Empir Software Eng (2014) 19:1383–1420
DOI 10.1007/s10664-013-9285-5

Labeling source code with information retrieval methods: an empirical study

Andrea De Lucia · Massimiliano Di Penta · Rocco Oliveto · Annibale Panichella · Sebastiano Panichella

Published online: 13 November 2013

Abstract To support program comprehension, software artifacts can be labeled—for example within software visualization tools—with a set of representative words, hereby referred to as labels. Such labels can be obtained using various approaches,

Communicated By: Michael Godfrey and Arie van Deursen

This paper is an extension of the work "Using IR Methods for Labeling Source Code Artifacts: Is It Worthwhile?" appeared in the *Proceedings of the 20th IEEE International Conference on Program Comprehension*, Passau, Bavaria, Germany, pp. 193–202, 2012. IEEE Press.

A. De Lucia · A. Panichella
Software Engineering Lab, University of Salerno,
Via Ponte don Melillo, 84084 Fisciano (SA), Italy

A. De Lucia
e-mail: adelucia@unisa.it
URL: http://www.dmi.unisa.it/people/delucia/

A. Panichella
e-mail: apanichella@unisa.it
URL: http://www.sesa.dmi.unisa.it/people/apanichella/

M. Di Penta · S. Panichella
RCOST, University of Sannio, Palazzo ex Poste,
Viale Traiano, 82100 Benevento, Italy

M. Di Penta
e-mail: dipenta@unisannio.it
URL: http://www.rcost.unisannio.it/mdipenta/

S. Panichella
e-mail: spanichella@unisannio.it
URL: http://www.ing.unisannio.it/spanichella/

R. Oliveto (✉)
Department of Bioscience and Territory, University of Molise,
C.da Fonte Lappone, 86090 Pesche (IS), Italy
e-mail: rocco.oliveto@unimol.it
URL: http://www.distat.unimol.it/people/oliveto/Home.html

including Information Retrieval (IR) methods or other simple heuristics. They provide a bird-eye's view of the source code, allowing developers to look over software components fast and make more informed decisions on which parts of the source code they need to analyze in detail. However, few empirical studies have been conducted to verify whether the extracted labels make sense to software developers. This paper investigates (i) to what extent various IR techniques and other simple heuristics overlap with (and differ from) labeling performed by humans; (ii) what kinds of source code terms do humans use when labeling software artifacts; and (iii) what factors—in particular what characteristics of the artifacts to be labeled—influence the performance of automatic labeling techniques. We conducted two experiments in which we asked a group of students (38 in total) to label 20 classes from two Java software systems, JHotDraw and eXVantage. Then, we analyzed to what extent the words identified with an automated technique—including Vector Space Models, Latent Semantic Indexing (LSI), latent Dirichlet allocation (LDA), as well as customized heuristics extracting words from specific source code elements—overlap with those identified by humans. Results indicate that, in most cases, simpler automatic labeling techniques—based on the use of words extracted from class and method names as well as from class comments—better reflect human-based labeling. Indeed, clustering-based approaches (LSI and LDA) are more worthwhile to be used for source code artifacts having a high verbosity, as well as for artifacts requiring more effort to be manually labeled. The obtained results help to define guidelines on how to build effective automatic labeling techniques, and provide some insights on the actual usefulness of automatic labeling techniques during program comprehension tasks.

Keywords Program comprehension · Software artifact labeling · Information retrieval · Empirical studies

1 Introduction

Program comprehension is a key activity for software maintenance and evolution. The importance of program comprehension has been summarized by Rajlich and Wilde: "*software that is not comprehended cannot be changed*" (Rajlich and Wilde 2002). During program comprehension, developers read the source code aiming at building a *cognitive model* that is used to form a *mental model*, i.e., the developers' mental representation of the program to be understood and changed (Storey 2006).

Such a cognitive process could be tedious, error prone and time consuming in large software systems, where the developer is requested to read (and comprehend) a large number of source code lines. In such a scenario, developers spend more time reading and navigating the code than writing it (LaToza et al. 2006; Ko et al. 2006). Recently, several approaches have been proposed to facilitate the comprehension of large software systems. The key idea is to reduce the amount of information to read and comprehend by providing a "short" description of the source code which can be read quickly. In order to reach such a goal, the source code is automatically labelled by means of some representative words.

Such labels can be obtained by using Information Retrieval (IR) methods. In summary, IR techniques are used to identify keywords contained in source code

identifiers and comments that properly describe the artifact. Such a representation provides a bird-eye's view of the source code artifacts, that allows developers to look over software components quickly, and make more informed decisions on which parts of the source code they need to analyze in detail (Haiduc et al. 2010b). Indeed, many researchers have applied IR techniques to automatically "label" software artifacts. For example, Kuhn et al. (2007) used discriminant words from Latent Semantic Indexing (LSI) (Deerwester et al. 1990) concepts to label software packages; Thomas et al. (2010) used latent Dirichlet allocation (LDA) (Blei et al. 2003) to label source code changes; Gethers et al. (2011) used Relational Topics Model (RTM) (Chang and Blei 2010) to identify and relate topics in high-level artifacts and source code.

Despite the aforementioned research efforts, up to now few studies have been performed to analyze whether these automatic techniques are able to extract words that make sense and are actually relevant to developers (Haiduc et al. 2010b; Hindle et al. 2012). This lack motivates our work. Specifically, we investigate to what extent an IR-based source code labeling technique is able to identify relevant words in the source code, compared to the words humans would manually select during a program comprehension task. In essence, we aim at verifying whether the terms identified by an automatic technique overlap with terms selected by developers when building their mental model of a source code component. To this aim we conducted two experiments in which we asked 17 Bachelor's students and 21 Master's students, respectively, to describe 20 Java classes taken from two Java software systems—JHotDraw[1] and eXVantage[2]—using at most ten words extracted from the class source code and comments. Then, we analyzed:

1. to what extent the keywords identified using various IR techniques, i.e., VSM, LSI, LDA, and some *ad hoc* heuristic picking terms from specific part of the source code and comments, overlap with those identified by humans;
2. what kind of source code (and comment) elements were used by participants to produce the labels; and
3. what characteristics of the analyzed artifacts could influence the effectiveness of the various techniques used to automatically produce labels.

Results show that, overall, automatic labeling techniques are able to well-characterize a source code class, as they exhibit a relatively good overlap—ranging between 50 % and 90 %—with the manually-generated labels. The highest overlap is obtained by using the heuristic considering only terms extracted from method signatures and class comment. Such a result confirms to some extent the results achieved by studies conducted in the field of programming psychology which indicate that the code author's semantics are embedded in method names, parameter names, and method comments (Detienne 2002; Liblit et al. 2006).

Another interesting result is that LSI and LDA (which are based on artifact clustering) generally provide the worst overlap. In particular, we observed that the high entropy of terms contained in the source code artifacts—i.e., artifacts do not contain terms that dominate over the others—inhibits the capability of such

[1] http://www.jhotdraw.org/

[2] http://www.research.avayalabs.com/

techniques to efficiently identify and cluster topics in source code. Indeed, LSI and LDA were designed for analyzing heterogeneous collections, where documents can contain information about multiple topics (Lavrenko 2009). Unfortunately, this heterogeneity is not always present in source code artifacts, especially when considering a single class having a well-defined set of responsibilities, and thus few and strongly-coupled topics. In such a scenario, it is difficult to identify dominant terms that could be used to characterize a class in terms of topics.

We also analyze the effect of other factors (e.g., comment verbosity) on time required to participants for labeling source code as well as the effectiveness of automatic source code labeling techniques. We observed that the higher the comment verbosity, the lower the time required by participants to identify the keywords. This result confirms the importance of having comments in comprehension tasks (Lawrie et al. 2007; Takang et al. 1996). In addition, we also observed that clustering-based approaches (i.e., LSI and LDA) are worthwhile to be used on source code artifacts having a high comment verbosity, and also on artifacts requiring more effort to be manually labeled.

Paper Structure Section 2 describes the empirical study definition and planning. Section 3 reports and discusses the results. Section 4 discusses the threats that could affect the validity of our study. Section 5 overviews related work, while Section 6 concludes the paper.

2 Study Definition and Planning

In the following, we report the definition and planning of our empirical study. The experiment replication package and working data sets of its results are available online.[3]

2.1 Study Definition

The *goal* of our study is to compare source code labelings automatically generated by means of IR techniques and other simple heuristics with labels produced by humans. The *quality focus* concerns the quality of automatically-generated source code labels, measured as their overlap with the human-generated labels. The *perspective* is of researchers interested in understanding to what extent automatic source code labeling approaches based on IR methods or simple heuristics can be used, and in which circumstances each technique performs well or not.

2.2 Study Context

The *context* of our study consists of *objects*, i.e., classes extracted from two Java software systems, and *participants*, i.e., undergraduate students from the University of Molise, Italy, and graduate students from the University of Salerno, Italy.

[3] http://distat.unimol.it/reports/labeling/

Table 1 Classes from JHotDraw and eXVantage used as objects of our study

JHotDraw	eXVantage
org.jhotdraw.draw.GraphicalCompositeFigure	com.avaya.exvantage.decision.gui.Browser
org.jhotdraw.draw.TextTool	com.avaya.exvantage.structures.cfg.cfgmanager.CFGManager
org.jhotdraw.draw.SelectionTool	com.avaya.exvantage.source.representation.ast.CodeFragment
org.jhotdraw.io.ExtensionFileFilter	com.avaya.exvantage.structures.dependency.DependencyGraph
org.jhotdraw.app.action.OpenAction	com.avaya.exvantage.util.graph.EdgeElement
org.jhotdraw.draw.GroupFigure	trace.EventThread
org.jhotdraw.util.prefs.PreferencesUtil	com.avaya.exvantage.ui.interfaces.cli.ExvantageCommand
org.jhotdraw.draw.ArrowTip	com.avaya.exvantage.trace.format.session.TraceBitEncoder
org.jhotdraw.draw.GridConstrainer	com.avaya.exvantage.ui.trace.tracetransfer.TraceTransfer
org.jhotdraw.draw.TextAreaTool	com.avaya.exvantage.decision.util.graph.VarMatcher

Specifically, the object systems used in our study are JHotDraw and eXVantage. JHotDraw[4] is an open source vectorial drawing tool, developed with the purpose of illustrating the usage of design patterns. In our study we used version 6.0 b1 of JHotDraw, which consists of 275 classes (29 KLOC). eXVantage[5] is a novel testing and test data generation tool developed in an industrial environment. The version used in our study (V20090507173755) is composed of 348 classes (28 KLOC). In the context of our study, we selected ten classes from JHotDraw and ten from eXVantage. We selected classes that are not too trivial, nor too complicated to be understood by the experiment participants. Table 1 reports the list of classes we selected from the two systems.

Concerning the participants involved in our study, we performed two experiments. The first one—in the following referred to as *Exp. I*— involved 17 Bachelor's students attending the Software Engineering course at the University of Molise, Italy. The second one—in the following referred to as *Exp. II*—involved 21 Master's students attending the course of Advanced Software Engineering at the University of Salerno, Italy. In both cases, all participants were from the same class and had comparable academic backgrounds, but different demographics. All of them had knowledge of Java development (ranging from 1 to 5 years of experience), including experience in dealing with existing large software systems. In addition, Master's students had a previous experience in comprehending and maintaining existing systems, and all of them had spent an internship period in industry.

[4] http://www.jhotdraw.org

[5] http://www.research.avayalabs.com

2.3 Research Questions

In the context of our study we address the following research questions:

- **RQ1**: *How do the labels provided by automatic techniques overlap with labels produced by humans?* This is the main research question of our study which aims at quantifying the performance of an automatic technique when used to identify the keywords characterizing a source code artifact. In presence of a high overlap, it can be argued that the automatic technique reflects the mental model of developers when identifying keywords in source code during a comprehension task.
- **RQ2**: *What code elements are often used by humans when labeling a source code artifact?* This research question is an investigation on the developer's cognitive process when reading source code. Given the artifact labels produced by humans, the research question investigates which elements of source code and comments were used to produce the labels. We determine whether words constituting labels come from class, method, or statement level comments, class, method, or attribute names, return types and parameters, local variables, programming language keywords, and native types.
- **RQ3**: *What co-factors influence the effectiveness of automatic source code labeling techniques?* Our third research question aims at identifying specific characteristics of source code artifacts—such as verbosity or comment density—that could inhibit the performance of the automated technique in producing labels similar to those produced by humans. Such an analysis is also useful to understand which IR technique or heuristic is more suitable for particular source code artifacts.

2.4 Experimental Procedure

The study was organized in three steps, preceded by a training phase. In the first step, we asked developers to describe the selected source code classes with a set of up to 10 keywords (but not necessarily 10). Then, we applied different IR techniques and heuristics to automatically extracting keywords from the selected classes. Once the set of keywords identified by the experiment participants and the set of keywords identified by the automatic techniques were collected, we computed the overlap between them, and performed the various kinds of analyses needed to answer the research questions of our study. In the next subsections we provide details for each of these steps.

2.4.1 Step 0: Participants' Training

Before asking participants to label software artifacts, we made them familiar with the objects of the study. We provided the participants with access to both systems a month before sending the questionnaire. In addition, during the experiment students periodically met system experts to enrich their domain knowledge, and one of the authors (instructor of the course) also participated to the meetings to check the learning progress. Then, we presented the experimental procedure to be followed, to make each participant aware of the exact sequence of steps to perform. However,

to avoid any bias, we made sure participants were not aware of the research questions of our study.

