual amount of effort associated with hundreds of C++ and Smalltalk methods. During this procedure, expert knowledge was also applied. Given Tables 2 and 3, it is easy to compute the AvgWMC metric.

Table 2
Method weightings by type and complexity.

Method type	Method complexity	Weight
Destructors/constructors	Low	1
	Average	4
	High	7
Selectors	Low	1
	Average	5
	High	10
Modifiers	Low	12
	Average	16
	High	20
Iterators	Low	3
	Average	9
	High	15

Table 3
Complexity assignments.

Message responses	Number of attributes modified		
	0 or 1	2–6	7+
0 or 1	Low	Low	Average
2 or 3	Average	Average	Average
4+	Average	High	High

However, AvgWMC is in general unavailable from the UML class diagram at the design phase [2,52]. In this sense, we could not use POPs to build a SLOC prediction model at the early development phase. To address this problem, Minkiewicz suggests using the typical percentage distributions of method types and complexity types to compute AvgWMC. More specifically, we can use the following steps to compute AvgWMC [2,52]:

(1) Compute the average number of methods per class

$$\text{Average methods per class} = \text{the total number of methods} / \text{the total number of classes}$$

(2) Compute the average number of methods in each method type

$$\text{Average constructor/destructor method count} = 20\% \times \text{Average methods per class}$$
$$\text{Average selector method count} = 30\% \times \text{Average methods per class}$$
$$\text{Average modifier method count} = 45\% \times \text{Average methods per class}$$
$$\text{Average iterator method count} = 5\% \times \text{Average methods per class}$$

(3) Compute the number of methods of each complexity type for each method type

$$\text{Low complexity method count} = 22\% \times \text{Average method count}$$
$$\text{Average complexity method count} = 45\% \times \text{Average method count}$$
$$\text{High complexity method count} = 33\% \times \text{Average method count}$$

(4) Apply method weighting rules in Table 2 to compute AvgWMC.

Step 2 divides the average methods per class into four categories according to method type. Step 3 further divides each of these four categories into three categories according to complexity type. Step 4 applies the weights in Table 2 to each of the 12 categories.

Note the percentage distributions of method types and complexity types in step 2 and step 3 were obtained from a manual investigation of source code by Minkiewicz and Fad [2]. In this way, all the four object-oriented metrics used in the POPs metric are available from the UML class diagram. Therefore, we may use POPs to build a SLOC prediction model at the early development phase.

2.3. Object-oriented project size points

The Oops (object-oriented project size points) metric is a size metric for object-oriented software [9]. In a class diagram, the analyst defines class names, attributes, methods and parameters of methods. According to [9,53], each unique token in their names contributes one Point value to Oops. Also, each type including the attribute type and the parameter type contributes one Point value to Oops. More specifically, the Oops metric for a class in the class diagram is calculated as follows:

(1) Initialize: Oops = 0 and TokenSet = {}.
(2) Process class name: for each token in the class name, if it is not in TokenSet, then add the token to the set TokenSet and add 1 to Oops.
(3) Process attributes in the class: first, for each token in an attribute, if it is not in TokenSet, then add the token to the set TokenSet; second, add the number of attributes to Oops.
(4) Process methods in the class: first, for each token in a method/parameter name, if it is not in TokenSet, then add the token to the set TokenSet; second, add the number of parameters to Oops.

The Oops metric for the system is the sum of the Oops points over all classes in the class diagram. Bradine reported that Oops has been successfully used to size object-oriented software at two corporations in Colorado [9].

However, in the above-mentioned Oops metric, it is very difficult to correctly tokenize a name for many cases, as we do not know where a token should start and end. For example, assume that "yellowland" is an attribute name. It is not clear whether this name should be broken into two tokens ("yellow" and "land") or three tokens ("yell", "owl", and "and"). To address this problem, in this paper, we regard each name as a single token when computing the Oops metric. Consequently, the Oops metric can be automatically collected from a class diagram. Therefore, we may use Oops to build a SLOC prediction model at the early development phase.

2.4. Fast&&serious class points

The FS_CP (fast&&serious class points) metric is a composite metric collected from a UML class diagram using a two-step process [4]. At the first step, the class points for each class are counted. At the second step, the FS_CP of the class diagram is obtained by summing the class points over all classes.

For a class c in the given class diagram D, the corresponding class points can be computed by the following steps [4]:

(1) *Compute state points*: First, classify the attributes in the class into three categories: light, heavy, and imported. Light attributes denote simple attributes such as integer and string. Heavy attributes denote complex attributes that have been developed and tested, such as attributes in predefined packages. Imported attributes denote complex attribute of a class not developed and not tested yet. The associated weights for light, heavy, and imported attributes are respectively 1, 3, and 5. Second, compute the number of state points (SP) for this class:

$$\mathrm{SP}(c) = \sum_{i=1}^{\mathrm{numAttr}(c)} \mathrm{Weight}(\mathrm{Attr}_i)$$

where numAttr(c) is the number of attributes in the class c.

(2) *Compute behavior points*: First, compute the PMS (percentage of methods with signature). If PMS is larger than the threshold 70%, then choose the fast approach; otherwise, choose the serious approach. Second, classify the methods in the classes into two categories: trivial and substantial, depending on the chosen Fast or Serious approach. In the fast approach, all methods in the class are classified as substantial with one imported attribute. In the serious approach, a method in the class will be classified as Trivial if its signature has at least 80% light parameters and no imported parameters. Otherwise, this method will be classified as Substantial (one Imported attribute will also be assigned if it has no signature). After that, compute the complexity of method (CM) for each method m:

$$\mathrm{CM}(m) = (\mathrm{T} + \mathrm{S}) \times \mathrm{numLA} + (2\mathrm{T} + 3\mathrm{S}) \times \mathrm{numHA} + (3\mathrm{T} + 5\mathrm{S}) \times \mathrm{numIA}$$

Here, numLA, numHA, and numIA are respectively the number of light, heavy, and imported parameters in the signature of m. T and S represent trivial (T = 1, S = 0) and substantial (T = 0, S = 1). Third, compute the behavioral point (BP) for the class c as follows:

$$\mathrm{BP}(c) = [1 + \mathrm{numAss}(c)] \times \sum_{i=1}^{\mathrm{numMet}(c)} \mathrm{CM}(m_i)$$

where numAss(c) is the number of associations of c and numMet(c) is the number of methods in c.

(3) Compute the number of class point (CP) for the class c as follows:

$$\mathrm{CP}(c) = 2 \times \mathrm{SP}(c) + 3 \times \mathrm{BP}(c)$$

For the system corresponding to the class diagram D, its FS_CP metric is defined as:

$$\mathrm{FS\_CP} = \sum_{c \in D} \mathrm{CP}(c)$$

The FS_CP metric for the system is the sum of the class points over all classes in the class diagram. Consequently, we may use FS_CP to build a SLOC prediction model at the early development phase.

2.5. Objective class points

The O_CP (objective class points) metric counts the number of class points in a class diagram [6]. Unlike other class points [2,4,10,17,20,33,57], the computation of O_CP does not involve any subjective factors. For a given class diagram, the O_CP metric of the corresponding system is defined as [6]:

$$\mathrm{O\_CP} = \mathrm{NC} + \mathrm{NGen} + \mathrm{NDep} + \mathrm{NORR} + \mathrm{NM} + \mathrm{NA} + \mathrm{NAssoc} + \mathrm{NAgg} + \mathrm{NComp}$$

Here, NC, NM, and NA are respectively the total numbers of classes, methods, and attributes. NGen, NDep, NORR, NAssoc, NAgg, and NComp are respectively the total numbers of generalization relationships, dependency relationships, realization relationships, association relationships, aggregation relationships, and composition relationships. Since the computation of O_CP only uses the information from the class diagram, we may use O_CP to build a SLOC prediction model at the early development phase.

2.6. Object-oriented function points

The OOFP (object-oriented function points) metric is an adaption of traditional function points (FP) to measure the size of an object-oriented system using the information available from its class diagram [10,32]. In the traditional FP approach, the core concepts in counting function points are logical files (LF) and transactions that operate on those files [21]. The former are classified as internal logical files (ILFs) and external interface files (EIFs) and the latter are classified as inputs, outputs, and inquiries. Here, the ILFs denote the logical files maintained by the application and the EIFs denote the logical files referenced by the application but maintained by other applications. In OOFP, logical files are mapped to classes and transactions are mapped to their methods [10]. More specifically, the classes within and outside the application boundary are respectively treated as ILFs and EIFs and their methods are simply treated as generic Service Requests (SRs). Note that the methods in the classes are not distinguished, as they in general do not contain the necessary information to tell whether they perform inputs, outputs, or inquiries.

Based on this rationale, Antoniol et al. use the following formula to count the OOFP metric [10]:

$$\mathrm{OOFP} = \mathrm{OOFP}_{\mathrm{ILF}} + \mathrm{OOFP}_{\mathrm{EIF}} + \mathrm{OOFP}_{\mathrm{SR}}$$

where

$$\mathrm{OOFP}_{\mathrm{ILF}} = \sum_{c \in A} W_{\mathrm{ILF}}(\mathrm{DET}_c, \mathrm{RET}_c)$$

$$\mathrm{OOFP}_{\mathrm{EIF}} = \sum_{c \notin A} W_{\mathrm{EIF}}(\mathrm{DET}_c, \mathrm{RET}_c)$$

$$\mathrm{OOFP}_{\mathrm{SR}} = \sum_{c \notin A} W_{\mathrm{SR}}(\mathrm{DET}_c, \mathrm{FTR}_c)$$

In the above formulae, A denotes the set of classes belonging to the application. DET_c, RET_c, and FTR_c are respectively the numbers of DETs (data element types), RETs (record element types), and FTRs (file types referenced) in the class c. W_{ILF}, W_{EIF}, and W_{SR} are respectively the complexity matrices for ILFs, EIFs, and SRs (shown in Table 4), which are given in the IFPUG counting practices manual release 4.0 [10,47].

For a given class diagram, the computation of the OOFP metric consists of the following steps:

(1) *Identify logical files*: There are four strategies to identify which classes are to be considered as logical files.
   - *Simple strategy*: Each separated class is considered as a single logical file.
   - *Aggregation strategy*: Each entire aggregation structure is considered as a single logical file.
   - *Generalization/specialization strategy*: For a given inheritance hierarchy, the collection of classes comprised in the entire path from the root class to each leaf class is considered as a different logical file.
   - *Mixed strategy*: Combination of the aggregation strategy and the generalization/specialization strategy.

(2) *Determine complexity*. This step determines the complexity of each logical file and each service request. For each logical file, the numbers of DETs and RETs are first counted using the following rules.
   - Each attribute having a simple type (such as integer and string) is counted as a DET.
   - Each single-valued association or aggregation is counted as a DET.
   - Each logical file itself is counted as a RET.
   - Each attribute having a complex type (such as a class or a reference to a class) is counted as a RET.
   - Each multiple-valued association or aggregation is counted as a RET.

Then, Table 4(a) is used to classify the complexity of this logical file as having a "low", "average", or "high" complexity, depending on the number of DETs and RETs. For each service request (i.e. a concrete method) of a class, the number of DETs and FTRs are first counted by examining its signature.

- Each parameter having a simple type (such as integer and string) is counted as a DET.
- Each parameter having a complex type (such as a class or a reference to a class) is counted as an FTR.

Then, Table 4(b) is used to classify the complexity of this service request as having a "low", "average", or "high" complexity, depending on the number of DETs and FTRs.

(1) Allocate OOFP values. This step assigns a complexity weight, i.e. an OOFP value, to each logical file/service request according to Table 4(c).
(2) Sum OOFP values. The individual OOFP values are summed to obtain the total OOFP value of the system.

In [10], Antoniol et al. investigate the relationships between the OOFP metric of a system and its source code size in lines of code (LOC). Their experimental results show that the estimation accuracy of LOC is not significantly influenced by the strategy for identifying logical files. In other words, there is no reason to prefer any of the four strategies for identifying logical files over another [10]. For the simplicity of data collection, in this paper, we hence choose the simple strategy to identify logical files when computing the OOFP metric based on the UML class diagram. After that, we use the resulting OOFP metric to build a SLOC prediction model.

Table 4
Complexity matrices for OOFP.

Number of RETs	Number of DETs		
	1–19	20–50	51+
(a) ILF/EIF complexity matrix			
1	Low	Low	Average
2–5	Low	Average	High
6+	Average	High	High
Number of FTRs	Number of DETs		
	1–5	6–19	20+
(b) SR complexity matrix			
0–1	Low	Low	Average
2–3	Low	Average	High
4+	Average	High	High
	ILF	EIF	SR
(c) Complexity weights			
Low	7	5	3
Average	10	7	4
High	15	10	6

3. Related work

The earliest research estimating source code size for object-oriented systems using the information from class diagrams appears to be the work of Mišić and Tešić [7]. They used OLS regression to predict the system size in SLOC based on the number of classes and the number of methods from the class model. In their experiment, the subject systems were seven C++ systems. They found that the OLS model based on the number of classes had a R^2 of 0.91 and the OLS model based on the number of methods had a R^2 of 0.73. As such, they concluded that the final source code size

in SLOC of a C++ system could be estimated by simple metrics from its class model constructed at the design phase.

Antoniol et al. conducted a pilot study to investigate the relationships between OOFP of a system and its final code size in LOC (the number of non-blank lines, including comments) [10]. In their study, they analyzed the effect of four different strategies for identifying logical files. In addition to OLS, they used three other modeling techniques: the least absolute deviation regression (L1) and two variants of robust regression (rregs and rlms). In particular, they employed NMSE (normalized mean squared error) and NMAE (normalized mean absolute error) from the leave-one-out (LOO) cross validation as the criteria to evaluate the accuracy of a prediction model. Based on eight industrial C++ systems, Antoniol et al. provided the following three findings. First, different counting strategies for logical files did not have a significant impact. This indicates that there is no reason to prefer any of the four strategies over another in practice. Second, L1 models had a lower accuracy and robust models did not gain much in explaining the data when compared to OLS models. This suggests that it is unnecessary to use more complex modeling techniques when using OOFP of a system to predict its final code size. Third, OOFP was much better than CDM when predicting LOC of a system, although it was unknown whether their difference was statistically significant.

Later, Antoniol et al. used a larger data set (29 C++ systems) to re-examine the ability of the OOFP approach to predict the final code size of a system [11]. In their experiment, they first identified a number of factors that influenced size estimation, including the use of components off the shelf (COTS), the use of library code, and automatically generated code. Then, after controlling for these factors, they obtained a refined data set consisting of 22 C++ systems. Based on this refined data set, they found that the model using CDM metrics suite was significantly better than the model using OOFP when predicting the final source code size of a system. In particular, the model using the raw counts of OOFP elements (ILF DET, ILF RET, and SR DET) or the number of DETs (ILF DET and SR DET) also outperformed the model using OOFP. Note that the OOFP value was collected using the generalization/specialization strategy. These results raised a question about the value of OOFP. Antoniol et al. hence stated that further research needed to be conducted to investigate the usefulness of the OOFP approach in practice.

Bianco et al. also empirically compared the ability of the OOFP approach and the CDM approach to predict the final source code of object-oriented systems [12]. Their data set consisted of 12 Java systems developed by students. In their experiment, they used the single class strategy to identify logical files during the computation of OOFP. They found that the model using OOFP was not better than the model using the number of methods and number of attributes. In [16], Chen et al. used linear, logarithmic, exponential, and polynomial equations to investigate the relationship between the number of classes and the final code size. Consequently, they found that the number of classes had a moderate correlation to the final code size and hence could be a useful size predictor.

Carbone et al. applied the FS_CP approach to estimate the final code size of a Java system [4]. In their experiment, they first computed the number of FS_CP for each class in the class diagram. Then, the number of FS_CP for each class was adjusted using the information from the use case diagram. Finally, the adjusted number of FS_CP for each class was used to estimate the final code size, which achieved a high accuracy (MMRE = 0.095). However, this result is based on only one system and no successive empirical studies were conducted to investigate whether it can be generalized to other systems.

Tan et al. investigated the ability of the CDM approach to predict the final code size of data intensive systems [18,25,69,70]. In [18], Tan et al. believed that the class diagram for a data intensive system can be viewed as an extended entity-relationship (ER) model. For most data intensive projects, they observed that the relationship types in class diagrams were not classified into associations, aggregations, and generalizations. In this context, they used the following three metrics to predict the code size of a system: the number of classes (i.e. entity types), number of attributes, and number of relationships. In their experiment, Tan et al. evaluated the accuracy of two prediction models. The first model was built on one data set from 13 VB systems but was validated on a different data set from 8 VB systems (i.e. hold-out validation method). The second model was built and validated on the same data set from 10 Java systems. Their results showed that the CDM approach was able to predict the final code size of data intensive systems. In [25], Tan et al. extended their previous work to more data sets. In this work, they used the number of classes, the number of relationships, and the average number of attributes in each class as the independent variables. In particular, in addition to the hold-out method, they also used the 10-fold cross-validation to select the best fitted parameters to build and validate the model based on the whole data set. Their overall results showed that the CDM approach had a good ability to predict the final code size of data intensive systems.

Overall, in the last two decades, only few empirical studies were done to investigate the accuracy of early size estimation approaches. In particular, there are three main limitations in these studies. First, the accuracies of most early size estimation approaches are not compared. Consequently, for practitioners, it is unknown which sizing approaches are better when applied in practice. Second, previous studies often use 1–30 systems to investigate the accuracy of early size estimation approaches, which are not large enough to draw statistically meaningful conclusions. Third, for a system, almost all studies assume a linear relationship between the metrics collected from its UML class diagram and its final source code size. However, it is unclear whether other modeling techniques are more appropriate for source code size prediction. From the above discussion, we can see that there is a strong need to use a large number of systems and modeling techniques to compare the actual accuracy of different early size estimation approaches.

4. Research method

In this section, we describe the data set used for this study, the modeling techniques that we will investigate, the model evaluation criteria, and the data analysis procedure.

4.1. Data set

This study makes use of the data collected from 100 open-source Java systems. We downloaded these systems from http://sourceforge.net/ and http://freshmeat.net, two well-established open-source software websites. These investigated systems cover a variety of application domains including software development, internet, science/engineering, games/entertainment, communications, office/business, system, multimedia, and database. Table 14 in Appendix A summarizes the system name, the version number, the total number of classes, and the source code size in SLOC, and the project website for these systems.

For each Java system, we used the Perl API provided by a reverse engineering tool called Understand for Java[1] to analyze its source code to extract the relevant information of the corresponding class diagram. In particular, the following relationships among classes are recovered: association, aggregation, composition, generalization,

[1] http://www.scitools.com.

Table 5
Modeling techniques employed in this study.

Technique	Learner	Description
Linear models	Ordinary least squares regression (OLS)	A linear function minimizing the sum of squared errors
	Least median squared regression (LMS)	A linear function minimizing the median of squared errors
Non-linear models	Support vector machine regression (SVM)	A linear function in a higher dimensional space mapped from the input space
	Multilayer perceptron neural networks (MLP)	A feedforward artificial neural network with back propagation training
Rule/tree-based models	Decision table (DT)	A simple decision table majority learner
	Fast decision tree (REPTree)	A fast decision tree learner based on information gain and reduced-error pruning
Instance-based models	K-nearest neighbours (IBk)	A nearest-neighbor learner that uses the Euclidean distance function
	KStar (K*)	A nearest-neighbor learner that uses the entropy-based distance function

realization, and dependency (see Appendix B for detail). Then, we collected those metrics described in Section 2 from the class diagram. Consequently, we obtained a data set having 100 data points. Each data point in the data set corresponds to one Java system and consists of: (1) 10 class diagram metrics (i.e. NC, NA, NM, NAssoc, NAgg, NComp, NDep, NGen, NGenH, and MaxDIT); (2) 5 function points metrics (i.e. POPs, Oops, FS_CP, O_CP, and OOFP); and (3) the total source code size in SLOC (i.e. non-blank, non-commentary source lines) for that system. The *dependent variable* used in this study is the total source size in SLOC. The *independent variables* consist of the above-mentioned 10 class diagram metrics and 5 function point metrics.

4.2. Modeling techniques

We want to investigate which combinations of source code size estimation approaches and modeling techniques give the best estimate of source code size. To this end, we consider eight typical modeling techniques which can be grouped into the categories of linear models, non-linear models, rule/tree-based models, and instance-based models. Table 5 summarizes these modeling techniques, including their names and descriptions. They are available in the Weka (Waikato Environment for Knowledge Analysis) open-source tool.[2] We select these techniques because they are either established techniques in size estimation studies or techniques with promising results in other regression contexts. In the literature of size estimation, OLS is the most commonly used modeling technique due to its simplicity and good performance. As an alternative to OLS, LMS has the advantage of being insensitive to the existence of outliers in the data set. To the best of our knowledge, LMS has not been applied to size estimation in previous literature. MLP and SVM are two popular non-linear modeling techniques which have been successfully applied in various domains but their suitability to size estimation has not been studied. The prediction models generated by DT or REPTree are simple to understand and interpret. It would be valuable to know whether they can be applied to size estimation. IBk and K* are two modeling techniques that do not explicitly induce a model. They search for the most similar instances in the data and then use them to produce the estimation value. Although they are commonly used in software effort estimation, their effectiveness in size estimation has not been examined.

In our data set, both the independent and dependent variables exhibit a highly skewed distribution (see the kurtosis and skewness values in Table 6 in Section 5.1). As stated in [38], by applying a natural logarithmic transformation to the data, such skewed distributions could be brought closer to a normal distribution. This might lead to an improvement in the prediction accuracy not only for linear regression models, but also for machine learning models [38,48,49,51]. To test the value of the logarithmic transformation, we build the following two models for each modeling technique: the model without a logarithmic transformation and the model with a logarithmic transformation. During the logarithmic transformation, we transform a variable x into $ln(x)$. To avoid numerical errors with $ln(0)$, all numeric values under 0.000001 are replaced with $ln(0.000001)$. This is the method followed in [48]. With six types of data (i.e. CDM, POPs, Oops, FS_CP, O_CP, and OOFP), eight modeling techniques, and two transformations (i.e. with/without a logarithmic transformation), we obtain a total number of 96 (i.e. $6 \times 8 \times 2$) size prediction models. By statistical testing, we can determine which models have the significantly highest accuracy. This will provide valuable information to guide the development and application of size estimation models in practice.

[2] http://www.cs.waikato.ac.nz/ml/weka/.

4.3. Model evaluation criteria

For any source code size prediction model, it is important to know the degree to which the size that the model predicted matches the actual size. In this study, we use the following three performance measures to quantify the accuracy of a prediction model: MAR (mean of absolute residuals), MMRE (mean magnitude of relative error), and Pred(q) (prediction capability at level q). All of these accuracy measures are based on only two terms, the actual and the predicted values.

Suppose the data set consists of n data points (in this study, one data point corresponds to one system). Given a data point i, let y_i be its actual source code size and $\hat{y}_i$ the predicted source code size from a prediction model, where $1 \leqslant i \leqslant n$. Then, the AR (absolute residuals) and MRE (magnitude of relative error) for the data point i are respectively:

$$\mathrm{AR}_i = |y_i - \hat{y}_i|$$

$$\mathrm{MRE}_i = \frac{|y_i - \hat{y}_i|}{y_i}$$

Based on AR_i, we can define the following accuracy measure MAR for the prediction model:

$$\mathrm{MAR} = \frac{1}{n}\sum_{i=1}^{n} AR_i$$

Based on MRE_i, we can define the following two accuracy measures for the prediction model:

$$\mathrm{MMRE} = \frac{1}{n}\sum_{i=1}^{n} MRE_i$$

$$\mathrm{Pred}(q) = \frac{1}{n}\sum_{i=1}^{n} \begin{cases} 1 & \text{if } \mathrm{MRE}_i \leqslant \frac{q}{100} \\ 0 & \text{otherwise} \end{cases}$$

MAR is the mean of all ARs and MMRE is the mean of all MREs. Pred(q) is a measure of the proportion of the predicted values that have an *MRE* value less than or equal to a specified value of $q/100$, i.e. it is the percentage relative error deviation within q. In this study, $q = 25$ is used because it is commonly used in the literature

Table 6
Descriptive statistics for all metrics.