2.4.2 Step 1: Human Labeling of Software Artifacts

The experiment was conducted offline, i.e., by sending the experimental material to the participants and asking them to return the result after a given period of time. Specifically, we sent to each participant two spreadsheet files (one for each object system) containing a questionnaire to be filled-in. Each spreadsheet file consists of eleven sheets. The first one aims at collecting participant's demographics, i.e., years of computer science schooling and years of programming experience with Java. The other ten sheets aim at collecting the keywords for each of the classes to be analyzed. Each sheet reports the full class name and requires the participant to provide a list of at most ten keywords, i.e., terms considered relevant in describing the class. A term could be any source code identifier or any word contained in a compound identifier (e.g., *createFileName*) or comments. In addition, we asked participants to provide for each class the time spent to identify the keywords, and rate the difficulty encountered to identify them on a Likert scale of 1 to 5, where 1 indicates low difficulty and 5 high difficulty.

Participants had two weeks to fill-in the questionnaire. This point deserves further discussion because the obvious alternative would have been performing the study on-line during a limited-time laboratory. On the one hand, the latter would have given us a higher level of control over the study settings, making sure all participants worked under the same condition, and also making sure that the information they provided us about the time spent to perform the task was correct. On the other hand, our priority was to make sure participants produced reliable labels, i.e., they did not produce poor labels because of lack of time or inadequate code understanding. For this reason, we opted for an off-line study.

Once collected all the questionnaires, we analyzed them to identify the keywords most used to label each class. In particular, for each class C_i, we first defined the set of unique terms $T_{C_i} = \{t_1, \ldots, t_m\}$ identified by the participants to describe C_i. For each term $t_j \in T_{C_i}$ we computed its level of agreement (LoA) as follows:

$$LoA_{C_i}(t_j) = \frac{f_{t_j}}{ns}\%$$

where f_{t_j} represents the frequency of the term t_j, i.e., the number of participants that used t_j to label C_i, and ns represents the number of participants involved in our study, i.e., 17 in Exp. I and 21 in Exp. II. The terms having a LoA higher than 50 % (i.e., the terms selected by at least half of the participants) represent the set of keywords (K_{C_i}) identified by participants to label the class C_i. As it should be clearer later, the purpose of this aggregation is to produce "aggregated labels" comprising words selected by the majority of participants, and to use such aggregated labels when comparing the overlap with automatic techniques (**RQ1**), when understanding the provenance of words (**RQ2**), and when analyzing the effect of co-factors on the effectiveness of automatic labeling techniques (**RQ3**).

2.4.3 Step 2: Automatic Labeling of Software Artifacts

In the second step of our study, we automatically identified the sets of keywords that could be used to label the selected classes. To identify such keywords, we used

three different IR techniques, namely VSM, LDA, and LSI, plus three customized heuristics extracting words from specific source code elements.

VSM (Baeza-Yates and Ribeiro-Neto 1999) aims at representing documents involved in an IR process as vectors in a m-dimensional space, where m is the size of the documents vocabulary. Documents can be represented as a $m \times n$ matrix (called *term-by-document matrix*), where n is the number of artifacts in the repository. A generic entry $w_{i,j}$ of this matrix denotes a measure of the weight (i.e., relevance) of the ith term in the jth document (Baeza-Yates and Ribeiro-Neto 1999).

LSI (Deerwester et al. 1990) is an extension of the VSM. It was developed to overcome the synonymy and polysemy problems, which occur with the VSM model (Deerwester et al. 1990). In LSI the dependencies between terms and between artifacts, in addition to the associations between terms and artifacts, are explicitly taken into account. For example, both "car" and "automobile" are likely to co-occur in different artifacts with related terms, such as "motor" and "wheel". To exploit information about co-occurrences of terms, LSI applies Singular Value Decomposition (SVD) (Cullum and Willoughby 1998) to project the original term-by-document matrix into a reduced space of concepts, and thus limit the noise terms may cause. Basically, given a term-by-document matrix A, it is decomposed into:

$$A = T \cdot S \cdot D^T$$

where T is the term-by-concept matrix, D the document-by-concept matrix, and S a diagonal matrix composed of the concept eigenvalues. After reducing the number of concepts to k, the matrix A is approximated with $A_k = T_k \cdot S_k \cdot D_k^T$.

Latent Dirichlet Allocation (LDA) (Blei et al. 2003) fits a generative probabilistic model from the term occurrences in a corpus of documents. The fitted model is able to capture an additional layer of latent variables which are referred to as topics. Basically, a document can be considered as a probability distribution of topics—fitting the Dirichlet prior distribution—and each topic consists of a distribution of words that, in some sense, represent the topic.

To apply VSM, LSI, and LDA we first extracted words from source code, by removing special characters, English stop words, and (Java) programming language keywords. Each remaining word is then split using the camel case splitting heuristic. Then, we performed a *morphological analysis* to bring back words to the same root, e.g., by removing plurals from nouns, and verb conjugations. The simplest way to do morphological analysis is by using a stemmer, e.g., the Porter stemmer (Porter 1980). In our study, we considered two kinds of corpora: (i) words from source code including comments and (ii) words from comments only.

Then, we weighted words using two possible indexing mechanisms:

1. *tf* (term frequency), which weights each words i in a document j as:

$$tf_{i,j} = \frac{rf_{i,j}}{\sum_{k=1}^{m} rf_{k,j}}$$

   where $rf_{i,j}$ is the raw frequency (number of occurrences) of word i in document j.
2. *tf-idf* (term frequency-inverse document frequency) which is defined as $tf\text{-}idf_{i,j} = tf_{i,j} \cdot idf_i$ where $tf_{i,j}$ is the term frequency defined above and idf_i (inverse document frequency) is defined as:

$$\log \frac{n}{df_i}$$

where df_i (document frequency) is the number of documents containing the word i. *tf-idf* gives more importance to words having a high frequency in a document (high *tf*) and appearing in a small number of documents, thus having a high discriminant power (high *idf*).

For what concerns VSM, we indexed all the classes of the system in a single corpus and weighted words using *tf* or *tf-idf*. Then, for each class we selected the h words (we chose $h = 10$) having the highest weights.

As for LSI, we applied it on each single artifact (i.e., class) rather than on the corpus composed of all the classes of the system. This was done by considering the textual corpus composing the body of each method as a document in the document-by-term space,[6] then projecting the document-by-term space into a document-by-concept space, reducing the number of concepts to k. We adopted such a strategy because we would like to precisely identify topics representing a given class, each of them consisting in distribution of words from the class itself (and thus not containing words belonging to other classes). Indeed, when applying LSI, a generic entry (i, j) of the term-by-document matrix that was zero before the application of SVD (indicating that the term i does not occur in the document j) can assume a value different than zero after the space reduction (indicating that even if the term i does not occur in the document j it has some importance in the LSI space for the document j). This could lead to select words to label the class that do not appear in the class.

For the choice of k we used the heuristic proposed by Kuhn et al. (2007) that provided good results when labeling source code artifacts, i.e., $k = n \cdot m^{0.2}$. After that, we multiplied again the three matrices T_k, S_k and D_k^T, obtaining a term-by-document matrix A_k where term weights have been projected into the LSI space. Once the matrix A_k was computed, we extracted the h words having the highest weights in the LSI space (i.e., A_k).

LDA was applied in the same way as LSI, i.e., by building a document-by-term space over the textual corpus of methods belonging to the class to be labeled, and then applying LDA over such a space.[7] A crucial issue in the application of LDA is choosing the number of topics. We started by setting it equal to the number of class methods (excluding getters and setters), thus assuming that each method has a specific behavior and hence brings a topic to the experiment (as done by Gethers et al. 2011), then we reduced it to half the number of methods, and finally we considered the extreme case of two topics only. LDA also requires the setting of the Dirichlet distribution parameters α and β (Porteous et al. 2008). We used for α and β default values, i.e., $\alpha = 0.1$ and $\beta = 0.1$ (Teh et al. 2006).

Once LDA has been applied, we labeled each class using two heuristics:

– *core topic*: the class is labeled by the h words of the topic having the highest probability in the obtained topic distribution;

[6]The number of unique terms ranges from 26 to 186, while the number of documents, i.e., methods, from 4 to 37.

[7]Note that both LSI and LDA were used in the same way by other authors to support different software engineering tasks. For instance, both the techniques have been applied at class level when computing class cohesion/coupling exhibiting good results (Liu et al. 2009; Marcus and Poshyvanyk 2005; Poshyvanyk and Marcus 2006).

– *core words*: all the words characterizing the extracted topics are considered, and ranked according to their probability in the obtained topic distribution. The top-*h* words are then used to label the class. In this way we can label a class using words belonging to different topics.

In addition to the IR methods above, we also use simple heuristics based on the conjecture that when labeling a class, developers are prone to give more emphasis to terms composing the class high-level structure. Based on such a conjecture, we label a class using three different heuristics:

1. The first one considers only terms from class name, signature of methods, and attribute names. In other words, it considers elements from the class design, whose names represent a description of the class state and behavior.
2. The second one considers the class-level comments (excluding licensing and copyright information)—similarly to what done by Haiduc et al. (2010a, b). The rationale here is that the class-level comments provide a meaningful description of the class itself.
3. The third one is a combination of the first two.

Finally, to obtain the labels, we selected the most representative words by ranking them using their *tf* and *tf-idf*, however always considering the words contained in the class name as part of the top-*h* words. This is because our conjecture is that the very first words a developer would use to describe a class are the words composing the class name itself. Also when using the above defined heuristics we (i) performed morphological analysis of the extracted terms; and (ii) we computed *tf* and *tf-idf* following the same indexing process used for VSM, i.e., we indexed all the classes of the system in a single corpus.

2.5 Addressing the Research Questions

In the following subsection we detail on how we analyzed the experimental results to address the research questions formulated in Section 2.3.

2.5.1 RQ1: Computing the Overlap Between Automatically- and Human-Generated Labels

To address **RQ1** we determined to what extent keywords identified by participants correspond to those generated by automatic techniques. To this aim, we computed the overlap between them using an asymmetric Jaccard overlap (Baeza-Yates and Ribeiro-Neto 1999). Formally, let $K(C_i) = \{t_1 \dots t_m\}$ and $K_{m_i}(C_i) = \{t_1 \dots t_h\}$ be the sets of labels of class C_i identified by the participants and the technique m_i, respectively. The overlap is computed as follows:[8]

$$overlap_{m_i}(C_i) = \frac{|K(C_i) \cap K_{m_i}(C_i)|}{K_{m_i}(C_i)}$$

[8]Note that in our case the asymmetric Jaccard overlap coincides with the precision measure (Baeza-Yates and Ribeiro-Neto 1999). Assuming that $K(C_i)$ represents the set of "correct" keywords, the overlap measures the number of identified keywords that are actually correct, i.e., precision.

It is worth noting that the size of $K(C_i)$ might be different than the size of $K_{m_i}(C_i)$. In particular, while the number of keywords identified by an automatic technique is always 10 (by construction we set $h = 10$), the number of keywords identified by participants could be more or less than 10 (depending on the level of agreement). For this reason, we used an asymmetric Jaccard to not penalize too much an automatic method when the size of $K(C_i)$ is higher than 10.

We statistically tested the presence of a significant difference among overlaps obtained for different labeling techniques using the Wilcoxon paired test (a paired test is necessary because we pairwise compare the overlap between the automatic and manual labeling). We used a two-tailed test to observe the differences in both directions; that, is, we do not know a priori which technique works better. Since we applied the Wilcoxon test multiple times, we had to adjust p-values. To this aim, we used the Holm's correction procedure (Holm 1979). This procedure sorts the p-values resulting from n tests in ascending order of values, multiplying the smallest by n, the next by $n - 1$, and so on. Results are interpreted as statistically significant at $\alpha = 5$ %.

In addition to the statistical comparison, we computed the effect-size of the observed differences using Cliff's delta (d) non-parametric effect size measure (Grissom and Kim 2005), defined as the probability that a randomly-selected member of one sample has a higher response than a randomly selected member of a second sample, minus the reverse probability. Cliff's d ranges in the interval $[-1, 1]$ and is considered small for $0.148 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$.

Finally, we checked whether results are consistent between the two experiments, by considering the effect of the two variables *Approach* (i.e., the IR technique or the heuristic used), *Experiment* (i.e., whether results pertain to Exp. I or Exp. II), and their interaction on the dependent variable *Overlap*. This was done using a two-way permutation test (Baker 1995), which is a non-parametric equivalent of the two-way Analysis of Variance (ANOVA). Differently from ANOVA, the permutation test does not require data to be normally distributed. The general idea behind such a test is that the data distributions are built and compared by computing all possible values of the test statistic while rearranging the labels (representing the various factors being considered) of the data points. We use the implementation available in the *lmPerm* R package. We set the number of iterations of the permutation test procedure to 500,000. Since this test samples permutations of combination of factor levels, multiple runs of the test may produce different results. We made sure to choose a high number of iterations such that results did not vary over multiple executions of the procedure.