Metric	Max.	75%	Median	25%	Min.	Mean	Std. dev.	Kurtosis	Skewness	NNZ > 5?
NC	1364	199.750	108	52.250	18	166.840	192.090	19.015	3.776	Yes
NA	5708	946.750	491	313.250	57	763.590	776.545	16.734	3.292	Yes
NM	9828	2035.750	1029.500	499.750	119	1541.200	1585.313	8.616	2.525	Yes
NAssoc	46	7	4	1	0	6.260	7.942	8.551	2.613	Yes
NAgg	89	8	4	1	0	7.740	14.104	16.513	3.797	Yes
NComp	1122	76.500	40	18.250	3	71.920	126.592	48.711	6.243	Yes
NDep	999	216.750	74.5	32	4	153.450	194.775	6.270	2.350	Yes
NGen	954	126	59.5	25	0	100.680	137.908	19.650	3.930	Yes
NGenH	49	11	6	2.250	0	8.600	9.188	4.807	2.072	Yes
MaxDIT	9	6	6	4	1	5.4300	1.465	0.200	-0.469	Yes
POPs	255367.613	37192.239	12994.708	5552.288	901.225	27280.966	39551.378	14.871	3.461	Yes
FS_CP	177815724	2736613	665531.500	187380.500	12518	4475986.430	1.856	78.725	8.521	Yes
O_CP	20411	3897.500	1955	1014	321	2874.980	2925.989	12.947	2.925	Yes
OOPS	36634	7290	3682	1879.500	600	5515.500	5482.979	10.627	2.666	Yes
OOFP	39335	7640	3933.500	1902.500	502	5784.720	5940.007	10.541	2.709	Yes
SLOC	131262	25752	10883.500	5763	1230	18481.860	20173.831	11.341	2.824	Yes

[38]. In [38], Dejaeger et al. state that the Pred(25) typically attains values between 0.1 and 0.6.

Note that there are a lot of debates on whether MMRE and Pred(25) should be used as model evaluation criteria. In [71], Kitchenham et al. show that MMRE and Pred(25) can be interpreted as the spread and kurtosis of the distribution of the variable $z_i = \hat{y}_i/y_i$, respectively. In [72], Foss et al. show that MMRE is an unreliable criterion to select competing prediction models, as it will select the worst model in most cases. In [73], Port et al. replicate Foss's experiment and extend it to Pred(25). Their results confirm that MMRE is an unreliable selection criterion. However, they do find that Pred(25) is a reliable selection criterion. In this paper, in addition to MAR, we also include MMRE and Pred(25) as the model evaluation criteria. The reasons are twofold. First, despite criticism, MMRE and Pred(25) remain as the most widely used model evaluation criteria in software engineering literature [38,74]. Second, using MMRE and Pred(25) as accuracy measures allows us to directly compare our results to the results reported in previous studies.

4.4. Data analysis procedure

At a high level, our analysis will proceed as follows:

(1) We first consider the descriptive statistics of the CDM, POPs, Oops, FS_CP, O_CP, and OOFP metrics in the data set. The distribution of each metric is examined to select those with enough variance for further analysis. Metrics with low variance do not differentiate systems very well and therefore are not useful size predictors. As a general rule of thumb, only metrics with more than five non-zero data points were considered for all subsequent analyses [62].
(2) We then use the Pearson correlation coefficients to analyze the relationships between each of the investigated metrics and source code size of the systems. The purpose of this step is to identify metrics that are significantly related to source code size. This will give us a preliminary indication of which types of metrics are possibly useful predictors for source code size.
(3) We use eight modeling techniques with/without a logarithmic transformation to build size prediction models. Consequently, for the CDM, POPs, Oops, FS_CP, O_CP, and OOFP metrics, we have a total number of 96 prediction models. Since CDM consists of a number of class diagram metrics, we first use CfsSubsetEval and GreedyStepwise search method [68] to select a subset of highly predictive metrics. Then, we use those selected metrics to build the prediction models. For each of the other five function points metrics, variable selection is not necessary as it consists of only one single metric. In our study, we employ Weka with default settings to build the prediction models.
(4) We apply 10 times 10-fold cross-validation to compare the prediction performances of these 96 models. More specifically, at each 10-fold cross-validation (CV), the order of the data points in the data set is first randomized. Then, the data set is divided into 10 parts of approximately equal size. Each part is used to compute the accuracy measure (e.g. MMRE, MdMRE, or Pred(25)) for the prediction models fitted using the remainder of the data set. Consequently, each prediction model has $10 \times 10 = 100$ accuracy measure values. Based on these 100 values, we use the Wilcoxon's signed-rank test to examine whether two prediction models have significantly different accuracy values at the significance level of 0.05. The Wilcoxon's signed-rank test is a non-parametric statistical method to test the null hypothesis that the population median of the paired differences of two samples is zero. If the Wilcoxon's signed-rank test reports a p-value lower than 0.05, then the two prediction models are considered to have a significantly different prediction accuracy and otherwise not. Note that, since multiple tests on the same data set may result in spurious statistically significant results, we used Benjamini–Hochberg correction of p-values to control for false discovery [76]. Furthermore, we use Cohen's d to examine whether the difference between the prediction accuracy of two models are important from the viewpoint of practical application. Numerically, the Cohen's d is defined as the difference between two means divided by the standard deviation for the data [75]. For the Cohen's d, a value lower than 0.2 indicates a trivial difference, between 0.2 and 0.5 indicates a small difference, between 0.5 and 0.8 indicates a moderate difference, and higher than 0.8 indicates a large difference [75].

5. Experimental results

In this section, we present in detail the experimental results. In Section 5.1, we report the descriptive statistics of the investigated metrics. In Section 5.2, we show the Pearson correlations of the investigated metrics to source code size. In Section 5.3, we

compare the accuracy of six types of SLOC estimation approaches in combination with/without a logarithmic transformation under eight modeling techniques. Section 5.4 compares our work to previous work. Section 5.5 discusses the threats to the validity of this study.

5.1. Descriptive statistics

In Table 6, columns "Max.", "75%", "Median", "25%", "Min.", "Mean", and "Std. dev." state for each metric the maximum value, upper quartile, median value, lower quartile, minimum value, mean value, and standard deviation. Columns "Skewness" and "Kurtosis" represent the skewness and kurtosis values of the metric. The skewness value characterizes the degree of asymmetry of the distribution around its mean and the kurtosis value characterizes the relative peakedness or flatness of the distribution compared to the normal distribution. Normal distributions produce a skewness statistic of zero and a kurtosis statistic of zero [35]. In practice, the distribution of a variable can be considered normal if its skewness and kurtosis statistics fall between −1.0 and +1.0 [34]. Column "NNZ > 5" indicates whether enough non-zero data points are present to allow further analysis with a specific metric: "Yes" means the metric has more than five non-zero data points and "No" means the metric has not.

From Table 6, we can see that, for all metrics except MaxDIT, there are large differences between the lower 25th percentile, the median, and the 75th percentile, thus showing strong variations across systems. This suggests that MaxDIT might not be a useful size predictor. In addition, as indicated by the values of skewness and kurtosis, the distributions of all metrics except MaxDIT deviate significantly from normal distributions. Since all the metrics have more than five non-zero values, they will be used for further analysis.

5.2. Pearson correlation to software system size

Table 7 shows the results of the Pearson correlation analysis of the investigated metrics to software system size. For each metric: (1) the second and third columns show the Pearson correlation coefficient and its statistical significance when not controlling the total number of classes; and (2) the fourth and fifth columns show the Pearson correlation coefficient and its statistical significance when controlling the total number of classes. Assuming a reasonably sized data set, Hopkins calls a correlation value less than 0.1 trivial, 0.1–0.3 minor, 0.3–0.5 moderate, 0.5–0.7 large, 0.7–0.9 very large, and 0.9–1.0 almost perfect [31].

As can be seen, when not controlling for the total number of classes, all metrics except MaxDIT exhibit a positive correlation ranging from large to almost perfect. Of these metrics, NA, NM, O_CP, OOPS, and OOFP have an almost perfect correlation. The implications of these results are twofold. First, this again suggests that MaxDIT might not be a useful size predictor. Second, other metrics especially NA, NM, O_CP, OOPS, and OOFP are good predictors for software size.

For most metrics, when controlling for the total number of classes, the degree of their correlations with software size dramatically decreases. The reason is that the total number of classes has a confounding effect on the associations between these metrics and software system size [65,66]. In particular, we have the following two findings: (1) NGen becomes negatively significant; and (2) NGenH and MaxDIT are insignificant. These findings are not unique for our data set. In [10], Antontiol et al. used 8 systems to investigate the ability of the OOFP and CDM metrics suite to predict the system size. In their data set, when controlling for the total number of classes, the Pearson correlation of NGen to LOC is also significant (p-value = 0.024). In other words, the larger the number of generalization relationships, the smaller is the system size. However, the number of structures with generalization relationships and the maximum depth of inheritance hierarchy do not have an influence on the system size.

5.3. Model evaluation

This section reports the accuracy of six types of metrics in combination with/without a logarithmic transformation under eight modeling techniques. Section 5.3.1 analyzes which metrics produces the highest prediction accuracy under each specific modeling technique. Section 5.3.2 investigates the influence of the logarithmic transformation on the prediction accuracy of size prediction models. Section 5.3.3 identifies the optimal combination of input metric, modeling technique, and data transformation method for the prediction of the system size.

5.3.1. Which metrics produces the highest accuracy for source code size prediction?

Tables 8–10 respectively summarize the mean of MAR, MMRE, and Pred(25) for different prediction models obtained from 10 times 10-fold CV. Under each specific modeling technique: (1) the best performing size prediction model is shown in bold and underlined; (2) those not significantly different from the best model are shown in bold and italic; and (3) the Cohen's *ds* between the best and the current models are shown in parentheses. Furthermore, for those best models identified under each specific modeling technique: (1) the globally best performing size prediction model is shown in a deep gray background; (2) those not significantly different from the globally best model are shown in a light gray background; and (3) the Cohen's *ds* relative to the globally best model, which are trivial (i.e. $|d| < 0.20$), are shown in a rectangle. Note that, we use the Benjamini–Hochberg adjusted p-values to determine whether two models are statistically different with respect to MAR, MMRE, and Pred(25).

Tables 8(a), 9(a), and 10(a) show the results without a logarithmic transformation. According to MAR, OOPS is the best metric under all modeling techniques except K*. Under K*, O_CP is the best metric but the Cohen's *d* shows that the difference between OOPS and O_CP is trivial. According to MMRE, OOPS significantly outperforms the other metrics under all modeling techniques. However, when K* is used, OOPS does not significantly outperform O_CP. According to Pred(25), when DT is used, CDM is the best metric. Under the other modeling techniques, OOPS is the best metric,

Table 7
Pearson correlation to software size.

Metric	Not controlling for the total number of classes		Controlling for the total number of classes	
	Pearson's r	P-value	Pearson's r	P-value
NC	0.781	<0.001		
NA	0.918	<0.001	0.793	<0.001
NM	0.941	<0.001	0.844	<0.001
NAssoc	0.562	<0.001	0.469	<0.001
NAgg	0.734	<0.001	0.494	<0.001
NComp	0.835	<0.001	0.604	<0.001
NDep	0.799	<0.001	0.559	<0.001
NGen	0.680	<0.001	−0.408	<0.001
NGenH	0.613	<0.001	0.178	0.075
MaxDIT	0.314	<0.001	0.144	0.151
POPs	0.816	<0.001	0.474	<0.001
FS_CP	0.758	<0.001	0.440	<0.001
O_CP	0.953	<0.001	0.899	<0.001
OOPS	0.967	<0.001	0.928	<0.001
OOFP	0.932	<0.001	0.852	<0.001

Table 8
MAR performance from 10 times 10-fold CV.

Model	OLS	LMS	SVM	MLP	DT	REPTree	IBk	K*
(a) Without log transformation								
CDM	3947(0.447)	3794(0.211)	4443(0.496)	6351(0.452)	7545(0.398)	7388(0.315)	5131(0.288)	4508(0.248)
POPs	7874(2.252)	7137(1.622)	7138(1.542)	9249(1.247)	8813(0.822)	9924(0.903)	9134(1.646)	8610(1.645)
FS_CP	12678(1.751)	12615(1.431)	24357(0.771)	8961(1.403)	11978(1.972)	7105(0.253)	7783(1.396)	14531(2.378)
O_CP	3803(0.359)	3741(0.175)	3815(0.189)	5985(0.412)	6640(0.151)	6390(0.070)	5046(0.297)	4000(0.000)
OOPS	3305(0.000)	3483(0.000)	3538(0.000)	4909(0.000)	6267(0.000)	6112(0.000)	4439(0.000)	4167(0.084)
OOFP	4704(0.901)	4856(0.838)	4819(0.756)	6863(0.668)	7265(0.362)	7643(0.375)	6881(1.026)	6236(1.004)
(b) With log transformation								
CDM	3545(0.203)	3556(0.132)	3677(0.310)	4754(0.156)	5584(0.370)	5799(0.160)	5202(0.284)	4543(0.000)
POPs	6523(1.539)	6431(1.493)	6572(1.635)	7409(1.107)	7841(1.010)	8122(0.839)	9545(1.715)	7332(0.807)
FS_CP	4878(0.987)	4976(0.945)	5106(1.005)	6004(0.576)	6041(0.547)	6599(0.431)	7927(1.386)	6182(0.483)
O_CP	3736(0.337)	3709(0.234)	3846(0.443)	5025(0.249)	5203(0.240)	5550(0.079)	5048(0.255)	5300(0.222)
OOPS	3260(0.000)	3368(0.000)	3243(0.000)	4375(0.000)	4554(0.000)	5311(0.000)	4516(0.000)	5012(0.138)
OOFP	4638(0.882)	4821(0.888)	4571(0.890)	5994(0.585)	6164(0.524)	6533(0.375)	6880(0.977)	6012(0.431)

Table 9
MMRE performance from 10 times 10-fold CV.

Model	OLS	LMS	SVM	MLP	DT	REPTree	IBk	K*
(a) Without log transformation								
CDM	0.232(0.385)	0.218(0.317)	0.227(0.410)	0.511(0.207)	0.630(0.469)	0.556(0.324)	0.287(0.479)	0.246(0.355)
POPs	0.764(3.038)	0.489(2.640)	0.438(2.125)	0.843(0.814)	0.835(1.167)	0.884(0.793)	0.535(2.167)	0.502(2.262)
FS_CP	1.405(3.040)	0.944(2.960)	0.681(2.151)	0.919(0.967)	1.477(2.163)	0.506(0.270)	0.497(2.630)	0.664(4.489)
O_CP	0.234(0.403)	0.223(0.400)	0.226(0.383)	0.539(0.280)	0.592(0.310)	0.409(0.074)	0.312(0.825)	0.234(0.148)
OOPS	0.211(0.000)	0.203(0.000)	0.206(0.000)	0.438(0.000)	0.527(0.000)	0.375(0.000)	0.254(0.000)	0.225(0.000)
OOFP	0.308(1.589)	0.282(1.445)	0.287(1.443)	0.627(0.470)	0.593(0.333)	0.512(0.276)	0.411(1.689)	0.345(1.596)
(b) With log transformation								
CDM	0.205(0.313)	0.203(0.132)	0.206(0.318)	0.248(0.29)	0.309(1.262)	0.285(0.415)	0.293(0.51)	0.217(0.000)
POPs	0.382(2.070)	0.388(2.055)	0.406(2.043)	0.449(1.576)	0.419(1.978)	0.464(1.837)	0.543(2.211)	0.425(2.117)
FS_CP	0.314(1.827)	0.325(1.858)	0.317(1.842)	0.365(1.327)	0.356(1.625)	0.386(1.602)	0.500(2.599)	0.354(1.746)
O_CP	0.220(0.569)	0.221(0.477)	0.225(0.655)	0.266(0.494)	0.268(0.710)	0.275(0.306)	0.312(0.789)	0.258(0.642)
OOPS	0.191(0.000)	0.197(0.000)	0.192(0.000)	0.228(0.000)	0.224(0.000)	0.255(0.000)	0.256(0.000)	0.232(0.251)
OOFP	0.281(1.656)	0.281(1.546)	0.278(1.599)	0.334(1.137)	0.326(1.361)	0.332(0.931)	0.410(1.653)	0.313(1.449)

Table 10
Pred(25) performance from 10 times 10-fold CV.

Model	OLS	LMS	SVM	MLP	DT	REPTree	IBk	K*
(a) Without log transformation								
CDM	0.629(0.289)	0.680(0.122)	0.627(0.661)	0.423(0.171)	0.387(0.000)	0.491(0.251)	0.522(0.320)	0.624(0.187)
POPs	0.307(2.836)	0.362(2.464)	0.436(1.978)	0.315(0.787)	0.323(0.457)	0.311(1.315)	0.272(2.065)	0.290(2.619)
FS_CP	0.110(4.826)	0.228(3.528)	0.294(3.195)	0.247(1.197)	0.121(2.265)	0.381(0.938)	0.255(2.267)	0.155(3.931)
O_CP	0.658(0.081)	0.657(0.284)	0.635(0.662)	0.410(0.240)	0.356(0.230)	0.503(0.192)	0.494(0.507)	0.649(0.021)
OOPS	0.669(0.000)	0.696(0.000)	0.727(0.000)	0.458(0.000)	0.380(0.052)	0.537(0.000)	0.572(0.000)	0.652(0.000)
OOFP	0.474(1.364)	0.539(1.164)	0.510(1.455)	0.372(0.457)	0.360(0.197)	0.444(0.517)	0.419(1.021)	0.440(1.494)
(b) With log transformation								
CDM	0.707(0.237)	0.710(0.080)	0.696(0.295)	0.604(0.176)	0.489(1.038)	0.564(0.042)	0.532(0.233)	0.660(0.000)
POPs	0.453(2.069)	0.456(1.848)	0.495(1.804)	0.412(1.270)	0.419(1.497)	0.374(1.336)	0.262(2.155)	0.409(1.704)
FS_CP	0.490(1.793)	0.493(1.546)	0.485(1.769)	0.447(1.084)	0.449(1.274)	0.398(1.215)	0.253(2.288)	0.425(1.594)
O_CP	0.683(0.432)	0.675(0.331)	0.660(0.583)	0.579(0.329)	0.567(0.531)	0.554(0.111)	0.494(0.498)	0.578(0.569)
OOPS	0.740(0.000)	0.721(0.000)	0.737(0.000)	0.634(0.000)	0.642(0.000)	0.570(0.000)	0.570(0.000)	0.634(0.176)
OOFP	0.542(1.425)	0.535(1.310)	0.554(1.314)	0.471(0.973)	0.468(1.125)	0.489(0.543)	0.420(1.008)	0.467(1.325)

although it does not significantly outperform all the other input metrics. Overall, these results suggest that, OOPS is the best metric if we do not apply a logarithmic transformation to pre-process the data.

Tables 8(b), 9(b), and 10(b) show the results with a logarithmic transformation. According to MAR, MMRE and Pred(25), under all the modeling techniques except K*, OOPS is the best metric. We also observe that CDM and O_CP have the similar performance to OOPS under many modeling techniques. Overall, these results indicate that OOPS is still the best option after a logarithmic transformation is applied to the data.

5.3.2. Can logarithmic transformation improve the accuracy for source code size prediction?

Table 11(a–c) respectively summarize the results for comparing the MAR, MMRE, and Pred(25) of the prediction models with a logarithmic transformation and without a logarithmic transformation. Here, "+" means that the model with a logarithmic transformation has a significantly better accuracy, "−" means that the model with a logarithmic transformation has a significantly worse accuracy, "0" means that the accuracy of the model with a logarithmic transformation is not significantly different from that of the model without a logarithmic transformation. Note that we use the Benja-

Table 11
Comparison of the prediction models with and without a logarithmic transformation.

Model	OLS	LMS	SVM	MLP	DT	REPTree	IBk	K*
(a) MAR								
CDM	+(0.258)	+(0.156)	+(0.408)	+(0.498)	+(0.607)	+(0.419)	0(0.026)	0(0.013)
POPs	+(0.510)	+(0.259)	+(0.202)	+(0.471)	+(0.274)	+(0.428)	−(0.117)	+(0.351)
FS_CP	+(1.430)	+(1.182)	+(0.712)	+(0.909)	+(2.077)	0(0.140)	0(0.053)	+(1.684)
O_CP	+(0.044)	+(0.021)	0(0.020)	+(0.342)	+(0.595)	+(0.230)	−(0.001)	−(0.438)
OOPS	0(0.036)	+(0.084)	+(0.222)	+(0.223)	+(0.621)	+(0.238)	−(0.036)	−(0.283)
OOFP	+(0.037)	0(0.019)	+(0.136)	+(0.269)	+(0.358)	+(0.278)	0(0.001)	0(0.071)
(b) MMRE								
CDM	+(0.500)	+(0.322)	+(0.409)	+(0.932)	+(1.821)	+(0.593)	0(0.086)	+(0.482)
POPs	+(1.922)	+(0.743)	+(0.223)	+(0.843)	+(1.728)	+(0.746)	−(0.050)	+(0.521)
FS_CP	+(2.757)	+(2.417)	+(1.606)	+(1.210)	+(2.655)	+(0.327)	0(0.026)	+(2.742)
O_CP	+(0.228)	+(0.040)	0(0.017)	+(0.928)	+(1.989)	+(0.398)	−(0.003)	−(0.363)
OOPS	+(0.453)	+(0.146)	+(0.311)	+(0.960)	+(2.089)	+(0.363)	−(0.040)	0(0.119)
OOFP	+(0.392)	+(0.026)	+(0.141)	+(0.840)	+(1.757)	+(0.474)	0(0.005)	+(0.387)
(c) Pred(25)								
CDM	+(0.545)	+(0.223)	+(0.447)	+(0.937)	+(0.726)	+(0.440)	0(0.062)	+(0.239)
POPs	+(1.101)	+(0.652)	+(0.404)	+(0.564)	+(0.654)	+(0.404)	−(0.078)	+(0.847)
FS_CP	+(3.113)	+(1.819)	+(1.340)	+(1.222)	+(2.555)	0(0.117)	0(0.017)	+(2.091)
O_CP	+(0.187)	+(0.127)	+(0.184)	+(0.918)	+(1.580)	+(0.320)	0(0.000)	−(0.489)
OOPS	+(0.530)	+(0.186)	0(0.074)	+(0.954)	+(1.852)	0(0.202)	0(0.013)	0(0.122)
OOFP	+(0.461)	0(0.028)	+(0.288)	+(0.577)	+(0.722)	+(0.270)	0(0.007)	0(0.189)

mini–Hochberg adjusted *p*-values to determine whether the prediction models are significantly different. In particular, the Cohen's *ds* between the prediction models with and without a logarithmic transformation are shown in parentheses.

From Table 11, we can see that the logarithmic transformation can in general significantly improve the MAR, MMRE, and Pred(25) values for the linear models, non-linear models, and rule/tree-based models. This is especially true for the MLP and DT models (the corresponding Cohen's *d* values are larger than 0.5 in most cases). The reason is that a logarithmic transformation can make the data closer to a normal distribution. This confirms the previous finding that a logarithmic transformation can lead to an improvement in the prediction accuracy not only for linear regression models, but also for machine learning models [38,48,49,51]. However, it appears that applying the logarithmic transformation is not always valid for the instance-based models (such as IBk) and even may lead to a significantly worse prediction performance.