2.5.2 RQ2: Determining the Origin of Human-Generated Labels

To address **RQ2**, we manually analyzed the labels produced by the participants and identified the code elements the words were taken from. The aim of this analysis is to investigate what elements of a source code file participants used more in their labels, and consequently understand why different IR techniques and especially different heuristics considered in **RQ1** exhibit different performance. Specifically, we considered the following source code elements:

– different kinds of comments, namely class level comments, method-level comments (e.g., Javadoc), and inline (statement) comments;

- different elements of method signatures, namely method names, return types, parameter names and types;
- attribute names and types;
- local variable names; and
- programming language keywords and pre-defined types, including Java pre-defined types such as *String*.

To determine the most likely used origins, for each source code file we used the oracle computed as explained in *Step 1* of our procedure and identified the element(s) each word is located in. Clearly, some words appeared in multiple locations (e.g., both in class names and comments). In such cases, the word was counted for both elements. Specifically, let $Terms(C_i, element) = \{t_1 \dots t_l\}$ be the terms extracted from the specific *element* of class C_i. We computed the percentage of terms extracted from each source code element that were also present in the oracle (*SrcTermsInOracle*):

$$SrcTermsInOracle(C_i, element) = \frac{|Terms(C_i, element) \cap K(C_i)|}{|K(C_i)|}$$

Then, to have a better picture of the behavior of developers when labeling source code, we computed the percentage of terms in the oracle that also appeared in each source code element: (*OracleTermsInSrc*)

$$OracleTermsInSrc(C_i, element) = \frac{|Terms(C_i, element) \cap K(C_i)|}{|Terms(C_i, element)|}$$

The rationale behind of these two indicators can be explained as follows. The former (*SrcTermsInOracle*) provides information about where do terms in the oracle come from, whereas the latter (*OracleTermsInSrc*) indicates the percentage of terms from a source that were deemed useful to produce the labels.

2.5.3 RQ3: Factors Influencing the Accuracy of Automatic Labeling

To address **RQ3**, we analyzed whether various characteristics of the source code artifacts could influence the overlap. First, we evaluated whether different labeling techniques might be affected by the specific characteristics of the analyzed documents. To this aim, we measured the entropy for distribution of terms in the class. Formally, let $t = \{t_1, \dots, t_m\}$ be the terms extracted from the class C_i. We can compute the term entropy of class C_i as follows:

$$H(C_i) = \sum_{j=1}^{m} \frac{tf_j}{n} \cdot \log\left(\frac{n}{tf_j}\right)$$

where tf_j represents the frequency of the term t_j, and $n = \sum_{k=1}^{m} tf_k$. Since $H(C_i)$ ranges between 0 and $log(m)$ we normalized it as $\widetilde{H}(C_i) = H(C_i)/\log(m)$.

The entropy of a class is inspired by the Shannon's definition of entropy (Shannon 1948), a measure of the uncertainty associated with a random variable which quantifies the information contained in a message produced by a data emitter. In our case, a class with low entropy is a class where there are few dominant terms, i.e., terms with higher *tf* as compared to the others. Classes with high entropy have terms with a uniformly distributed *tf*, hence labeling may become problematic. In such cases, techniques such as LSI or LDA may not effectively work to cluster artifacts. Once the

entropy distribution has been computed, we divide classes in those with high entropy (above median) and low entropy (below median), and analyze the performance of the various techniques on classes belonging to the "high" or "low" group.

Besides the class entropy, we investigated whether other characteristics of the classes, namely their size in terms of LOC, and the verbosity of their comments in terms of number of words, influence the performance of the automatic labeling approaches. We divided the classes into two groups, large classes (having a size in LOC higher than the median) and small classes (having a size smaller or equal than the median), and we performed a similar subdivision for the comments' verbosity. Then, we investigated whether the performance of the various automatic labeling techniques—in terms of their overlap with manual labels—vary between small/large classes and between classes with high/low comment verbosity. Our conjecture is that a larger textual corpus could favor some IR techniques that cluster artifacts, such as LSI and LDA, whereas for small classes simpler techniques would perform just as well. From a statistical point of view, the effect of the entropy, class size and comment verbosity co-factors was analyzed by means of a two-way permutation test (Baker 1995).

Finally, we investigated the reasons why some classes required more time than others to be labeled. To this aim, we computed a Spearman rank correlation between the time needed to label the class–as reported by the participants—and the class size and comment verbosity. Our conjecture is that class size would be an indicator of difficulty in labeling the class; instead, verbose comments may either play a negative role (because a long time is required to read them) or a positive role (because they explain the source code in detail).

3 Analysis of the Results

This section discusses the results of our experiments aiming at answering the research questions formulated in Section 2.

3.1 Variation of Agreement Among Participants

Before comparing the labels produced by participants with those automatically generated by IR techniques, we analyzed the agreements among the manually-produced labels. Figure 1 shows—for the two systems and for the two experiments separately—the agreement between participants. Specifically, each point of the graph indicates the number of words selected by at least x participants. The analysis has been performed at class level, i.e., each line reports the agreement achieved on a specific object class.

As we can see, the variation of agreement between classes is fairly limited. This suggests that on each class the level of agreement among participants is almost the same. On the one side, only few words were selected by all the participants. On the other side, there are several words selected by only one participant (difference between the number of words selected by at least one participant and the number of words selected by at least two participants). This result highlights the subjectivity of source code labeling tasks. However, there is a good number of words (ranging from 5 to 10) selected by the majority of participants (LoA threshold represented

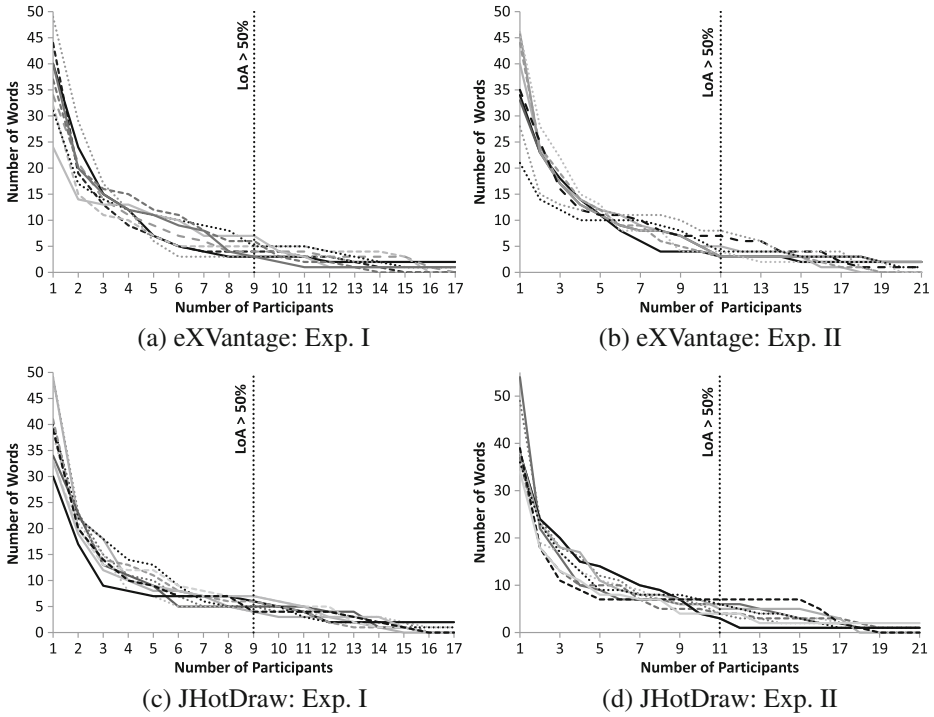


Fig. 1 Agreement between human generated labels. Each point of the graph represents the number of words selected by at least *x* participants

by the vertical dashed line in Fig. 1). As discussed in Section 2.4.2, this confirms that when producing an "aggregate" label by considering the LoA, a threshold of 50 % represents a reasonable choice. These labels were used as oracles for answering our research questions.

We also compared the overlap between the oracles. Table 2 reports for each object class the number of words in the oracle, as well as the intersection and the union of the oracles obtained in the two experiments. The results indicate that the number of words in the oracles is quite stable among the object classes, especially on JHotDraw. In addition, on JHotDraw, participants of Exp. I (Bachelor's students) selected a number of words greater than participants of Exp. II (Master's students). Nevertheless, the agreement between the participants of the two experiments is greater than 75 % for both the object systems. This confirms that there is a quite large set of words judged to be representative by the participants for labeling the object classes of both the experiments.

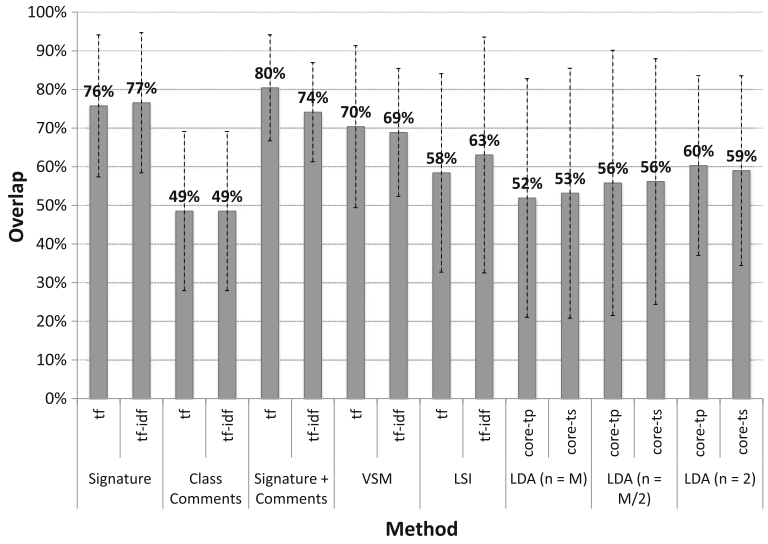
3.2 **RQ1**: How do the Labels Provided by Automatic Techniques Overlap with Labels Produced by Humans?

Figures 2 and 3 show—for eXVantage and JHotDraw respectively—the average overlap between the human-generated labels and the automatic labels obtained (i) using different IR methods (VSM, LSI, and LDA), (ii) the heuristic that extracts

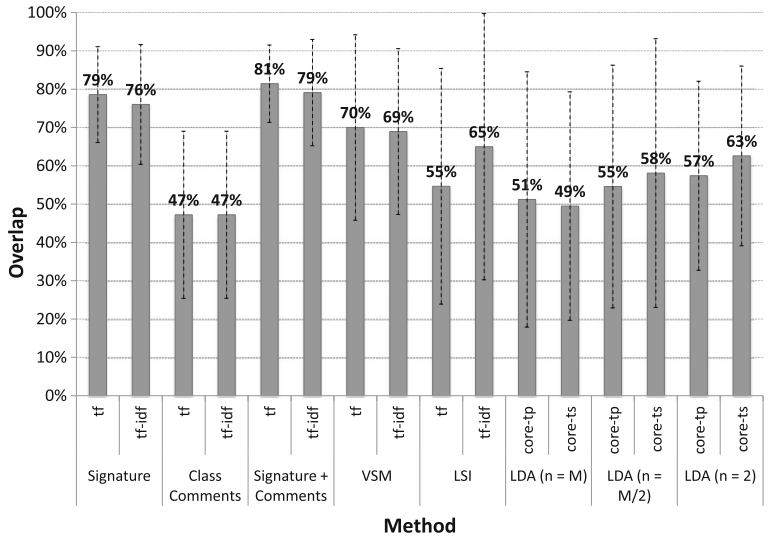
Table 2 Number of words in the "aggregate" labels (oracle) generated by humans

Class	Exp. I	Exp. II	Inters.	Union
(a) eXVantage				
Browser	5	3	3	5
CFGManager	5	6	4	6
CodeFragment	5	7	4	8
DependencyGraph	7	7	5	9
EdgeElement	4	5	4	5
EventThread	3	4	3	4
ExvantageCommand	8	7	7	8
TraceBitEncoder	9	8	8	9
TraceTransfer	10	7	7	10
VarMatcher	7	8	6	9
Total	63	62	51	73
(b) JHotDraw				
ArrowTip	9	7	7	9
ExtensionFileFilter	4	4	4	4
GridConstrainer	9	6	6	9
GroupFigure	7	4	4	7
OpenAction	6	5	5	6
TextAreaTool	8	7	7	8
PreferenceUtil	8	7	6	9
GraphicalCompositeFigure	7	6	6	7
SelectionTool	9	7	7	9
TextTool	8	7	6	9
Totale	75	60	58	77

words from class interfaces (referred as "signature"), (iii) the heuristic that extracts words from class-level comments (referred as "class comments") (iv) the heuristic that combines the previous two heuristics (referred as "signature + comments"). The figures also show the variability (standard deviation) of the overlap obtained by the different methods over different classes. The analysis reveals that the overlap achieved by all the experimented methods for automatic labeling varies between 40 % and 90 %. In particular, results indicate that—in both experiments and for both systems—LSI and LDA achieve the worst overlap if compared to the simpler VSM (which, unlike LSI and LDA, does not reduce the term space into a topic/concept space), and to the signature and signature + comments heuristics. Specifically, the different variants of LSI and LDA achieve an average overlap with human labeling varying between 40 % and 65 %, whereas VSM obtains an average overlap about 3 % higher than what achieved by LSI and LDA. The best performance are achieved by the simple heuristics (signature and signature + comments): in such cases the overlap is always higher than 75 % for eXVantage and 85 % for JHotDraw. Instead, considering comments only does not guarantee good performance (the overlap is always smaller than 50 %). In essence, a heuristic that considers class comments to automatically produce labels is not sufficient to achieve good performance. Our interpretation of this result is the following: taking the most frequent words from class-level comments would likely pick random words, because it is difficult to build a vector space model on such a small corpus. For this reason, automatically generating a label from class comments does not produce satisfactory results.