5.3.3. Which combination is optimal for source code size prediction?

From Section 5.3.1, we can see that OOPS is the overall best metric for size estimation, regardless of whether a logarithmic transformation is applied to pre-process the data. From Section 5.3.2, we can see that a logarithmic transformation can in general significantly improve the accuracy of the prediction models. This is especially true for OOPS. As such, OOPS + logarithmic transformation is the best option for the estimation of system size.

We next examine which technique is the most suitable modeling method when the OOPS metric with a logarithmic transformation is used. From Table 8(b), we can see that, the SVM model significantly outperform all the other models except the OLS model. In particular, according to the Cohen's *d*, the differences between the SVM model and the OLS model, between the SVM model and the LMS model are trivial. From Table 9(b), we can see that, according to MMRE, the OLS model is the best, although it does not significantly outperform the SVM model. From Table 10(b), we can see that the OLS model has a significantly higher Pred(25) value than all the other models except the SVM model. The overall observations show that OLS is the best modeling technique for the system size estimation. This finding is consistent with the studies of Tan et al. [18,25,69,70], who also found that OLS was a good modeling technique for size estimation. Consequently, we conclude that the model built with the combination of OOPS + logarithmic transformation + OLS is optimal for the system size prediction (mean MMRE = 0.191 and mean Pred(25) = 0.740 from 10 times 10-fold CV). Appendix C shows the detail of the OOPS based prediction models. Note that we use class diagrams recovered from source codes rather than real class diagrams constructed at the early phase of software development in this study. Therefore, the reported levels of accuracy might be optimistic.

5.4. Comparison to previous work

Table 12 compares our work to previous work. For each study, this table lists the year published, the metric used for size estimation, the source of class diagram, the characteristics of data sets (system type, development language, and number of systems), the validation method, the modeling techniques, and the prediction performance obtained. Here, OOFP_S denotes the OOFP metric collected using the single class strategy and OOFP_G denotes the OOFP metric collected using the generalization/specialization strategy. Note that, for Antoniol et al.'s study [10], we only list the OOFP result from the single class strategy (i.e. OOFP_S), as the results from the other strategies are similar. In particular, Antoniol et al.'s study [10] uses NMSE and NMAE to evaluate the prediction model. For ease of presentation, the cells shown in gray background under column MMRE and column MdMRE respectively denote NMSE and NMAE. For each type of approach in our study, we show the MMRE, MdMRE, and Pred(25) results obtained from the best performing modeling technique(s). From this table, we can see that: (1) previous studies focus on the validation of the CDM approach and the OOFP approach; (2) most studies are based on a small number of object-oriented systems; (3) compared to previous work, our study not only use a large number of object-oriented systems to investigates six types of size estimation approaches, but also employs more modeling techniques to analyze the estimation accuracy of these approaches.

5.5. Threats to validity

We consider the most important threats to the construct, internal, and external validity of our study. Construct validity is the degree to which the dependent and independent variables accurately measure the concept they purport to measure. Internal validity is the degree to which conclusions can be drawn about the causal effect of independent variables on the dependent variables. External validity is the degree to which the results of the research can be

Table 12
Comparison to previous work.

Study	Year	Metric	RCD?	Subject systems	Dev. language	#Data points	Validation type	Modeling techniques	MMRE	MdMRE	Pred(25)
Mišić and Tešić [7]	1998	CDM	No	Industrial	C++	7	LOO	OLS	0.21	0.20	0.57
Antoniol et al. [10]	1999	OOFP_S	No	Industrial	C++	8	LOO	OLS	0.39	0.66	
								L1	0.55	0.81	
								RREG	0.40	0.67	
								RLM	0.40	0.67	
		CDM						OLS	2.14	1.25	
Carbone et al. [4]	2002	FS_CP	No	Student	Java	1	N/A	N/A	0.095		
Antoniol et al. [11]	2003	OOFP_G	No	Industrial	C++	29	LOO	OLS	0.54		0.59
		Elements							0.27		0.68
		DETs							0.25		0.73
		CDM							0.45		0.64
Chen et al. [16]	2004	CDM	No	Student	Various	14	Fit	OLS, Exp, Poly.	$0.28 \leqslant R^2 \leqslant 0.53$		
Bianco and Lavazza [12]	2005	OOFP_S	Yes	Student	Java	12	Fit	OLS	0.13	0.08	0.92
		CDM							0.13	0.09	0.75
Tan and Zhao [18]	2006	CDM	N/A	Industrial	VB	21	Hold-out	OLS	0.22		0.65
					Java	10	Fit		0.03		1.00
Tan et al. [25]	2009	CDM	No	Industrial	VB	32	Hold-out	OLS	0.18		0.81
			Yes	OS	PHP	63			0.14		0.81
			No	Industrial	Java	32			0.15		0.81
			Yes	OS	Java	54			0.21		0.79
			No	Industrial	VB	32	10 fold-CV + Fit		0.13		0.94
			Yes	OS	PHP	63			0.31		0.65
			No	Industrial	Java	32			0.11		0.94
			Yes	OS	Java	54			0.18		0.76
Our study	2013	CDM	Yes	OS	Java	100	10 times 10-fold CV	OLS, LMS, SVM, MLP, DT, REPTree, IBk, K*	0.20	0.17	0.71
		POPs							0.40	0.30	0.50
		FS_CP							0.30	0.26	0.49
		O_CP							0.22	0.18	0.68
		OOPS							0.19	0.16	0.74
		OOFP							0.28	0.23	0.55

RCD: recovered class diagram; N/A: not applicable; Elements: number of ILF DETs, number of ILF RETs, and number of SR DETs; DETs: number of ILF.
DETs and number of SR DETs; OS: open source; Various: Python, JSP, PHP, ASP, Perl, Java, and JavaScript; Exp.: exponential; Poly: polynomial.

generalized to the population under study and other research settings.

5.5.1. Construct validity

The dependent variable used in this study is the total non-blank and non-commentary source lines of code (SLOC) of the classes in each Java system. We used the mature commercial tool, Understand for Java, to collect SLOC. This tool has been used to collect the metric data in previous studies [66,67]. In particular, we randomly selected a number of classes to manually inspect the SLOC data reported by Understand for Java and found that they were reliable. Therefore, the construct validity of the dependent variable can be considered as satisfactory.

The independent variables used in this study are the CDM, POPs, FS_CP, O_CP, OOPS, and OOFP metrics, which should be collected from the UML class diagram of each system. However, our study used 100 open-source Java systems whose class diagrams were not available. To collect the data, we therefore first recovered their class diagrams from source codes. Then, we computed these metrics from the recovered class diagrams. In our study, we use the Perl APIs provided by Understand for Java to develop a reverse engineering tool to automatically recover class diagrams from source codes. We took two measures to ensure that class diagrams were correctly recovered. First, we used our reverse engineering tool to recover the class diagram for a small-scale Java system (developed by ourselves) in which different relationships among classes were implemented. We found that our tool produced reliable results. Second, we randomly examined a small number of Java systems and found that the UML class diagrams produced by our reverse engineering tool were reliable. Nonetheless, these recovered class diagrams may be different from those constructed at the early phase of software development, which is a potential threat to the construct validity of the independent variables. This threat needs to be eliminated by using the metric data collected from the real class diagrams constructed at the early development phase in the future work.

5.5.2. Internal validity

The first threat to the internal validity of this study is the unknown effect of not counting interfaces. In our study, we do not take into account interfaces in UML class diagrams, as the original definitions of POPs, FS_CP, O_CP, OOPS, and OOFP are based on classes. However, excluding interface, an important element in class diagrams, may bias the relationships between these metrics and the system size. To examine this effect, we regarded "interfaces" as a special kind of classes and then re-computed the POPs, FS_CP, O_CP, OOPS, and OOFP metrics. In particular, we added two new class diagram metrics, NI (number of interfaces) and NORR (number of realization relationships), to the CDM metrics suite. After that, we rerun the analysis to investigate the accuracy of six types of SLOC estimation approaches in combination with/without a logarithmic transformation under eight modeling techniques. We find that the overall results are very similar. Table 13 summarizes the mean and standard deviation (shown in parentheses) of MAR, MMRE, and Pred(25) for different prediction models with a logarithmic transformation obtained from 10 times 10-fold CV. As can be seen, OOPS + logarithmic transformation + OLS is still the optimal combination to build the size prediction model. This means that the inclusion of interfaces does not have a substantial effect on our results.

The second threat is the unknown effect of the formatting and style of source codes. In our study, SLOC is counted as the total non-blank and non-commentary source lines of code. However, SLOC is sensitive to logically irrelevant formatting and style conventions, which is a potential threat to the causal effect of independent variables on the dependent variables. Nonetheless, we believe that this threat should not have a substantial effect on our results, as our study uses 100 different software systems.

5.5.3. External validity

The most important threat to the external validity of this study is that our results may not be generalized to other systems. Although our study investigated 100 Java systems, we do not claim that our results can be generalized to all systems. This is because our experiments do not take into account many other project and human factors, such as developer experience, language used, and application domain. These factors can only be considered as our study is replicated across a wide variety of systems in future work.

6. Conclusions and future work

Based on 100 open-source Java systems, we analyzed and compared the accuracy of six types of metrics from UML class diagrams

Table 13
Comparison of prediction models with log transformation when interfaces are included.

Model	OLS	LMS	SVM	MLP	DT	REPTree	IBk	K*
(a) MAR								
CDM	3785(0.202)	3854(0.225)	3920(0.308)	5095(0.156)	6159(0.534)	6152(0.252)	5301(0.202)	4824(0.000)
POPs	6607(1.428)	6549(1.434)	6616(1.498)	7406(0.878)	7811(0.926)	8123(0.847)	9487(1.633)	7481(0.750)
FS_CP	4929(0.834)	5045(0.868)	5104(0.896)	5982(0.447)	5466(0.311)	6466(0.368)	7832(1.263)	6241(0.405)
O_CP	4047(0.371)	4062(0.351)	4110(0.448)	5227(0.199)	5129(0.163)	5661(0.090)	6298(0.667)	5514(0.197)
OOPS	3465(0.000)	3503(0.000)	3446(0.000)	4662(0.000)	4657(0.000)	5390(0.000)	4853(0.000)	5219(0.113)
OOFP	4699(0.732)	4827(0.776)	4653(0.756)	6099(0.491)	6261(0.524)	6543(0.354)	7003(0.961)	6100(0.364)
(b) MMRE								
CDM	0.209(0.263)	0.211(0.271)	0.211(0.297)	0.254(0.242)	0.335(1.406)	0.307(0.630)	0.262(0.003)	0.217(0.000)
POPs	0.380(1.994)	0.389(2.062)	0.400(1.997)	0.441(1.462)	0.424(1.747)	0.454(1.814)	0.546(2.298)	0.422(2.172)
FS_CP	0.310(1.677)	0.323(1.789)	0.313(1.726)	0.359(1.209)	0.329(1.239)	0.375(1.471)	0.505(1.909)	0.348(1.705)
O_CP	0.227(0.570)	0.230(0.603)	0.229(0.614)	0.269(0.431)	0.266(0.611)	0.276(0.261)	0.328(1.017)	0.261(0.711)
OOPS	0.196(0.000)	0.198(0.000)	0.197(0.000)	0.236(0.000)	0.229(0.000)	0.259(0.000)	0.262(0.000)	0.234(0.297)
OOFP	0.283(1.547)	0.281(1.519)	0.282(1.544)	0.336(1.048)	0.334(1.493)	0.333(0.909)	0.404(1.563)	0.312(1.455)
(c) Pred(25)								
CDM	0.654(0.640)	0.649(0.569)	0.640(0.725)	0.607(0.095)	0.469(1.072)	0.522(0.388)	0.555(0.312)	0.656(0.000)
POPs	0.475(1.912)	0.474(1.797)	0.465(2.101)	0.418(1.194)	0.446(1.247)	0.378(1.407)	0.273(2.343)	0.388(1.834)
FS_CP	0.511(1.587)	0.511(1.487)	0.503(1.687)	0.448(1.059)	0.488(0.929)	0.412(1.132)	0.346(1.715)	0.455(1.388)
O_CP	0.640(0.753)	0.650(0.587)	0.639(0.772)	0.576(0.294)	0.566(0.447)	0.551(0.196)	0.438(1.085)	0.582(0.504)
OOPS	0.741(0.000)	0.728(0.000)	0.740(0.000)	0.622(0.000)	0.631(0.000)	0.581(0.000)	0.600(0.000)	0.611(0.316)
OOFP	0.531(1.501)	0.521(1.492)	0.528(1.602)	0.474(0.938)	0.444(1.258)	0.492(0.594)	0.407(1.319)	0.475(1.227)

in combination with/without a logarithmic transformation under eight modeling techniques. We find that object-oriented project size metric, class diagram metrics, and objective class points metric are able to accurately predict source code size of object-oriented systems. Of these approaches, object-oriented project size metric performs best when predicting code size. We also find that applying a logarithmic transformation to the data can in general significantly improve the accuracy of the prediction model, regardless of which metrics and modeling techniques are considered. Of the investigated modeling techniques, ordinary least square regression with a logarithmic transformation performs as good as or better than the other modeling techniques. In particular, the model built with the combination of object-oriented project size metric, ordinary least square regression, and a logarithmic transformation produces the highest accuracy for the software code size prediction.