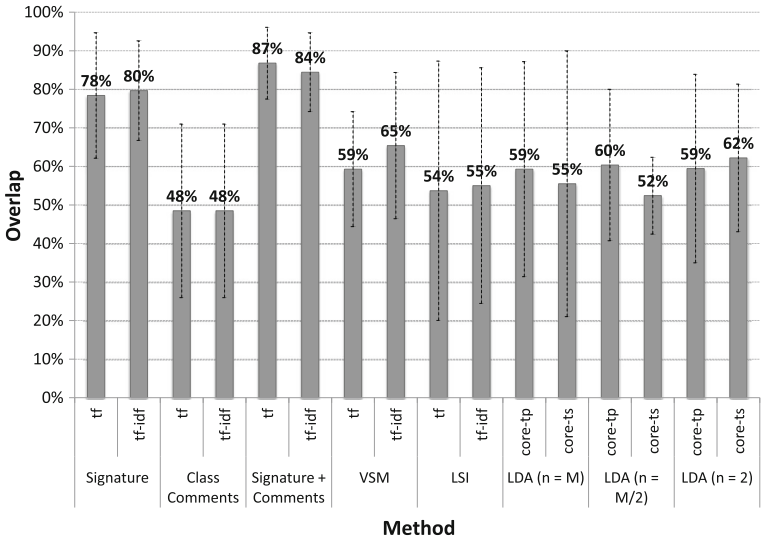
(a) Exp. I



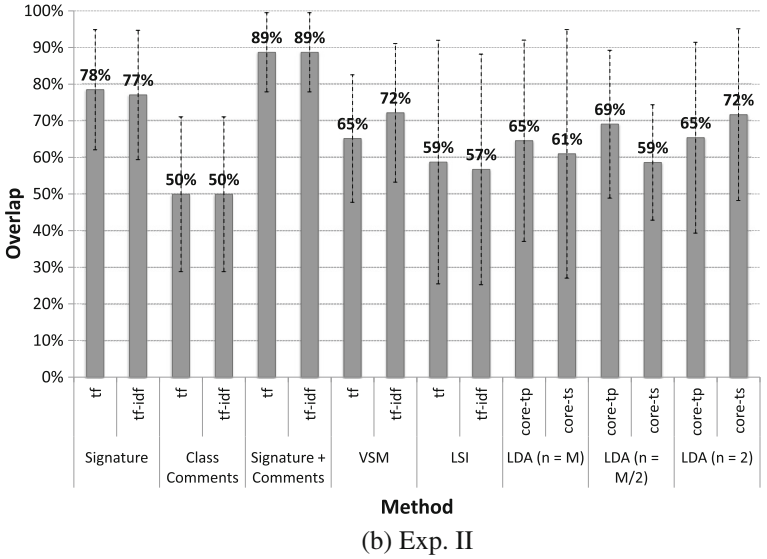
(b) Exp. II

Fig. 2 eXVantage: mean overlap between automatically-produced labels and manually-generated labels

With the aim of statistically supporting our results, Table 3 reports the Cliff's *d* values obtained in pairwise comparison of the various labeling methods, highlighted in bold face when p-values (adjusted using the Holm's correction) of the Wilcoxon Rank Sum test are statistically significant (i.e., when the adjusted p-value < 0.05). Since it would not be practical to show results related to all possible comparisons, we only compared the top-two performing heuristics (i.e., signature



(a) Exp. I



(b) Exp. II

Fig. 3 JHotDraw: mean overlap between automatically-produced labels and manually-generated labels

and signature + comments) with all other techniques. Results of the statistical tests and effect size measures confirm our preliminary findings discussed above. The signature+comments heuristic produces a statistically significant improvement with respect to other automatic methods in the majority of cases (38 cases out of 40). Moreover, the Cliff's *d* effect size is large in 73 % of the cases (29 out of 40), medium in 20 % of the cases (9 out of 40), and small in 5 % of the cases (2 out of 40). However, it is important to note that for the few cases (2 cases out of 80) for which

Table 3 Cliff's d for differences of overlap between automatic labeling and human labeling provided by each participant

Comparison	Exp. I		Exp. II	
	eXVantage	JHotDraw	eXVantage	JHotDraw
Sign. + class com. > VSM-tf	0.50 *(L)	0.45* (M)	0.62* (L)	0.31* (S)
Sign. + class com. > VSM-tf-idf	0.34^{+} (M)	0.39 (M)	0.48* (L)	0.29^{+} (S)
Sign. + class com. > LDA-core-tp (n = M)	0.89* (L)	0.51 * (L)	0.97* (L)	0.39* (M)
Sign. + class com. > LDA-core-ts (n = M)	0.90 *(L)	0.73 * (L)	0.94 *(L)	0.61 *(L)
Sign. + class com. > LDA-core-tp (n = M/2)	0.79* (L)	0.45* (M)	0.80 *(L)	0.41 *(M)
Sign. + class com. > LDA-core-ts (n = M/2)	0.82* (L)	0.68* (L)	0.89* (L)	0.56 *(L)
Sign. + class com. > LDA-core-tp (n = 2)	0.58* (L)	0.41* (M)	0.65 *(L)	0.45* (M)
Sign. + class com. > LDA-core-ts (n = 2)	0.53 *(L)	0.54* (L)	0.66 *(L)	0.43* (M)
Sign. + class com. > LSI-tf	0.61* (L)	0.76 *(L)	0.80 *(L)	0.72* (L)
Sign. + class com. > LSI-tf-idf	0.54* (L)	0.79* (L)	0.67* (L)	0.69 *(L)
Sign. > VSM-tf	0.39 *(M)	0.38* (M)	0.50 *(L)	0.26 *(S)
Sign. > VSM-tf-idf	0.23^{+} (S)	0.33^{+}(M)	0.36 * (M)	0.25 * (S)
Sign. > LDA-core-tp (n = M)	0.86 * (L)	0.41 * (M)	0.91* (L)	0.32 *(S)
Sign. > LDA-core-ts (n = M)	0.84* (L)	0.66 *(L)	0.88 * (L)	0.54 *(L)
Sign. > LDA-core-tp (n = M/2)	0.70* (L)	0.36* (M)	0.73* (L)	0.27 *(S)
Sign.> LDA-core-ts (n = M/2)	0.78* (L)	0.60* (L)	0.80* (L)	0.44* (M)
Sign. > LDA-core-tp (n = 2)	0.48 *(L)	0.34 *(M)	0.51* (L)	0.37 *(M)
Sign. > LDA-core-ts (n = 2)	0.45* (M)	0.46* (M)	0.51 *(L)	0.35* (M)
Sign. > LSI-tf	0.55* (L)	0.69 *(L)	0.71* (L)	0.61 *(L)
Sign. > LSI-tf-idf	0.48 *(L)	0.71 *(L)	0.55 *(L)	0.60 *(L)
Sign. + class comm. > Sign.	0.27 *(S)	0.08	0.22* (S)	0.06

Values shown with $*$ for comparisons where the Wilcoxon Rank Sum test indicates a significant difference. We use S, M, and L to indicate a small, medium and large effect size, respectively

there is no statistical difference—e.g., when comparing *Signature* with *VSM-tf-idf* on eXVantage and JHotDraw—the adjusted p-values range between 0.05 and 0.06, thus they are marginally significant. Moreover, the correspondent Cliff's *d* is positive (medium in some cases, small in other cases), highlighting an improvement for these cases too. The heuristic based on the signatures (without taking into account the class comments) also achieves a significant improvement with respect to other heuristics in the majority of cases (38 cases out of 40). The Cliff's *d* effect size value is large in 57 % of the cases (23 out of 40), medium in 30 % of the cases (12 out of 40), and small in 5 % of the cases (4 out of 40). In summary, the signature + comments heuristic outperforms the heuristics considering class signature and comments separately. A direct comparison between the two heuristics shows that there is a significant difference between them only for eXVantage where, as discussed above, comments seem to be able to improve the performance of the signature-based heuristic.

As also shown in Figs. 2 and 3, it is worthwhile to point out how, **RQ1** results are pretty consistent between Exp. I and Exp. II. This is also confirmed by a two-way permutation test (where we consider the overlap as dependent variable, and both the IR method used and the experiment as independent variables), which for both JHotDraw and eXVantage indicated that the overlap is significantly influenced by the *Approach* (p-value < 0.001), while it is not influenced by the *Experiment* (p-value $= 1$) nor by the interaction of the *Approach* and the *Experiment* (p-value $= 1$).

RQ1 Summary The achieved results indicate that the best heuristics for class labeling are simple heuristics considering class signatures and combining class signatures with class comments, while class comments alone do not perform well. As for IR-based methods, a simple VSM outperforms more sophisticated techniques such as LSI and LDA.

3.3 **RQ2**: What Code Elements are Often Used by Humans When Labeling a Source Code Artifact?

Table 4 reports *SrcTermsInOracle*, the percentage of terms in the human-generated oracle (obtained by computing the level of agreement, as explained in Section 2.4.2) appearing in various source code entities. Also, the table reports (in parentheses) *OracleTermsInSrc*, the percentage of terms belonging to source code entity that were actually used in the oracle.

The obtained results highlight that the majority of the terms suggested by the participants belong to *method names* (ranging between 43 % and 62 %), *method parameters* (ranging between 29 % and 52 %), *class comments* (ranging between 35 % and 60 %) and *method comments* (ranging between 46 % and 68 %). The percentage of labels taken from terms composing a class names is relatively lower (between 25 % and 48 %) than what obtained for method names and parameters, however, this only happens because class names are composed of few words, that do not suffice to properly describe the responsibility of the class itself. However, as the percentages in parentheses show, over 70 % of terms belonging to class names are used in the oracles, confirming that terms from class names are almost always taken when producing labels. We can also notice that, for eXVantage, the percentage of labels taken from terms composing class names is higher in the first experiment than in the second one. There is no straight-forward explanation for that. However, it might have happened that Exp. 1 participants (Bachelor's students) usually kept terms from class names as first and obvious choice, whereas Exp. 2 participants (Master's students) also looked elsewhere, and this happened in particular more for eXVantage than for JHotDraw, probably because the former has more comments than the latter (see Section 3.4.2).

Thus, terms describing the class interface, i.e., coming from method names and parameters, result to be particularly useful for the labeling process. The same can be said for class-level comments and method comments: this is quite intuitive because

Table 4 *SrcTermsInOracle* (percentage of oracle words belonging to different source code entities) and, in parentheses, *OracleTermsInSrc* (percentage of entity words considered in the oracle)

System	Exp.	Class comm.	Method comm.	Inline comm.	Class name	Attrib.	Param.	Method name	Returned type	Local var.	Keywords and predef. types
eXVantage	Exp. 1	44 % (26 %)	46 % (8 %)	17 % (12 %)	48 % (79 %)	29 % (24 %)	52 % (38 %)	43 % (33 %)	21 % (59 %)	37 % (39 %)	0 % (0 %)
	Exp. 2	35 % (21 %)	66 % (12 %)	15 % (10 %)	26 % (80 %)	34 % (28 %)	29 % (20 %)	48 % (37 %)	0 % (0 %)	15 % (15 %)	5 % (3 %)
JHotDraw	Exp. 1	51 % (15 %)	58 % (14 %)	32 % (20 %)	25 % (76 %)	34 % (60 %)	46 % (30 %)	62 % (39 %)	4 % (9 %)	30 % (23 %)	0 % (0 %)
	Exp. 2	60 % (14 %)	68 % (13 %)	35 % (17 %)	30 % (72 %)	35 % (49 %)	50 % (25 %)	50 % (25 %)	3 % (6 %)	32 % (19 %)	0 % (0 %)

class-level comments often contain short descriptions of the class and method behavior and responsibilities. This finding is fully in agreement with results of the study done by Haiduc et al. (2010b), who used the first words of a source code file—often corresponding to class-level comments—to summarize a class.

We can also observe that the percentage of oracle terms that appear in class and method comments is relatively low (below 30 %). Thus, on one hand oracles contain a high percentage of class and method comments. On the other hand comments also contain many words that do not appear in the oracle. This is quite expected considering the verbosity of comments if compared to a 10-words label. As a consequence, using comments alone to produce labels is not worthwhile due to the presence of many "false keywords" (see results of **RQ1** in Figs. 2 and 3). In essence, although words contained in class and method comments are useful to produce the label, the automatic heuristic is not able to discern the relevant words from the noise.

We found that terms taken from method local variables are relatively less used (between 15 % and 37 %), likely because they represent low-level entities that are, generally, not particularly relevant when describing the overall responsibilities of a class. Similar considerations apply for inline comments. Finally, method return types and programming language pre-defined types and keywords are almost never used. Again, such entities could be used when describing methods (where one could mention algorithmic details or returned values) while they are not needed when describing a class.