In the future work, we will replicate this study to validate the above-mentioned findings using more systems. In particular, our findings are based on Java systems. However, the usefulness of these code size estimation approaches may depend on the programming language used. There is, therefore, also a need to validate our findings with systems developed in other object-oriented programming languages such as C++ and C#.

Acknowledgement

We are very grateful to Professor Pierre Bourque for his help during the preparation of this paper. This work is supported by the National Key Basic Research and Development Program of China (2014CB340702), the National Natural Science Foundation of China (61272082, 61073029, 61300051, and 61021062), Natural Science Foundation of Jiangsu Province (BK20130014), the Climbing Plan of Nanjing University, the Hong Kong Competitive Earmarked Research Grant (PolyU5219/06E), and PolyU Grant (4-6934).

Table 14
Java systems used in this study.

No.	System name	Version	NC	SLOC	Website
1	Abbot Java Gui testing framework	1.0.0rc1	202	25212	http://sourceforge.net/projects/abbot/
2	Barcode4j	1.0	84	5009	http://sourceforge.net/projects/barcode4j/
3	Chess diagram editor	0.03	31	3608	http://sourceforge.net/projects/chessdiagedi/
4	Cofax	2.0rc1	96	17091	http://sourceforge.net/projects/cofax/
5	CSV loader	2.0	50	6306	http://sourceforge.net/projects/csvloader/
6	CUBA	3.0.0	131	7968	http://sourceforge.net/projects/cuba/
7	Daffodil Replicator	2.0	172	31909	http://sourceforge.net/projects/daffodilreplica/
8	DBEdit Plugin for Eclipse	2.3.3	56	3341	http://sourceforge.net/projects/dbedit/
9	DbForms	2.4	200	26497	http://sourceforge.net/projects/jdbforms/
10	DESMO-J	2.1.4b	234	28400	http://desmoj.sourceforge.net/home.html
11	DITA open toolkit	1.5M1	86	10503	http://sourceforge.net/projects/dita-ot/
12	DocSearcher	3.91.0	73	13971	http://sourceforge.net/projects/docsearcher/
13	DSpace	1.4.1	228	37767	http://sourceforge.net/projects/dspace/
14	Eclipse instant messenger plugin	1.0	105	7830	http://sourceforge.net/projects/eimp/
15	Eisenkraut	0.73	125	28273	http://sourceforge.net/projects/eisenkraut/
16	Ekit (Java HTML editor)	1.4	51	9479	http://sourceforge.net/projects/ekit/
17	eXtremeComponent	1.0.1RC3	122	8642	http://sourceforge.net/projects/extremecomp/
18	File explorer	0.1.0.a10	19	2096	http://sourceforge.net/projects/fileexplorer/
19	4Ever XML framework	3.3.1	102	9238	http://sourceforge.net/projects/fourever/
20	FreeMarker	2.3.10	231	36737	http://sourceforge.net/projects/freemarker/
21	ftp4j	1.5.1	27	3582	http://sourceforge.net/projects/ftp4j/
22	HTML parser	1.5	135	19376	http://sourceforge.net/projects/htmlparser/
23	httpunit	1.6	139	11992	http://sourceforge.net/projects/httpunit/
24	iValidator test framework	2.2.0	117	12607	http://sourceforge.net/projects/ivalidator/
25	j2ssh	0.2.8	395	36302	http://sourceforge.net/projects/j2ssh/
26	JAGA	0.9.b.1	87	5979	http://sourceforge.net/projects/j-a-g-a/
27	Jameleon	3.3-M1	110	9479	http://sourceforge.net/projects/jameleon/
28	Jasmin – a Java assembler	2.3	118	10906	http://sourceforge.net/projects/jasmin/
29	jAudio	1.0.1	379	49378	http://sourceforge.net/projects/jaudio/
30	JavaML	0.1.4	192	16563	http://sourceforge.net/projects/javaml/
31	javaXUL	0.4	38	3549	http://sourceforge.net/projects/javaxul/
32	Jaxe	3.2	106	23395	http://sourceforge.net/projects/jaxe/
33	JaxoDraw	2.0.0	177	29053	http://sourceforge.net/projects/jaxodraw/
34	Jazzy	0.5.1	38	4282	http://sourceforge.net/projects/jazzy/
35	JChart2D	3.0.0	182	12872	http://sourceforge.net/projects/jchart2d/
36	jCharts	0.7.5	90	6268	http://sourceforge.net/projects/jcharts/
37	jcurses	0.9.4	48	3694	http://sourceforge.net/projects/jcurses/
38	jdictionary	1.6	69	6518	http://sourceforge.net/projects/jdictionary/
39	jDip: A Diplomacy Mapper and Judge	1.5.1	199	41454	http://sourceforge.net/projects/jdip/
40	encreecia	0.0.12	102	10994	http://sourceforge.net/projects/encreecia/
41	jFileCrypt	021	22	1604	http://sourceforge.net/projects/jfilecrypt/
42	JGAP	3.4.4	275	24866	http://sourceforge.net/projects/jgap/
43	jGossip	1.0.0	170	8873	http://sourceforge.net/projects/jgossipforum/
44	JAVA gnuplot GUI	0.1.1	43	3121	http://sourceforge.net/projects/jgp/
45	Java interactive profiler	1.1.1	82	10056	http://sourceforge.net/projects/jiprof/
46	JLine	0.9.94	18	2530	http://sourceforge.net/projects/jline/
47	jManage	1.0.0	226	10756	http://sourceforge.net/projects/jmanage/
48	jMemorize	1.2.3	89	11034	http://sourceforge.net/projects/jmemorize/
49	Java mass JPEG resizer tool	1.0	23	3655	http://sourceforge.net/projects/jmjrst/
50	Java MSN messenger library	1.0b3	181	8681	http://sourceforge.net/projects/java-jml/
51	jMusic – Composition in Java	1.6.01	173	26579	http://sourceforge.net/projects/jmusic/

(*continued on next page*)

Table 14 (*continued*)

No.	System name	Version	NC	SLOC	Website
52	Joda – Time	1.5	132	24540	http://sourceforge.net/projects/joda-time/
53	Jodd	3.2.5	282	22393	http://sourceforge.net/projects/jodd/
54	joscar	0.9	309	14830	http://sourceforge.net/projects/joustim/
55	jphonelite	0.22	63	10431	http://sourceforge.net/projects/jphonelite/
56	JPoD	5.2	440	39230	http://sourceforge.net/projects/jpod/
57	Java persistent objects (JPOX)	1.1.1	546	100593	http://sourceforge.net/projects/jpox/
58	JRobin	1.2.0	105	10325	http://sourceforge.net/projects/jrobin/
59	JSettlers	1.1.11	195	48433	http://sourceforge.net/projects/jsettlers/
60	JSummer	0.0.6.0	56	5691	http://sourceforge.net/projects/jsummer/
61	Jsystem	1.3.0	41	2754	http://sourceforge.net/projects/jsystemtest/
62	jTDS	1.2.1	69	22082	http://sourceforge.net/projects/jtds/
63	LOCKSS	1.11.5	404	62235	http://sourceforge.net/projects/lockss/
64	Mandarax	3.3	342	22615	http://sourceforge.net/projects/mandarax/
65	MaVerickDBMS	2.2.0	48	2379	http://sourceforge.net/projects/maverick/
66	MeshCMS	3.4.1	91	9106	http://sourceforge.net/projects/meshcms/
67	Mime type detection utility	2.1.2	21	3281	http://sourceforge.net/projects/mime-util/
68	mvnForum	1.0.0rc4	290	50157	http://sourceforge.net/projects/mvnforum/
69	MX4J	3.0.1	464	36938	http://sourceforge.net/projects/mx4j/
70	Neuros Database Manipulator	1.40	241	25932	http://sourceforge.net/projects/neurosdbm/
71	Niggle web application framework	1.1pre5	98	7631	http://sourceforge.net/projects/niggle/
72	Observation manager	0.920	230	29781	http://sourceforge.net/projects/observation/
73	OpenFAST	0.9.8	141	6566	http://sourceforge.net/projects/openfast/
74	Open Java trading system	0.10	44	2975	http://sourceforge.net/projects/ojts/
75	opennlp-maxent	2.5.3	43	1963	http://sourceforge.net/projects/maxent/
76	OpenSAML	2.4.0	1075	21442	http://freshmeat.net/projects/opensaml
77	Pauker	1.8	37	12700	http://sourceforge.net/projects/pauker/
78	PDFBox	0.7.2	358	34304	http://sourceforge.net/projects/pdfbox/
79	Java SIP softphone	0.3.1	106	6186	http://sourceforge.net/projects/peers/
80	PMD	4.2.1	520	44659	http://sourceforge.net/projects/pmd/
81	Prevayler	2.02.006	28	1230	http://sourceforge.net/projects/prevayler/
82	Redstone XML RPC	1.1	40	2256	http://sourceforge.net/projects/xmlrpc/
83	RSS Owl|RSS/RDF/atom feed reader	1.2.1	190	42262	http://sourceforge.net/projects/rssowl/
84	Sesame	2.0alpha1	177	23997	http://sourceforge.net/projects/sesame/
85	SISC	1.9.0	151	13351	http://sourceforge.net/projects/sisc/
86	SketchEl	1.50	71	16409	http://sourceforge.net/projects/sketchel/
87	SQuirreL SQL Client	2.6.1	1364	131262	http://sourceforge.net/projects/squirrel-sql/
88	StatCVS – Stat your repository	0.4.0	120	7549	http://sourceforge.net/projects/statcvs/
89	TinyUML – free Java UML 2 editor	0.12	85	6116	http://sourceforge.net/projects/tinyuml/
90	Tin Whistle Calculator	2.04	65	5329	http://sourceforge.net/projects/twjcalc/
91	UML-Editor	3.1.5	315	67113	http://sourceforge.net/projects/umleditor/
92	UMLGraph	4.8	19	2408	http://freshmeat.net/projects/umlgraph
93	Veettukaaran	2.1.0	50	5459	http://sourceforge.net/projects/veettukaaran/
94	WebWork	1.2	157	10861	http://sourceforge.net/projects/webwork/
95	WIFE – open source SWIFT	6.0rc2	71	4596	http://sourceforge.net/projects/wife/
96	WLAN Traffic Visualizer	1.4.0	36	4398	http://sourceforge.net/projects/wlantv/
97	xBaseJ – xBase Engine for Java	20080710	40	8162	http://sourceforge.net/projects/xbasej/
98	XDoclet	1.2	221	20333	http://sourceforge.net/projects/xdoclet/
99	xsocket	2.8.14	50	14638	http://sourceforge.net/projects/xsocket/
100	YAWL	Beta8.1	195	28430	http://sourceforge.net/projects/yawl/

Appendix A. 100 Java systems used in this study

Table 14 summarizes the Java systems used in our study, including the system name, the version number, the total number of classes (NC), the source code size (SLOC), and the project website.

Appendix B. Steps to recover class diagrams from source code

For each Java system, we use the following steps to recover the class diagram from its source code.

Step 1: exclude test code from the source code. We find that, for many Java systems, the source code downloaded from http://sourceforge.net/ and http://freshmeat.net contains test code. In [11], Antoniol et al. identified test code as an important factor that influenced the accuracy of code size estimation approaches and suggested that it should be excluded. Following this suggestion, we manually exclude the test code from each Java system.

Step 2: recover the nodes in the class diagram. Classes and interfaces are two types of nodes in a class diagram. In a Java system, classes/interfaces can be inner or non-inner. In our study, only non-inner classes/interfaces in the source code are identified as the nodes of the class diagram. In other words, we exclude inner classes/interfaces. The reason for this is that we aim to make the recovered class diagram as close to the real class diagram constructed at the early development phase as possible.

Step 3: recover the relationships among the nodes in the class diagram. We recover the following six types of relationships: association, aggregation, composition, generalization, realization, and dependency. The recovery of composition, generalization, realization, and dependency is trivial.

- *Composition*: If class A has an attribute whose type is class B, then there is a composition relationship between A and B. In the class diagram, A is the containing class and B is the contained class.

- *Generalization*: If class A "extends" class B, then there is a generalization relationship between A and B. In the class diagram, A inherits from B.
- *Realization*: If class A "implements" interface I, then there is a realization relationship between A and I. In the class diagram, A realizes I.
- *Dependency*: If a method in class A has a parameter whose type is class B, then there is a dependency relationship between A and B. In the class diagram, A depends on B.

The recovery of association and aggregation is a bit complex.

- *Aggregation*: Assume that `attributeName` is an attribute in class A. If one of the following conditions hold: (1) its type is an array and the type of the elements in this array is class B (for example "`B attributeName[]`"); or (2) its type is a parameterized type and the parameter type is class B (for example "`HashMap<B> attributeName`"), then there is an aggregation relationship between A and B. In the class diagram, A is the containing class and B is the contained class. Note that many commercial tools such as Rational Rose 2003 are unable to recover the second kind of aggregation (i.e. they are unable to deal with the parameterized types in Java code).
- *Association*: There is an association relationship between A and B, if the following two conditions hold: (1) there is a composition/aggregation relationship from A to B; and (2) there is also a composition/aggregation relationship from B to A. In the class diagram, A and B are connected by a bi-directional association relationship.

Step 4: recover the attributes/methods for the nodes in the class diagram. This step is trivial. For each non-inner class/interface, the remaining attributes are recovered as the regular attributes and the methods are recovered as the regular methods in the corresponding node in the class diagram.

Appendix C. OOPS based size prediction models

Our experimental results show that OOPS is the overall best metric for size estimation, regardless of whether a logarithmic transformation is applied to the data. In the following, we present the OOPS based size prediction models under different modeling techniques and give their prediction accuracy obtained from the model fitting. Note that IBk and K* are instance-based modeling techniques, which do not build an explicit model.

1. The models on the data without a logarithmic transformation

(1) OLS model

$$\mathrm{SLOC} = -1145.010 + 3.559 \times \mathrm{OOPS}$$

(2) LMS model

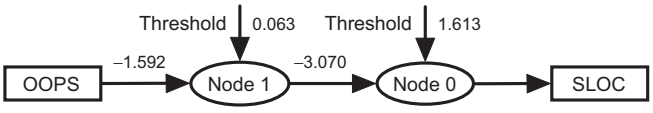

Fig. 1. MLP model when not applying a logarithmic transformation.

Table 15
Decision table model when not applying a logarithmic transformation.

OOPS	SLOC
(33030.6, +∞)	131262
(18617, 22220.4]	62235
(22220.4, 28523.8]	100593
(15013.6, 18617]	5824.5
(11410.2, 15013.6]	38121.333
(7806.8, 11410.2]	31028.583
(4203.4, 7806.8]	22857.55
(−∞, 4203.4]	6655.842

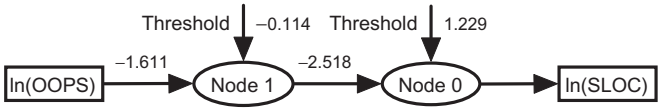

Fig. 3. MLP model when applying a logarithmic transformation.

Table 16
Decision table model when applying a logarithmic transformation.

ln(OOPS)	ln(SLOC)
(10.098, +∞)	11.785
(9.686, 10.098]	11.120
(9.275, 9.686]	10.516
(8.864, 9.275]	10.300
(8.453, 8.864]	9.908
(8.042, 8.453]	9.315
(7.630, 8.042]	8.939
(7.219, 7.630]	8.528
(6.808, 7.219]	8.018
(−∞, 6.808]	7.765

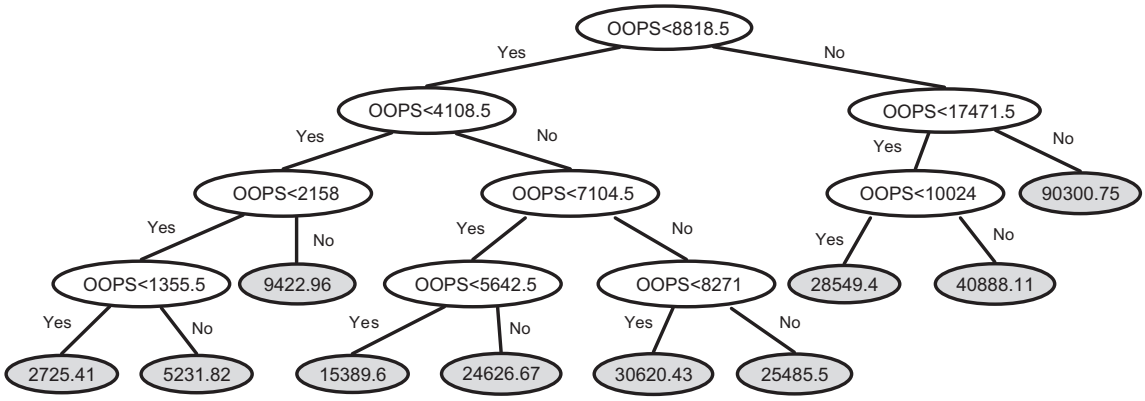

Fig. 2. REPTree model when not applying a logarithmic transformation.

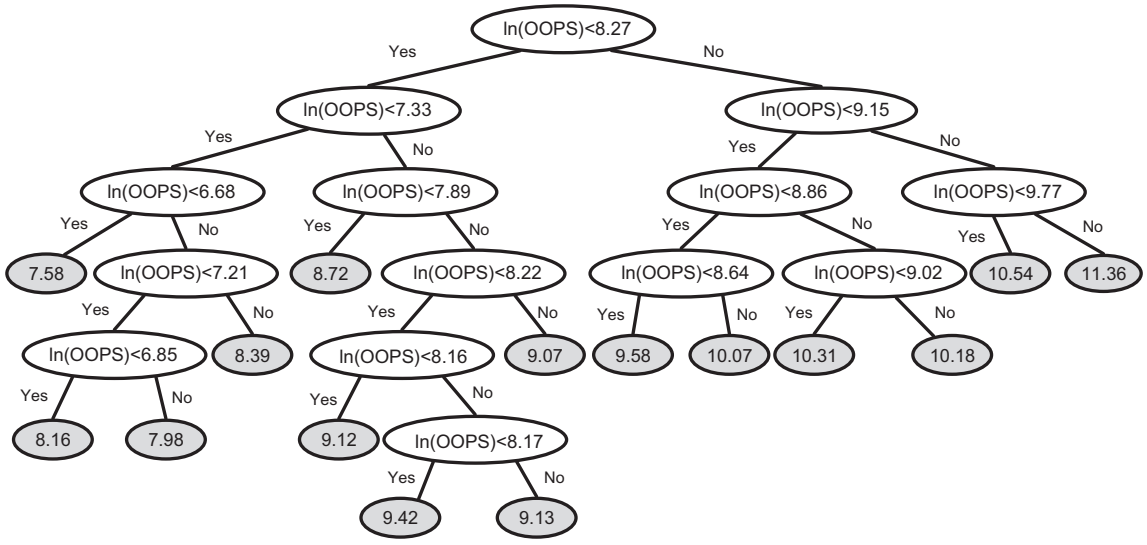

Fig. 4. REPTree model when applying a logarithmic transformation.

Table 17
Goodness-of-fit for OOPS based models

Model	MAR	MMRE	MdMRE	Pred(25)
(a) Without a logarithmic transformation				
OLS	3187	0.207	0.160	0.670
LMS	3213	0.192	0.160	0.730
SVM	3190	0.212	0.170	0.680
MLP	3438	0.282	0.200	0.600
DT	4629	0.503	0.290	0.460
REPTree	3588	0.206	0.160	0.740
(b) With a logarithmic transformation				
OLS	3168	0.187	0.160	0.740
LMS	3178	0.189	0.150	0.730
SVM	3146	0.186	0.150	0.740
MLP	3648	0.185	0.150	0.710
DT	3407	0.196	0.170	0.690
REPTree	3539	0.166	0.130	0.830

$$\mathrm{SLOC} = -1266.240 + 3.618 \times \mathrm{OOPS}$$

(3) SVM model

$$\mathrm{SLOC} = -411.048 + 3.364 \times \mathrm{OOPS}$$

(4) MLP model (see Fig. 1).
(5) DT model (see Table 15).
(6) REPTree model (see Fig. 2).

2. The models on the data with a logarithmic transformation

(1) OLS model

$$\ln(\mathrm{SLOC}) = 0.796 + 1.042 \times \ln(\mathrm{OOPS})$$

(2) LMS model

$$\ln(\mathrm{SLOC}) = 0.730 + 1.050 \times \ln(\mathrm{OOPS})$$

(3) SVM model

$$\ln(\mathrm{SLOC}) = 0.909 + 1.030 \times \ln(\mathrm{OOPS})$$

(4) MLP model (see Fig. 3).
(5) DT model (see Table 16).
(6) REPTree model (see Fig. 4).

Table 17 summarizes the model fitting results for the OOPS based models with/without a logarithmic transformation, including MAR, MMRE, MdMRE, and Pred(25). As can be seen, on the one hand, the model fitting results show that REPTree is the best performing technique, regardless of whether a logarithmic transformation is applied to the data. However, their 10 times 10-fold CV results shown in Tables 8–11 are far from the corresponding model fitting results. This reveals that the REPTree models overfit the data. On the other hand, we can see that, for OLS, LMS and SVM, their model fitting results are very close to the results obtained from the 10 times 10-fold CV. In particular, they have a higher prediction performance than the other modeling techniques. This indicates that OLS, LMS, and SVM models are more appropriate for system size prediction, although REPTree has the best goodness-of-fit performance. Furthermore, when a logarithmic transformation is applied to the data, the 10 times 10-fold CV results show that OLS is the best performing modeling technique.