Table 5 reports the same analysis performed in Table 4 but aggregating code entities on the basis of the three heuristics used in this paper to extract labels: (i) signature (i.e., terms from class name, signature of methods, and attribute names), (ii) class-level comments; and (ii) a combination of signature and class-level comments. In all the cases more than the 80 % —and sometimes more than the 90 %— of terms used by participants are contained in the class name, signature of methods, and attribute names (signature). When considering both signature and comments, the percentage of oracle words belonging to these two code parts is very high. This further emphasize the benefits of the heuristic that labels classes by extracting terms from method signatures and class-level comments.

Turning to the percentage of terms belonging to source code entity that were actually used in the oracle, we observe that over 50 % of terms belonging to class name, signature of methods, and attribute names are used in the oracles, while this percentage is lower when considering the class-level comments. This confirms the important role of terms contained in the class signature when producing labels.

Table 5 *SrcTermsInOracle* (percentage of oracle words belonging to different source code entities) and, in parentheses, *OracleTermsInSrc* (percentage of entity words considered in the oracle)

System	Exp.	Comments	Signature	Comments + Signature
eXVantage	Exp. 1	38 % (21 %)	94 % (73 %)	97 % (51 %)
	Exp. 2	35 % (19 %)	80 % (59 %)	90 % (46 %)
JHotDraw	Exp. 1	49 % (40 %)	79 % (57 %)	81 % (39 %)
	Exp. 2	57 % (37 %)	90 % (52 %)	93 % (35 %)

RQ2 Summary In summary, we can conclude that, to label a class the experiment participants mainly used class and method names and signatures, as well as some words of class and method comments. However, comments also contain a high percentage of words not appearing in the oracle. This confirms why the heuristics signature and signature + comments produces very good results, while the comment heuristic alone does not.

3.4 RQ3: What Co-Factors Influence the Effectiveness of Automatic Source Code Labeling Techniques?

In the following, we analyze how the characteristics of the source code artifacts influence the performance of automatic labeling approaches. First, we study how such performance are related to the term entropy in the artifacts. Then, we investigate how the performance correlates with the class size and comment verbosity, and how such factors correlate with the difficulty for labeling such classes, measured in terms of the effort our participants spent in the labeling tasks.

3.4.1 Effect of Term Entropy

Results of **RQ1** indicate that techniques such as LSI and LDA—considered as promising ones for a number of software engineering tasks, including artifact labeling (Kuhn et al. 2007)—do not perform well if compared to simple heuristics or with the VSM. This could be due to the fact that both LSI and LDA are based on clustering analysis: all documents (classes in our case) are implicitly clustered on the basis of their shared latent concepts/topics. Source code artifacts having the same latent concepts/topics are considered as components of the same cluster, while artifacts having different latent concepts/topics are placed in different clusters. For this reason, such techniques are often used to identify topics in source code (see for example the work by Kuhn et al. (2007)). Even if such techniques represent an appealing solution for labeling and clustering source code documents, they were designed to analyze heterogeneous document collections, where documents usually contain information about multiple topics, which are pretty different from each other (Srivastava and Sahami 2009; Lavrenko 2009). In other words, such approaches work well if documents contain dominant terms—i.e., terms having a higher frequency than others—that can be used to derive and extract the topics discussed in the documents. More formally, LDA and LSI work well for documents having a low term entropy, i.e., documents for which the distribution of terms—proportional to their frequency—is not uniform across different documents, with few terms that are more frequent than others. In essence, because of the non-uniform distribution of terms—and their low entropy—it is easier for the clustering-based techniques to derive the dominant topics discussed in the documents.

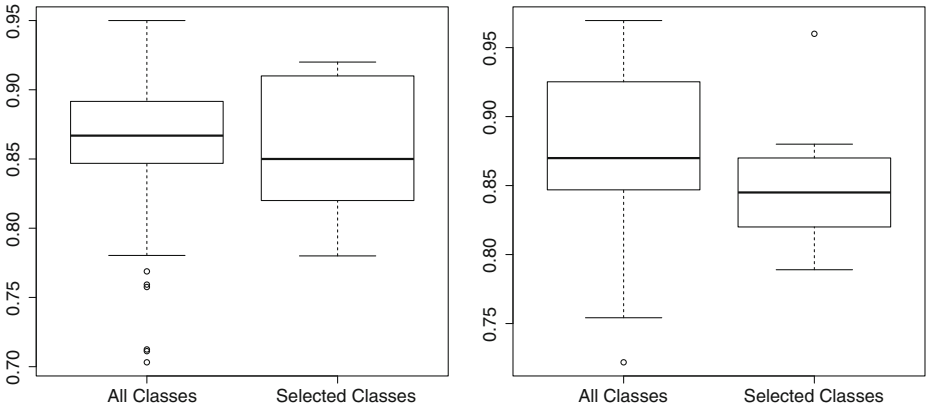
Differently from traditional natural language artifacts, heterogeneity is not always present in source code artifacts, especially when the goal is to extract topics from single classes, composed of "conceptually cohesive" (Marcus et al. 2008) methods. A possible explanation is that a class is a crisp abstraction of a domain/solution object, and should have a few, clear, responsibilities. Thus, a class has generally a few number of strongly-coupled topics. Moreover, the frequency of terms contained in the source code is very low. This means that when analyzing the textual content of classes, it is difficult to identify dominant terms that characterize the topics discussed

in the class itself. To verify such a conjecture, we computed the entropy of terms contained in the classes used in our study as described in Section 2.5.3. High entropy indicates that the probability distribution of the terms is quite uniform, reducing the capability of clustering-based techniques to identify dominant terms that can be used to characterize the class topics.

Figure 4 shows boxplots of the term entropy for the classes considered in our study. For both systems, the median term entropy (and even the first quartile) is greater than 0.8, indicating that the selected classes have a quite high entropy. Since a high entropy means that terms occurring in a class have almost the same probability, then it is hard to identify dominating terms that can be used to label such a class. This explains the poor performance of the two clustering-based IR methods, namely LSI and LDA.

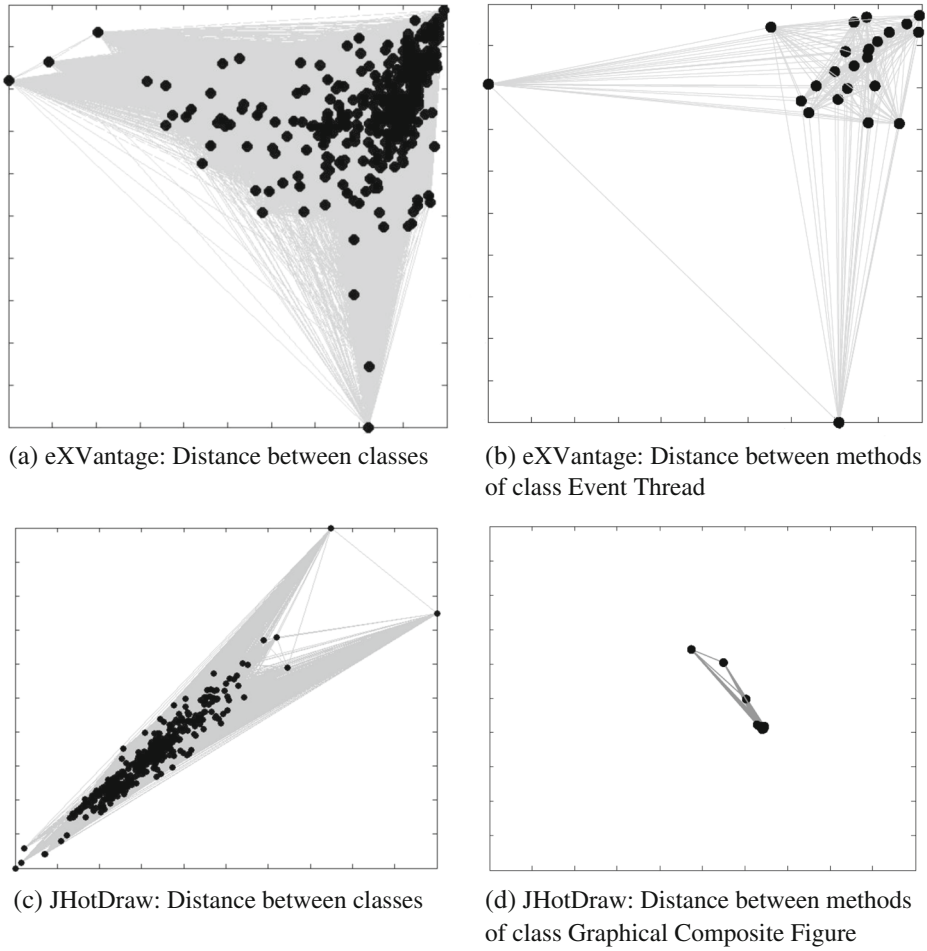
It is important to note that the classes selected in our study are not the only ones having a high entropy. Indeed, Fig. 4 also shows the entropy for all the classes of the two object systems. As we can see, the boxplots of the selected classes are comparable with those obtained computing the term entropy for all the classes, indicating that the selected class can be considered as representative of whole systems from entropy point of view.

Thus, we can conclude that clustering-based approaches will lead to some difficulties when dealing with highly-homogeneous collections, such as source code classes. To provide further evidence to this issue, Fig. 5 displays pairwise document distances achieved using LDA when clustering source code artifacts of eXVantage at two different levels of granularity. For eXVantage, Fig. 5a shows the results achieved by clustering all eXVantage classes, while Fig. 5b shows the clustering of methods of a class, `EventThread` (such a class was just taken as an example to show the distribution of topics among methods). Similarly, Fig. 5c shows the clustering of JHotDraw classes, while Fig. 5d shows the clustering of methods belonging to the JHotDraw `GraphicalCompositeFigure` class. Each node of the graph represents a class (or a method), while the weight of the edges measures the distance



(a) eXVantage (b) JHotDraw

Fig. 4 Entropy of terms in the object classes of our experiments and in all the classes of the object systems



(a) eXVantage: Distance between classes

(b) eXVantage: Distance between methods of class Event Thread

(c) JHotDraw: Distance between classes

(d) JHotDraw: Distance between methods of class Graphical Composite Figure

Fig. 5 Distance between source code elements

between topic distributions of each pair of classes (methods). Thus, if two classes (or methods) have different distribution of topics (i.e., they belong to different topics), then the corresponding nodes are far from each other in the graph. As we can notice from the figures, in all cases most classes (or methods) are concentrated in a small area. In such a scenario, it is quite difficult to discriminate between classes (or methods), and consequently efficiently clustering them.

To provide further evidence of the relationship between entropy and the overlap achieved by LDA, Table 6 shows two examples of eXVantage classes having high and low term entropy, respectively. The terms identified by LDA that overlap with those identified by humans are shown in bold face. For the class `CFGManager`, the entropy is very high (0.97), highlighting that all the terms extracted from the source code have the same frequency. In such a scenario it turns out to be difficult for LDA to identify the most important topic. Indeed, the class `CFGManager` has the same probability to discuss about one of the two main topics indifferently, since the two topics are quite

Table 6 Examples of high and low labeling overlap achieved on eXVantage

Class	Approach	# Topics	Topic Probability	Terms	Overlap	Entropy
CFGManager	LDA core-tp	2	0.49	string, **cfg**, key, mapping, filename, temp, file, global, instrument, integer	0.20	0.97
			0.51	list, temp, string, value, **cfg**, table, **classname**, name, map, key		
EdgeElement	LDA core-tp	2	0.26	**edge**, set, **node**, **element**, name, boundary, undefined, defined, illegal, access	1	0.78
			0.74	**node**, attribute, **graph**, **edge**, boundary, **element**, doc, inherit, parameter, value		

Terms in bold face indicate words also contained in the oracle

equiprobable. This results in a very low overlap between automatic and human label. In the latter case, there are some dominating terms (as indicated by the much lower entropy) and LDA is able to identify well-distinct topics in the class under analysis. Indeed, when applying LDA, we can observe that in `EdgeElement` one of the two topics has a probability greater than the other one. Such terms are selected by LDA to label the class and, as suggested by the high overlap, the same terms were also selected by subjects. The factor *entropy* affects not only LDA but also LSI that is a clustering technique too. In particular, for the two classes considered in the previous example when using *tf* as terms weighting schema, LSI obtains a higher overlap with human labeling (100 %) for `EdgeElement`, which exhibits a low term entropy, while for `CFGManager` the overlap is about 40 %.

Similar considerations apply for JHotDraw. Table 7 shows two examples of classes from JHotDraw with high and low term entropy respectively and using LSI as method for automatic labeling. The entropy is high for `PreferenceUtil`, while it is low for `GraphicalCompositeFigure`. Such different entropy values affect the capability of LSI to consistently identify and extract the main topic. Indeed, for `PreferenceUtil` LSI can not identifies the dominant terms and then the dominant

Table 7 Examples of high and low labeling overlap achieved on JHotDraw

Class	Approach	Terms	Overlap	Entropy
GraphicalCompositeFigure	LSI tf	**figure**, key, **presentation**, **attribute**, **composite**, value, **graphical**, set, **draw**, **bound**	1.00	0.82
PreferenceUtil	LSI tf	**preference**, name, **windows** , set, location, base, **frame**, **handler**, **install**, retrieve	0.62	0.87

Terms in bold face indicate words also contained in the oracle

topic, resulting in a very low overlap with automatic and human label. Conversely, for `GraphicalCompositeFigure` LSI is able to extract the dominant topic since the terms have not the same frequency for that class. This is mirrored by a high overlap. Indeed, the achieved overlap with human labeling is 62 % for `PreferenceUtil` and the 100 % for `GraphicalCompositeFigure`. In summary, these examples allow us to highlight the strong relationship between the entropy of terms and the performance of the clustering based automatic techniques, i.e., LDA and LSI. The higher the term entropy, the lower the performance achieved by the clustering techniques.