References

[1] M. Genero, E. Manso, A. Visaggio, G. Canfora, M. Piattini, Building measure-based prediction models for UML class diagram maintainability, Empirical Software Engineering 12 (5) (2007) 517–549.
[2] A.F. Minkiewicz, B.E. Fad, Parametric Software Forecasting System and Method, United States Patent (6073107), 2000.
[3] F.G. Wilkie, I.R. McChesney, P. Morrow, C. Tuxworth, N.G. Lester, The value of software sizing, Information and Software Technology 53 (11) (2011) 1236–1249.
[4] M. Carbone, G. Santucci, Fast&&Serious: a UML based metric for effort estimation, in: Proceedings of the 6th International ECOOP Workshop on Quantitative Approaches in Object-oriented Software Engineering, 2002, pp. 35–44.
[5] M.V. Jahan, R. Sheibani, A new method for software size estimation based on UML metrics, in: The National Conference on Software Engineering, 2001. <http://vafaeijahan.com/e107/files/SiteFiles/SizeEstimationwithName.pdf>.
[6] S. Kim, W. Lively, D. Simmons, An effort estimation by UML points in the early stage of software development, in: Proceedings of the 2006 International Conference on Software Engineering Research & Practice, 2006, pp. 415–421.
[7] V.B. Mišić, D.N. Tešić, Estimation of effort and complexity: an object-oriented case study, Journal of Systems and Software 41 (2) (1998) 133–143.
[8] A.F. Minkiewicz, Measuring object oriented software with predictive object points, PRICE Systems (1997).
[9] D. Bradine, Oops, there it is: object-oriented project size estimation, Enterprise Systems Journal (2000).
[10] G. Antoniol, C. Lokan, G. Caldiera, R. Fiutem, A function point-like measure for object-oriented software, Empirical Software Engineering 4 (3) (1999) 263–287.
[11] G. Antoniol, C. Lokan, R. Fiutem, Object-oriented function points: an empirical validation, Empirical Software Engineering 8 (3) (2003) 225–254.
[12] V.D. Bianco, L. Lavazza, An empirical assessment of function point-like object-oriented metrics, in: Proceedings of the 11th International Software Metrics Symposium, 2005.

[13] V.D. Bianco, L. Lavazza, An assessment of function point-like metrics for object-oriented open-source software, in: Proceedings of the 2006 International Conference on Software Process and Product Measurement, 2006, pp. 21–28.
[14] V.D. Bianco, L. Lavazza, Object-oriented model size measurement: experiences and a proposal for a process, Workshop on Model Size Metrics, Part of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, 2006.
[15] M. Cartwright, M. Shepperd, An empirical investigation of an object-oriented software system, IEEE Transactions on Software Engineering 26 (8) (2000) 786–796.
[16] Y. Chen, B.W. Boehm, R. Madachy, R. Valerdi, An empirical study of eServices Product UML sizing metrics, in: Proceedings of the 2004 International Symposium on Empirical Software Engineering, 2004, pp. 199–206.
[17] G. Costagliola, F. Ferrucci, G. Tortora, G. Vitiello, Class point: an approach for the size estimation of object-oriented systems, IEEE Transactions on Software Engineering 31 (1) (2005) 52–74.
[18] H.B.K. Tan, Y. Zhao, Sizing data-intensive systems from ER model, IEICE Transactions on Information and Systems E89 (4) (2006) 1321–1326.
[19] G. Caldiera, G. Antoniol, C. Lokan, Definition and experimental evaluation of function points for object-oriented systems, in: Proceedings of the 5th International Software Metrics, Symposium, 1998, pp. 167–178.
[20] S. Abrahão, G. Poels, O. Pastor, Functional Size Measurement Method for Object-oriented Conceptual Schemas: Design and Evaluation Issues, Working Paper, Ghent University, 2004.
[21] Cigdem Gencel, O. Demirors, Functional size measurement revisited, ACM Transactions on Software Engineering and Methodology 17 (3) (2008).
[22] S.D. Conte, R.L. Campbell, A Methodology for Early Software Size Estimation, Technical Report, Purdue University, 1989.
[23] M.K. Rathi, An Objective Methodology for Early Software Size Estimation, Ph.D. Thesis, Purdue University, 1988.
[24] G. Stikkel, Z. Theisz, Prediction of size in executable model driven architectures, in: The Combined 14th Workshop for PhD Students in Object-oriented and Doctoral Symposium, in Association with the 18th European Conference on Object Oriented Programming, 2004.
[25] H.B.K. Tan, Y. Zhao, H. Zhang, Conceptual data model-based software size estimation for information systems, ACM Transactions on Software Engineering and Methodology 19 (2) (2009).
[26] H.E. Dunsmore, A.S. Wang, A step toward early size estimation for use in productivity models, in: Proceedings of the National Joint Computer Conference on Software Quality, 1985.
[27] C.F. Kemerer, An empirical validation of software cost estimation, Communication of the ACM 30 (5) (1987) 416–430.
[28] K. Lind, R. Heldal, A practical approach to size estimation of embedded software components, IEEE Transactions on Software Engineering 38 (5) (2012) 993–1007.
[29] J. Verner, G. Tate, A software size model, IEEE Transactions on Software Engineering 18 (4) (1992) 265–278.
[30] P. Bourque, V. Côté, An experiment in software sizing with structured analysis metrics, Journal of Systems and Software 15 (2) (1991) 159–172.
[31] W.G. Hopkins, A New View of Statistics, Sport Science, New Zealand, 2003.
[32] G. Antoniol, F. Calzolari, L. Cristoforetti, R. Fiutem, G. Caldiera, Adapting function points to object-oriented information systems, in: Proceedings of the 10th International Conference on Advanced Information Systems Engineering, 1998, pp. 59–76.
[33] S. Abrahão, G. Poels, O. Pastor, A functional size measurement method for object-oriented conceptual schemas: design and evaluation issues, Software and System Modeling 5 (1) (2006) 48–71.
[34] J.C. Reinard, Communication Research Statistics, Sage Publications Inc., Thousand Oaks, California, 2006.
[35] D.C. Howell, Statistical Methods for Psychology, Duxbury Press, Belmont, CA, 2002.
[36] D. Janaki Ram, S.V.G.K. Raju, Object oriented design function points, in: Proceedings of the 1st Asia–Pacific Conference on Quality Software, 2000, pp. 121–126.
[37] L.A. Laranjeira, Software size estimation of object-oriented systems, IEEE Transactions on Software Engineering 16 (5) (1990) 510–522.
[38] K. Dejaeger, W. Verbeke, D. Martens, B. Baesens, Data mining techniques for software effort estimation: a comparative study, IEEE Transactions on Software Engineering 38 (2) (2012) 375–397.
[39] S. Mishra, K.C. Tripathy, M.K. Mishra, Effort estimation based on complexity and size of relational database system, International Journal of Computer Science & Communication 1 (2) (2010) 419–422.
[40] B. Henderson-Sellers, Corrigenda: software size estimation of object-oriented systems, IEEE Transactions on Software Engineering 23 (4) (1997) 260–261.
[41] H. Leung, Z. Fan, Software cost estimation, Handbook of Software Engineering & Knowledge Engineering, vol. 2, World Scientific Publishing, 2002.
[42] S.L. Pfleeger, F. Wu, R. Lewis, Software Cost Estimation and Sizing Methods: Issues and Guidelines, RAND Corporation, 2005.
[43] M. Genero, M. Piattini, C. Calero, Early measures for UML class diagrams, L'Objet 6 (4) (2000) 489–515.
[44] M. Jorgensen, M. Shepperd, A systematic review of software development cost estimation studies, IEEE Transactions on Software Engineering 33 (1) (2007) 35–53.
[45] B. Boehm, Software Engineering Economics, Prentice-Hall, Englewood Cliffs, NJ, 1981.
[46] B. Boehm, C. Abts, A.W. Brown, S. Chulani, B.K. Clark, K. Horowitz, R. Madachy, D. Reifer, B. Steece, Software Cost Estimation with COCOMO II, Prentice Hall, NJ, 2000.
[47] The Function Point Counting Practices Manual Version 4.3.1. International Function Point Users Group (IFPUG), January 2010.
[48] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, IEEE Transactions on Software Engineering 32 (11) (2007) 2–13.
[49] Y. Shi, R.D. Reitz, Assessment of multi-objective genetic algorithms with different niching strategies and regression methods for engine optimization and design, Journal of Engineering for Gas Turbines and Power 132 (5) (2010).
[50] T.E. Hastings, A.S.M. Sajeev, A vector-based approach to software size measurement and effort estimation, IEEE Transactions on Software Engineering 27 (4) (2001) 337–350.
[51] T. Menzies, J.S. DiStefeno, M. Chapman, K. Mcgill, Metrics that matter, in: Proceedings of the 27th NASA SEL Workshop on Software Engineering, 2002.
[52] M. Haug, E.W. Olsen, L. Bergman, Software Process Improvement: Metrics, Measurement, and Process Modelling, Springer, 2001.
[53] S. Moser, O. Nierstrasz, The effect of object-oriented frameworks on developer productivity, IEEE Computer 29 (9) (1996) 45–51.
[54] A.J. Albrecht, J.E. Gaffney, Software function, source lines of code, and development effort prediction: a software science validation, IEEE Transactions on Software Engineering 9 (6) (1983) 639–648.
[55] C. Gencel, R. Heldal, K. Lind, On the relationship between different size measures in the software life cycle, in: Proceedings of the 16th Asia–Pacific Software Engineering Conference, 2009, pp. 19–26.
[56] C.J. Lokan, Early size prediction for C and Pascal programs, Journal of Systems and Software 32 (1) (1996) 65–72.
[57] C.J. Lokan, Function points, Advances in Computers 65 (2005) 297–347 (Chapter 7).
[58] B. Touesnard, Software Cost Estimation: SLOC-based Models and the Function Points Model, 2004. <http://bradt.ca/docs/SWE4103-report.pdf>.
[59] J.E. Matson, B.E. Barrett, J.M. Mellichamp, Software development cost estimation using function points, IEEE Transactions on Software Engineering 20 (4) (1994) 275–287.
[60] K. Lind, R. Heldal, On the relationship between functional size and software code size, in: Proceedings of the 32nd ICSE Workshop on Emerging Trends in Software Metrics, 2010.
[61] K. Lind, R. Heldal, Estimation of real-time software component size, Nordic Journal of Computing 14 (4) (2008) 282–300.
[62] L.C. Briand, J. Wüst, J.W. Daly, D. Victor Porter, Exploring the relationships between design measures and software quality in object-oriented systems, Journal of Systems and Software 51 (3) (2000) 245–273.
[63] T. Mukhopadhyay, S. kekre, Software effort models for early estimation of process control applications, IEEE Transactions on Software Engineering 18 (10) (1992) 915–924.
[64] S.G. MacDonell, Software source code sizing using fuzzy logic modeling, Information and Software Technology 45 (7) (2003) 389–404.
[65] K. El Emam, S. Benlarbi, N. Goel, S.N. Rai, The confounding effect of class size on the validity of object-oriented metrics, IEEE Transactions on Software Engineering 27 (7) (2001) 630–650.
[66] Y. Zhou, H. Leung, B. Xu, Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness, IEEE Transactions on Software Engineering 35 (5) (2009) 607–623.
[67] A.G. Koru, J. Tian, Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products, IEEE Transactions on Software Engineering 31 (8) (2005) 625–642.
[68] I.H. Witten, E. Frank, Data Mining: Practical Machine Learning Tools and Techniques, second ed., 2005.
[69] H.B.K. Tan, Y. Zhao, H. Zhang, Estimating LOC for information systems from their conceptual models, in: Proceedings of the 28th IEEE International Conference on Software Engineering (ICSE 2006), 2006, pp. 321–330.
[70] Y. Zhao, H.B.K Tan, Software cost estimation through conceptual requirement, in: Proceedings of the 3rd International Conference on Quality Software, 2003, pp. 141–144.
[71] B. Kitchenham, L. Pickard, S. MacDonell, M. Shepperd, What accuracy statistics really measure, IEE Proceedings – Software 148 (3) (2001) 81–85.
[72] T. Foss, E. Stensrud, B. Kitchenham, I. Myrtveit, A simulation study of the model evaluation criterion MMRE, IEEE Transactions on Software Engineering 29 (11) (2003) 985–995.
[73] D. Port, M. Korte, Comparative Studies of the Model Evaluation Criterions mmre and pred in Software Cost Estimation Research, ESEM, 2008, pp. 51–60.
[74] T. Menzies, Z. Chen, J. Hihn, K. Lum, Selecting best practices for effort estimation, IEEE Transactions on Software Engineering 32 (11) (2006) 883–895.
[75] J. Cohen, Statistical Power Analysis for the Behavioral Sciences, Lawrence Erlbaum Associates, 1988.
[76] Y. Benjamini, Y. Hochberg, Controlling the false discovery rate: a practical and powerful approach to multiple testing, Journal of the Royal Statistical Society, Series B (Methodological) 57 (1) (1995) 289–300.