In order to generalize this kind of analysis for all the techniques, we investigate whether the term entropy influence the achieved overlap, or interacts with the adopted technique to this regard. For a first analysis, we grouped the classes in two groups (*low* and *high*), based on their terms entropy. In particular, a first group contains all classes for which the terms entropy value is higher than the median, and a second cluster contains the remaining ones. Then, we analyzed the overlap—as done in **RQ2**—for low and high entropy separately. Results are shown in Table 8. Such results indicate that all the techniques are effected by the entropy factor. For example, it can be noticed that VSM performs, on average, 10 % better on classes with lower entropy. Instead, the performance of LDA are not necessarily better for classes with lower terms entropy than the other: in the 50 % of cases LDA achieves a higher overlap with human labeling for classes with lower entropy values, while the scenario is opposite in the remaining cases. The proposed heuristics is also affected by the entropy factors, however, they always outperforms all the other techniques independently of the entropy of terms.

Table 8 Average overlap between manual labeling and automated labeling collected by the terms entropy (*high* and *low* terms entropy)

Corpus type		exVantage				JHotDraw			
		Exp. I		Exp. II		Exp. I		Exp. II	
		Low (%)	High (%)	Low (%)	High (%)	Low (%)	High (%)	Low (%)	High (%)
Signature	tf	**89**	63	81	76	75	82	74	83
	tf-idf	86	67	74	78	78	82	74	80
Class Comments	tf	52	45	47	47	50	47	53	46
	tf-idf	52	45	47	47	50	47	53	46
Sign. + Class Comments	tf	**89**	72	**84**	79	**84**	**90**	**89**	**89**
	tf-idf	73	**75**	77	**81**	81	88	**89**	**89**
VSM	tf	82	58	81	59	59	59	70	61
	tf-idf	77	61	76	61	64	67	76	69
LSI	core-tp	71	46	71	39	57	50	63	54
	core-ts	80	46	81	49	41	69	36	77
LDA(n = M)	core-tp	56	48	52	50	57	62	66	63
	core-ts	48	58	47	52	55	56	81	41
LDA(n = M/2)	core-tp	65	46	64	45	59	62	75	63
	core-ts	61	51	61	55	51	54	62	55
LDA(n = 2)	core-tp	72	58	72	42	54	65	71	60
	core-ts	71	47	70	55	59	65	80	64

M represents the number of methods in the class
Bold values indicate the technique that obtained the higher overlap in each of the two experiments

Table 9 Permutation test by *Approach* and term entropy

System	Exp.	Approach p-value	Entropy p-value	Approach: Entropy p-value
eXVantage	Exp. I	**0.04**	< **0.001**	**0.02**
	Exp. II	< **0.001**	< **0.001**	0.14
JHotDraw	Exp. I	< **0.001**	0.87	0.98
	Exp. II	< **0.001**	**0.004**	0.81

Bold values indicate p-values that are statistically significant (p-value < 0.05)

Table 9 reports the results of the permutation tests with the aims at providing statistical support to the results shown in Table 8. The entropy factor plays a significant role for both systems while it shows a significant interaction only for Exp. I and only on eXVantage. These results confirm that the automatic labeling is more difficult for classes with high term entropy independently of the specific automatic technique used.

3.4.2 Effect of Class Size and Comment Verbosity

The second kind of analysis performed to address **RQ3** investigates whether the class size (in LOC) and the verbosity of its comments influence the achieved overlap, or interacts with the adopted technique to this regard. For a first analysis, we grouped the classes in two groups (*large* and *small*), based on their size. In particular, a first group contains all classes for which the number of lines of code is higher than the median, and a second cluster contains the remaining ones. Then, we analyzed the overlap—similarly to what was done in **RQ2** for large and small classes separately. Results are shown in Table 10. Such results indicate that the simple heuristics

Table 10 Average overlap between manual labeling and automated labeling collected by the time needed to label classes (*small* and *large* class size)

Corpus type		exVantage				JHotDraw			
		Exp. I		Exp. II		Exp. I		Exp. II	
		Small (%)	Large (%)	Small (%)	Large (%)	Small (%)	Large (%)	Small (%)	Large (%)
Signature	tf	72.70	78.79	**85.00**	72.14	80.00	76.79	84.10	72.86
	tf-idf	69.84	83.29	79.17	72.86	80.00	79.29	81.24	72.86
Class Comments	tf	58.32	38.74	58.00	36.43	47.90	49.05	52.00	47.86
	tf-idf	58.32	38.74	58.00	36.43	47.90	49.05	52.00	47.86
Sign. + Class Comments	tf	**76.70**	**84.14**	**85.00**	**77.86**	**86.14**	**87.42**	**90.95**	**86.43**
	tf-idf	75.62	72.62	82.50	75.71	83.64	85.20	90.95	86.43
VSM	tf	74.48	66.29	80.00	60.00	64.99	53.57	71.71	58.57
	tf-idf	71.62	66.12	80.00	57.86	67.21	63.57	75.05	69.29
LSI	core-tp	54.54	62.29	55.00	54.31	72.01	35.36	78.15	39.29
	core-ts	70.48	55.62	75.00	54.94	52.17	57.86	54.15	59.29
LDA (n = M)	core-tp	51.89	51.93	56.00	46.43	62.49	56.07	65.52	63.57
	core-ts	50.67	55.62	46.83	52.14	64.53	46.43	64.10	57.86
LDA (n = M/2)	core-tp	45.33	66.29	49.17	60.00	62.13	58.57	71.71	66.43
	core-ts	48.89	63.43	53.33	62.86	53.74	51.07	61.52	55.71
LDA (n = 2)	core-tp	56.32	64.29	57.67	57.14	63.14	55.71	66.48	64.29
	core-ts	58.54	59.43	65.17	60.00	68.32	56.07	79.05	64.29

M represents the number of methods in the class
Bold values indicate the technique that obtained the higher overlap in each of the two experiments

Table 11 Average overlap between manual labeling and automated labeling collected by the time needed to label classes (*small* and *large* class comment verbosity)

Corpus type		exVantage				JHotDraw			
		Exp. I		Exp. II		Exp. I		Exp. II	
		Low (%)	High (%)	Low (%)	High (%)	Low (%)	High (%)	Low (%)	High (%)
Signature	tf	73	78	**80**	78	82	75	79	78
	tf-idf	72	81	69	83	82	78	76	78
Class Comments	tf	52	45	51	44	57	40	60	40
	tf-idf	52	45	51	44	57	40	60	40
Sign. + Class Comments	tf	**77**	**84**	**80**	83	**90**	**84**	**91**	**86**
	tf-idf	69	79	72	**86**	85	**84**	**91**	**86**
VSM	tf	71	70	74	66	66	52	67	63
	tf-idf	62	76	72	66	69	62	70	75
LSI	core-tp	53	63	56	53	80	27	83	35
	core-ts	62	64	64	66	62	48	59	54
LDA (n = M)	core-tp	54	50	69	33	64	55	61	68
	core-ts	47	59	57	42	61	50	56	66
LDA (n = M/2)	core-tp	42	70	61	49	66	55	73	65
	core-ts	50	62	68	49	53	52	57	60
LDA (n = 2)	core-tp	55	65	58	57	62	57	65	66
	core-ts	57	61	68	57	70	55	77	66

M represents the number of methods in the class
Bold values indicate the technique that obtained the higher overlap in each of the two experiments

(signature and signature + comments) exhibit good performance independently of the class size. Indeed, it can be noticed that VSM performs, on average, 9 % better on small classes than on large ones. Instead, the performance of LDA are not necessarily better for small classes than for larger classes. LDA seems to be independent of the class size: in the 46 % and the 62 % of cases, for the first and the second experiments respectively, achieves a higher overlap with human labeling for small classes, while the scenario is opposite in the remaining cases.

As shown in Table 11, consistent results were achieved when splitting classes in two different, groups, i.e., classes with *high verbosity* and *low verbosity* comments.

Results of Table 10 are also supported by the permutation test, whose results are shown in Tables 12 and 13, respectively. The two tables show consistent results. First, the *Approach* always has a significant effect (marginal for eXVantage in Exp. I). This is consistent with results of **RQ1**, which indicate that different labeling methods exhibit different performance. The class size plays a significant role for both systems in Exp. II, and for JHotDraw it also significantly interacts with the *Approach*. This partially contradicts with the results achieved for **RQ1**, which are fully consistent for Exp. I and II. As it can also be noticed from results of Table 10, participants of

Table 12 Permutation test by *Approach* and class size (LOC)

Bold values indicate p-values that are statistically significant (p-value < 0.05)

System	Exp.	Approach p-value	LOC p-value	Approach:LOC p-value
eXVantage	Exp 1	0.05	0.31	1.00
	Exp 2	**0.001**	**0.04**	1.00
JHotDraw	Exp 1	< **0.001**	0.88	0.62
	Exp 2	< **0.001**	**0.03**	**0.009**

Table 13 Permutation test by *Approach* and class comment verbosity

System	Exp.	Approach p-value	Verbosity p-value	Approach: Verbosity p-value
eXVantage	Exp 1	**0.04**	0.09	0.91
	Exp 2	< **0.001**	0.10	0.66
JHotDraw	Exp 1	< **0.001**	1.00	0.79
	Exp 2	< **0.001**	**0.04**	0.21

Bold values indicate p-values that are statistically significant (p-value < 0.05)

Exp. II (Master's students) performed worse on larger classes than in smaller classes, which surprisingly did not happen in Exp. I (Bachelor's students). Similar results (though we have significant evidence only for JHotDraw, see Table 13) were found for comment verbosity.

We also show to what extent class size and comment verbosity correlate with the effort needed to label classes. Table 14 reports results of the Spearman rank correlation between such factors and the labeling effort. The correlation analysis indicates that the *comment verbosity* has a statistically significant negative correlation with the time needed to label the selected classes for both eXVantage and JHotDraw and for both the experiments. Such a result suggests that, when the source code is not well commented, then it is very difficult for a human to identify the responsibilities of a class and thus the keywords to describe them. For what concerns the *class size* factor, we observed that there is a small positive correlation with the time for JHotDraw, while such a factor has a highly positive (and statistically significant) correlation for eXVantage. One factor that can have produced such a correlation is the different verbosity of comments in the two software systems. Indeed, while for eXVantage comments contain an average number of 14 terms for each method, for JHotDraw the average verbosity of comments is of about 6 terms for each method.

Thus, the time needed to understand and describe the classes depends on the *class size* when the code is well commented (i.e., the comments represent an important percentage of the class corpus), while the required time does not depend on the *class size* when the comments are poor or totally absent, i.e. in this case the difficulty tends to be high independently of the class size.

Finally, it is important to note that, although better comments correlate with lower effort, this does not mean that comment words are used to form labels. Indeed, as indicated in Table 5, oracles contain more words belonging from comments in JHotDraw than in eXVantage, despite the latter has comments with higher verbosity.

RQ3 Summary Simple heuristics exhibit good performance independently of the class size and comment verbosity. Instead, IR techniques—above all LSI and LDA—

Table 14 Influence of code size and comment verbosity on the time needed to label the selected classes

Participant	System	Class size	Comment verbosity
Exp. I	JHotDraw	0.13	−0.37
	eXVantage	0.35	−0.14
Exp. II	JHotDraw	0.15	−0.28
	eXVantage	0.43	−0.41

Pearson product-moment correlations

work generally better on a larger corpus, i.e., large classes and classes having a high comment verbosity. Furthermore, LDA and LSI work well on artifacts having a low term entropy, which is quite infrequent for source code artifacts.

4 Threats to Validity

This section describes threats that can affect the validity our study. Threats to *construct validity* concern relationship between the theory and the observation. In our study, threats to construct validity are mainly related to the measurements we performed to address our research questions. To investigate to what extent labelings identified by humans match those identified by automatic approaches (**RQ1**), we rely on a widely used similarity measure, i.e., the Jaccard overlap score. For what concerns the origin of terms used by humans to label software artifacts (**RQ2**), we performed a manual analysis to map the list of terms provided by participants onto source code elements. To avoid mistakes, the analysis was performed by two independent persons (two of the authors), and results were compared and discussed to solve inconsistencies. The time measurements annotated by participants—investigated in **RQ3**—could be affected by imprecision as the experiment was conducted offline: however, we clearly explained to participants to carefully record such a time, and there is no reason they would have cheated, also because the experiment was entirely on a voluntary basis and participants were not evaluated on their performance. Also, the pre-experiment code understanding would have unlikely influenced such a time, since participants did not know exactly which classes were the objects of the study.

Threats to *conclusion validity* concern the relationship between the treatment and the outcome. Wherever appropriate, we use statistical procedures to corroborate our results. Specifically, we use non-parametric tests (Wilcoxon) and correlation procedures (Spearman rank), adjusting p-values using the Holm procedure when performing multiple tests on the same data. In addition, we report Cliff d effect size to provide a quantitative assessment of the differences found. Also, whenever a co-factor analysis is needed, we use permutation tests that, differently from ANOVA, does not require data to be normally distributed.

Threats to *internal validity* are related to factors that can influence our results. In terms of human factors, there was no abandonment, i.e., all participants completed the task. To mitigate the effect of labeling variability across participants, we chose the most frequent words they used to label each class. Another factor that can influence our results could be the choice of the number of topics for LDA, and the number of concepts in LSI. For the former, we used different settings (i.e., number of topics equal to the number of methods, half of them, and two topics only). For the latter, we used the heuristics adopted by Kuhn et al. (2007). Furthermore, we investigate whether other factors—such as the class entropy and the textual similarity among classes—could have influenced the results of the various techniques.

Threats to *external validity* concern the generalization of results. We tried to investigate as many IR-based techniques as possible to perform automatic labeling: VSM, LDA, LSI, and three simple heuristics, considering words form class comments, signatures, and their combinations. Also, we compared different weighting schemes (*tf* and *tf-idf*), and different sources of information (source code including comments, and comments only). However, we are aware that there could be other heuristics

we did not consider. We are also aware that the study involved objects from two Java systems only. Therefore, results could be different if replicating the study on other systems, and in particular on objects developed with different programming languages. For instance it would be interesting to replicate on programs written using programming languages like C where, differently from Java, term separators (e.g., Camel Case) are not consistently used, and also there is a high usage of abbreviations (Guerrouj et al. 2011). Last, but not least, we are aware that our labelings were performed by students. Although we carefully avoided (by means of a proper training) that the limited knowledge of the objects could have influenced the results, we are aware that results could possibly vary if replicating with professionals. For example, it may (or may not) happen that professionals would, in some case, pick other terms than those in the method signatures/class names, thus decreasing the performance of the simple heuristic. As a partial mitigation to this threat, participants employed in Exp. II had more experience than those of Exp. I—also including industrial experience gained during periods of internship.

5 Related Work

The rapid development of software engineering methods and tools and the increasing complexity of software projects has led, in the last decades, to a significant production of textual information contained in structured and unstructured project artifacts. As consequence, several researchers investigated the analysis of textual information contained in the artifacts of software repositories to support activities such as impact analysis (Canfora and Cerulo 2005), clone detection (Marcus and Maletic 2001), feature location (e.g., Poshyvanyk et al. 2007), definition of new cohesion and coupling metrics (e.g., Marcus et al. 2008; Poshyvanyk and Marcus 2006), assessment of software quality (e.g., Lawrie et al. 2007; Takang et al. 1996; De et al. 2011; Binkley et al. 2007), and traceability recovery (e.g., Antoniol et al. 2002; Marcus and Maletic 2003; De Lucia et al. 2007; Hayes et al. 2006; Gethers et al. 2011; Asuncion et al. 2010; Cleland-Huang et al. 2010).

Related to our work is the use of textual analysis, and in particular topic modeling techniques, to mine and understand topics within source code. In the next subsection we discuss approaches aim at providing support to program comprehension by deriving a snapshot of the system that is easier to understand. In addition, we also discuss approaches for the automatic summarization of source code artifacts aiming at aiding developers in comprehension tasks.

5.1 Mining of Topics in Source Code

Maletic and Marcus (2001) proposed the combined use of semantic and structural information of programs to support comprehension tasks. Semantic information, captured by LSI, refers to the domain specific issues (both problem and development domains) of a software system, while structural information refers to issues such as the actual syntactic structure of the program along with the control and data flow that it represents. Components within a software system are then clustered together using the combined similarity measure.

Kuhn et al. (2007) extended the work by Maletic and Marcus introducing the concept of semantic clustering, a technique based on LSI to group source code documents that share a similar vocabulary. After applying LSI to the source code, the documents are clustered based on their similarity into semantic clusters, resulting in clusters of documents that implement similar features. The authors also used LSI to label the identified clusters. Finally, a visual notation is provided, which is aimed at giving an overview of all the clusters and their semantic relationships.

Baldi et al. (2008) applied LDA to source code to automatically identify concerns. In particular, they used LDA to identify topics in the source code. Then, they used the entropies of the underlying topic-over-files and files-over-topics distributions to measure software scattering and tangling. Candidate concerns are latent topics with high scattering entropy.

Linstead et al. (2008) used LDA to identify functional components of source code and study their evolution over multiple project versions. The results of a reported case study highlight the effectiveness of probabilistic topic models in automatically summarizing the temporal dynamics of software concerns.

Thomas et al. (2011) also applied LDA to the history of the source code of a project to recover its topic evolutions. The authors considered additional topic metrics (i.e., scatter and focus) to better understand topic change events, and providing a detailed, manual analysis of the topic change events to validate the results of the approach.

Generally, works on topic analysis produced project-specific topics that needed to be manually labeled. Hindle et al. (2011) presented a cross-project data mining technique leveraging software engineering standards to produce a method of partially-automated (supervised) and fully-automated (semi-unsupervised) topic labeling. Since the proposed approach is not project-specific, it is possible to use it to compare two distinct projects.

Hindle et al. (2012) used LDA in an industrial context to relate requirements to code. They performed an empirical study in order to verify whether the information extracted with LDA matches the perception that program managers and developers have about the effort put into addressing certain topics. The results indicated that in general the identified topics made sense to practitioners and matched their perception of what occurred even if in some particular cases practitioners had difficulty interpreting and labeling the extracted topics.

Recently, Medini et al. (2012) used information retrieval methods and formal concept analysis to produce sets of words helping the developer to understand the concept implemented in execution traces. The authors performed both a qualitative as well as a quantitative analysis of the proposed approach. The analysis revealed that the approach is quite accurate in identifying topics in execution traces and in most cases the suggested labeling terms are effective to help grasping the segment functionality.

5.2 Summarization of Source Code

Besides topic analysis, summarization techniques have also been applied to source code artifacts for different purposes. Rastakar et al. used a machine learning approach to automatically generate summaries of bug reports (Rastkar et al. 2010) and software concerns (Rastkar 2010). Buse and Weimer proposed an approach to au-

tomatically generate human-readable documentation for arbitrary code differences (Buse and Weimer 2010).

Murphy (1996) presented the software reflection model and the lexical source model extraction. Such models can be considered as a lightweight summarization approach of software. Sridhara et al. used natural language processing techniques to automatically generate leading method comments (Sridhara et al. 2010), and comments for high-level actions (Sridhara et al. 2011). Also the automatic generation of comments can be considered as a kind of summarization of source code components.

In summary, several of the works described above used different techniques to label or summarize software artifacts. Our study constitutes a complementary contribution to such approaches, because it aims at assessing automatic labeling techniques by comparing them with human-generated labels.

Haiduc et al. (2010a, b) recently applied several summarization techniques for the automatic summarization of source code artifacts with the purpose of aiding developers in comprehension tasks. In a reported case study (Haiduc et al. 2010b) they found that a combination between techniques making use of the position of terms in software and Textual Retrieval (TR) techniques capture the meaning of methods and classes better than any other of the studied approaches. In addition, an experiment conducted with four developers revealed that the summaries produced using this combination make sense.

To the best of our knowledge, the work by Haiduc et al. (2010b) is the most relevant to our work. However, while Haiduc et al. asked developers to validate the summaries, we compare the labelings obtained with automatic techniques with humans' labels. This provides an objective and more precise evaluation of the accuracy of automated techniques in approximating the cognitive model of developers. In addition, we used a larger and different set of techniques based on advanced IR methods and also on "ad hoc" heuristics. Finally, we involved in our experimentation a larger number of subjects (37 *vs.* 4) having different experience (both undergraduate and graduate students), showing that results are almost perfectly consistent between the two experiments. In terms of results, while we share with Haiduc et al. (2010b) findings about the importance of class-level comments for artifact labeling, our results also highlight that: (i) comments alone do not produce good labels, because they contain several words that were discarded by humans. Instead, comments are useful when combined with class signatures; and (ii) simpler heuristics considering class signatures (possibly combined with comments) outperform IR techniques.

6 Conclusion and Future Work

In recent years, researchers have applied various IR methods to "label" software artifacts by means of some representative words, with the aim of facilitating their comprehension or just to better visualize them. This paper reported an empirical study aimed at investigating to what extent a source code labeling based on IR techniques would identify relevant words in the source code, compared to the words a human developer would have selected during a program comprehension task.

We conducted two experiments, in which we asked 17 Bachelor's Students and 21 Master's Students, respectively, to describe 20 classes taken from two software

systems using at most ten words extracted from the class source code and comments. Then, we analyzed (i) what kind of source code (and comment) elements were used by subjects to produce the labels; (ii) to what extent the keywords identified using various IR techniques overlap with those identified by humans; and (iii) what characteristics of the analyzed artifacts could influence the effectiveness of the various techniques used to automatically produce labels. As possible techniques, we considered VSM, LSI, LDA, and some *ad hoc* heuristics picking terms from specific parts of the source code and comments.

Results show that overall there is a relatively high overlap between automatic and human-generated labels, ranging between 50 % and 90 %. However, the highest overlap is obtained by using the simplest heuristic, while the most sophisticated techniques, i.e., LSI and LDA, provide generally the worst accuracy. One reason of the result is that developers mainly used words from class names, method names and signatures, and (partially) from class and method comments to label artifacts. Therefore, the remaining textual corpus (considered by IR techniques such as VSM, LSI, and LDA) tends to add more noise than useful information. We also found that the high entropy of terms in the classes inhibits the capability of topic modeling techniques—i.e., LSI and LDA—to efficiently identify and cluster topics in source code. This result highlights that approaches such as LDA and LSI are worthwhile of being used when analyzing heterogeneous collections, where documents can contain information about multiple topics (Lavrenko 2009). Unfortunately, such an heterogeneity is not always present in source code artifacts. Thus, for labeling source code artifacts, *ad-hoc* heuristics better reflect keywords identified by developers.

Future work will aim at replicating the study on other source code artifacts, possibly developed using different programming languages, and considering other groups of subjects, e.g., contributors of open source projects. We also plan to experiment and compare other IR methods (e.g., RTM and summarization methods), as well as other heuristics.

Acknowledgements We would like to thank all the students that participated in our study. We would also like to thank anonymous reviewers for their careful reading of our manuscript and high-quality feedback. Their detailed comments have helped us to improve the original version of this paper.

References

Antoniol G, Canfora G, Casazza G, De Lucia A, Merlo E (2002) Recovering traceability links between code and documentation. IEEE Trans Softw Eng 28(10):970–983

Asuncion HU, Asuncion A, Taylor RN (2010) Software traceability with topic modeling. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering. ACM Press, Cape Town, South Africa, pp 95–104

Baeza-Yates R, Ribeiro-Neto B (1999) Modern information retrieval. Addison-Wesley

Baker RD (1995) Modern permutation test software. In: Edgington E (ed) Randomization tests. Marcel Decker

Baldi P, Lopes CV, Linstead E, Bajracharya SK (2008) A theory of aspects as latent topics. In: Proceedings of the 23rd annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications. ACM Press, Nashville, TN, USA, pp 543–562

Binkley D, Feild H, Lawrie D, Pighin M (2007) Software fault prediction using language processing. In: Proceedings of the testing: academic and industrial conference practice and research techniques. IEEE Computer Society, pp 99–110

Blei DM, Ng AY, Jordan MI (2003) Latent dirichlet allocation. J Mach Learn Res 3:993–1022

Buse RPL, Weimer W (2010) Automatically documenting program changes. In: Proceedings of the 25th IEEE/ACM international conference on automated software engineering. ACM Press, Antwerp, Belgium, pp 33–42
Canfora G, Cerulo L (2005) Impact analysis by mining software and change request repositories. In: Proceedings of 11th IEEE international symposium on software metrics. IEEE CS Press, Como, Italy, pp 20–29
Chang J, Blei DM (2010) Hierarchical relational models for document networks. Ann Appl Stat 4(1):124–150
Cleland-Huang J, Czauderna A, Gibiec M, Emenecker J (2010) A machine learning approach for tracing regulatory codes to product specific requirements. In: Proc. of ICSE, pp 155–164
Cullum JK, Willoughby RA (1998) Lanczos algorithms for large symmetric eigenvalue computations, vol 1, chapter Real rectangular matrices. Birkhauser, Boston
De Lucia A, Di Penta M, Oliveto R (2011) Improving source code lexicon via traceability and information retrieval. IEEE Trans Softw Eng 2(37):205–227
De Lucia A, Fasano F, Oliveto R, Tortora G (2007) Recovering traceability links in software artefact management systems using information retrieval methods. ACM Trans Soft Eng Methodol 16(4), article no. 13
Deerwester S, Dumais ST, Furnas GW, Landauer TK, Harshman R (1990) Indexing by latent semantic analysis. J Am Soc Inf Sci 41(6):391–407
Detienne F (2002) Software design: cognitive aspects. Springer Verlag
Gethers M, Oliveto R, Poshyvanyk D, De Lucia A (2011) On integrating orthogonal information retrieval methods to improve traceability recovery. In: Proceedings of the 27th international conference on software maintenance. IEEE Press, Williamsburg, USA, pp 133–142
Gethers M, Savage T, Di Penta M, Oliveto R, Poshyvanyk D, De Lucia A (2011) Codetopics: which topic am i coding now? In: Proceedings of the 33rd International conference on software engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, 21–28 May 2011. ACM, pp 1034–1036
Grissom RJ, Kim JJ (2005) Effect sizes for research: a broad practical approach, 2nd edn. Lawrence Earlbaum Associates
Guerrouj L, Di Penta M, Antoniol G, Guèhèneuc Y-G (2011) TIDIER: an identifier splitting approach using speech recognition techniques. J Softw Evol Process 25(6):575–599
Haiduc S, Aponte J, Marcus A (2010) Supporting program comprehension with source code summarization. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering. ACM Press, Cape Town, South Africa, pp 223–226
Haiduc S, Aponte J, Moreno L, Marcus A (2010) On the use of automated text summarization techniques for summarizing source code. In: Proceedings of the 17th working conference on reverse engineering. IEEE Computer Society, Beverly, MA, USA, pp 35–44
Hayes JH, Dekhtyar A, Sundaram SK (2006) Advancing candidate link generation for requirements tracing: The study of methods. IEEE Trans Softw Eng 32(1):4–19
Hindle A, Bird C, Zimmermann T, Nagappan N (2012) Relating requirements to implementation via topic analysis: Do topics extracted from requirements make sense to managers and developers? In Proceedings of the 28th international conference on software maintenance. IEEE CS Press, Riva del Garda, Italy
Hindle A, Ernst NA, Godfrey MW, Mylopoulos J (2011) Automated topic naming to support cross-project analysis of software maintenance activities. In: Proceedings of the 8th international working conference on mining software repositories. IEEE CS Press, Waikiki, Honolulu, USA, pp 163–172
Holm S (1979) A simple sequentially rejective Bonferroni test procedure. Scand J Stat 6:65–70
Ko AJ, Myers BA, Coblenz MJ, Aung HH (2006) An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Trans Softw Eng 32(12):971–987
Kuhn A, Ducasse S, Gîrba T (2007) Semantic clustering: Identifying topics in source code. Inf Softw Technol 49(3):230–243
Kuhn A, Ducasse S, Gîrba T (2007) Semantic clustering: Identifying topics in source code. Inf Softw Technol 49(3):230–243
LaToza TD, Venolia G, DeLine R (2006) Maintaining mental models: a study of developer work habits. In: Proceedings of the 28th international conference on software engineering. ACM Press, Shanghai, China, pp 492–501
Lavrenko V (2009) A generative theory of relevance, vol 26. Springer
Lawrie D, Feild H, Binkley D (2007) An empirical study of rules for well-formed identifiers. J Softw Maint 19(4):205–229

Liblit B, Begel A, Sweetser E (2006) Cognitive perspectives on the role of naming in computer programs. In: Proceedings of the 18th annual workshop on psychology of programming. University of Sussex, Brighton, UK

Linstead E, Lopes CV, Baldi P (2008) An application of latent dirichlet allocation to analyzing software evolution. In: Proceedings of the 7th international conference on machine learning and applications. IEEE CS Press, San Diego, California, USA, pp 813–818

Liu Y, Poshyvanyk D, Ferenc R, Gyimóthy T, Chrisochoides N (2009) Modeling class cohesion as mixtures of latent topics. In: Proc. of ICSM, pp 233–242

Maletic JI, Marcus A (2001) Supporting program comprehension using semantic and structural information. In: Proceedings of 23rd international conference on software engineering. IEEE CS Press, Toronto, Ontario, Canada, pp 103–112

Marcus A, Maletic JI (2001) Identification of high-level concept clones in source code. In: Proceedings of 16th IEEE international conference on automated software engineering. IEEE CS Press, San Diego, California, USA, pp 107–114

Marcus A, Maletic JI (2003) Recovering documentation-to-source-code traceability links using latent semantic indexing. In: Proceedings of 25th international conference on software engineering. IEEE CS Press, Portland, Oregon, USA, pp 125–135

Marcus A, Poshyvanyk D (2005) The conceptual cohesion of classes. In: Proceedings of 21st IEEE international conference on software maintenance. IEEE CS Press, Budapest, Hungary, pp 133–142

Marcus A, Poshyvanyk D, Ferenc R (2008) Using the conceptual cohesion of classes for fault prediction in object-oriented systems. IEEE Trans Softw Eng 34(2):287–300

Medini S, Antoniol G, Guéhéneuc Y-G, Di Penta M, Tonella P (2012) Scan: an approach to label and relate execution trace segments. In: Proceedings of the 19th working conference on reverse engineering. IEEE Press, Kingston, Ontario, Canada

Murphy G (1996) Lightweight structural summarization as an aid to software evolution. PhD thesis, University of Washington

Porteous I, Newman D, Ihler A, Asuncion A, Smyth P, Welling M (2008) Fast collapsed gibbs sampling for latent dirichlet allocation. In: Proceedings of the 14th ACM SIGKDD international conference on knowledge discovery and data mining. ACM, New York, NY, USA, pp 569–577

Porter MF (1980) An algorithm for suffix stripping. Program 14(3):130–137

Poshyvanyk D, Gael-Gueheneuc Y, Marcus A, Antoniol G, Rajlich V (2007) Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE Trans Softw Eng 33(6):420–432

Poshyvanyk D, Marcus A (2006) The conceptual coupling metrics for object-oriented systems. In: Proceedings of 22nd IEEE international conference on software maintenance. IEEE CS Press, Philadelphia, PA, USA, pp 469–478

Rajlich V, Wilde N (2002) The role of concepts in program comprehension. In: Proceedings of the 10th international workshop on program comprehension. IEEE Computer Society, Paris, France, pp 271–280

Rastkar S (2010) Summarizing software concerns. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering – student research competition. ACM Press, Cape Town, South Africa, pp 527–528

Rastkar S, Murphy GC, Murray G (2010) Summarizing software artifacts: a case study of bug reports. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering. ACM Press, Cape Town, South Africa, pp 505–514

Shannon CE (1948) A mathematical theory of communication. Bell Syst Tech J 27:379–423, 625–56

Sridhara G, Hill E, Muppaneni D, LL Pollock, Vijay-Shanker K (2010) Towards automatically generating summary comments for java methods. In: Proceedings of the 25th IEEE/ACM international conference on automated software engineering. ACM Press, Antwerp, Belgium, pp 43–52

Sridhara G, Pollock LL, Vijay-Shanker K (2011) Automatically detecting and describing high level actions within methods. In Proceedings of the 33rd International conference on software engineering. ACM Press, Honolulu, HI, USA, pp 101–110

Srivastava A, Sahami M (2009) Text mining: classification, clustering, and applications. Chapman & Hall/CRC

Storey M-AD (2006) Theories, tools and research methods in program comprehension: past, present and future. SQJ 14(3):187–208

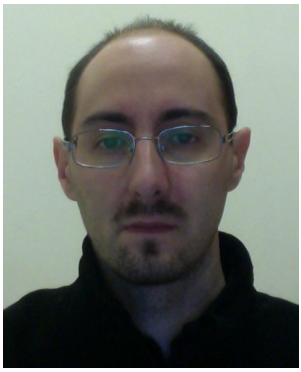
Takang A, Grubb P, Macredie R (1996) The effects of comments and identifier names on program comprehensibility: an experiential study. J Program Lang 4(3):143–167

Teh YW, Newman D, Welling M (2006) A collapsed variational bayesian inference algorithm for latent dirichlet allocation. In: NIPS, pp 1353–1360

Thomas SW, Adams B, Hassan AE, Blostein D (2010) Validating the use of topic models for software evolution. In: Tenth IEEE international working conference on source code analysis and manipulation, SCAM 2010. IEEE Computer Society, Timisoara, Romania, 12–13 Sept 2010, pp 55–64

Thomas SW, Adams B, Hassan AE, Blostein D (2011) Modeling the evolution of topics in source code histories. In: Proceedings of the 8th international working conference on mining software repositories. IEEE Press, Honolulu, HI, USA, pp 173–182

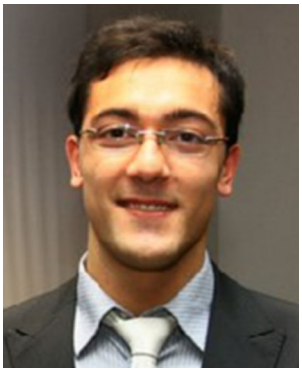
Andrea De Lucia is a full professor of software engineering at the Department of Management & Information Technology of the University of Salerno, Italy, head of the Software Engineering Lab, and Director of the International Summer School on Software Engineering. From 1996 to 2003 he was with the Department of Engineering and the Research Centre on Software Technology (RCOST) at the University of Sannio, Italy. He received his PhD in Electronic Engineering and Computer Science from the University of Naples "Federico II", Italy, in 1996. Prof. De Lucia has coordinated several research and technology transfer projects most of them conducted in cooperation with industrial partners. His research interests include software maintenance and testing, program comprehension, reverse engineering, reengineering, migration, global software engineering, software configuration management, empirical software engineering, sarch-based software engineering, software defect prediction, software cost estimation, workflow management, document management, visual languages, web engineering, and e-learning. He has published more than 200 papers on these topics in international journals, books, and conference proceedings. He has also edited books and special issues of international journals and serves on the editorial boards of international journals, including Empirical Software Engineering (published by Springer) and Journal of Software: Evolution and Process (published by Wiley). He has also been serving on the organizing and program committees of several international conferences, including the IEEE/ACM International Conference on Software Engineering, the IEEE International Conference on Software Maintenance, the Working Conference on Reverse Engineering, the IEEE Conference on Program Comprehension, and the IEEE Working Conference on Source Code Analysis and Manipulation. Prof. De Lucia is a senior member of the IEEE and the IEEE Computer Society. He was also at-large member of the executive committee of the IEEE Technical Council on Software Engineering (TCSE) and committee member of the IEEE Real World Engineering Project (RWEP) Program.



Massimiliano Di Penta is associate professor at the University of Sannio, Italy. His research interests include software maintenance and evolution, reverse engineering, empirical software engineering, search-based software engineering, and service-centric software engineering. He is author of over 170 papers appeared in international conferences and journals. He serves and has served in the organizing and program committees of over 60 conferences such as ICSE, FSE, ASE, ICSM, ICPC, CSMR, GECCO, MSR, SCAM, WCRE, and others. He has been general chair of SCAM 2010, WSE 2008, general co-chair of SSBSE 2010, WCRE 2008, and program co-chair of MSR 2013 and 2012, ICPC 2013, ICSM 2012, SSBSE 2009, WCRE 2006 and 2007, IWPSE 2007, WSE 2007, SCAM 2006, STEP 2005, and of other workshops. He is steering committee member of ICSM, CSMR, IWPSE, SSBSE, PROMISE, and past steering committee member of ICPC, SCAM, and WCRE. He is in the editorial board of the Empirical Software Engineering Journal edited by Springer, and of the Journal of Software: Evolution and Processes edited by Wiley. He is member of IEEE, IEEE Computer Society, and of the ACM. Further info on www.rcost.unisannio.it/mdipenta.

Rocco Oliveto is Assistant Professor in the Department of Bioscience and Territory at University of Molise (Italy). He is the Director of the Laboratory of Informatics and Computational Science of the University of Molise. He received the PhD in Computer Science from University of Salerno (Italy) in 2008. From 2008 to 2010 he was research fellow at the Department of Mathematics and Informatics of University of Salerno. From 2005 to 2010 he is also adjunct professor at the Faculty of Science of University of Molise (Italy). In 2011 he joined the STAT Department of University of Molise. His research interests include traceability management, information retrieval, software maintenance and evolution, search-based software engineering, and empirical software engineering. He has published more than 50 papers on these topics in international journals, books, and conference proceedings. He serves and has served as organizing and program committee member of international conferences in the field of software engineering. In particular, he was the program co-chair of TEFSE 2009, the Traceability Challenge Chair of TEFSE 2011, the Industrial Track Chair of WCRE 2011, the Tool

Demo Co-chair of ICSM 2011, the program co-chair of WCRE 2012 and WCRE 2013, and he will be the program co-chair of SCAM 2014. Dr. Oliveto is member of IEEE Computer Society, ACM, and IEEE-CS Awards and Recognition Committee.



Annibale Panichella was born in Isernia (Italy). He received (cum laude) the Laurea in Computer Science from the University of Salerno (Italy) in 2010 defending a thesis on Software Project Effort Estimation, advised by Dr. Carmine Gravino and Dr. Rocco Oliveto. He is currently a PhD student at the Department of Mathematics and Informatics of the University of Salerno under the supervision of Prof. Andrea De Lucia and Dr. Rocco Oliveto. His research interests include traceability management, information retrieval, empirical software engineering, search based software engineering, software testing, evolutionary computation.

Sebastiano Panichella was born in Isernia (Italy). He received (cum laude) the Laurea in Computer Science from the University of Salerno (Italy) in 2010 defending a thesis on IR-based Traceability Recovery, advised by Prof. Andrea De Lucia and Dr. Rocco Oliveto. He is currently a PhD Candidate at the Department of Engineering, University of Sannio - Benevento (Italy) under the supervision of Prof. Gerardo Canfora and Prof. Massimiliano Di Penta. His research interests include Mining Software Repositories, IR-based Traceability Recovery, Textual Analysis, Software Maintenance and Evolution and Empirical Software Engineering. He is author of thirteen papers appeared in International Conference and Journals.