Abstract—Information Retrieval (IR) techniques have been used for various software engineering tasks, including the labeling of software artifacts by extracting “keywords” from them. Such techniques include Vector Space Models, Latent Semantic Indexing, Latent Dirichlet Allocation, and also of providing complementary information to what traditional code analysis—e.g., the extraction of structural information—could provide [15], [8].


I. INTRODUCTION

Source code lexicon (identifier names and comments) plays an essential role in program comprehension, especially when (i) the high-level documentation is scarce or outdated, or when (ii) the source code is complex enough such that the lexicon would tell more to developers than what the code semantics would do.

In past and recent years, several researchers analyzed the role of lexicon in program comprehension and how it would influence code quality [1], [2]. Furthermore, software lexicon has been used—as an alternative or as a complement to source code structure—to perform various kinds of analyses, for example traceability recovery (e.g., [3]), change impact analysis [4], clone detection [5], feature location (e.g., [6]), cohesion and coupling computation (e.g., [7], [8]), software quality assessment [9], and fault prediction [10].

All these approaches rely on Information Retrieval (IR) techniques, such as Vector Space Models [11], Latent Semantic Indexing (LSI) [12], Latent Dirichlet Allocation (LDA) [13], or Relational Topic Models (RTM) [14]. Textual analysis of source code has the advantage of being lightweight (as it does not require parsing), and also of providing complementary information to what traditional code analysis—e.g., the extraction of structural information—could provide [15], [8].

In summary, IR techniques represent source code artifacts by assigning a weight or a probability to the words they contain. Recently many researchers have also applied IR techniques to automatically “label” software artifacts by means of some representative words (i.e., words having the highest weight or probability). For example, Kuhn et al. [16] used discriminant words from LSI concepts to label software packages; Thomas et al. [17] used LDA to label source code changes; Gethers et al. [18] used RTM to identify and relate topics in high-level artifact and source code.

This paper investigates to what extent an IR-based source code labeling would identify relevant words in the source code, compared to the words a human would manually select during a program comprehension task. Specifically, it aims at answering the following research question:

To what extent terms selected by developers to describe a source code artifact overlap with those that can be identified by an automated technique?

The main motivation of the study is to investigate whether such techniques are worthwhile of being used for labeling purposes, when they work well, and when not. To answer the above research question, we conducted an experiment in which we asked 17 subjects—selected among the Bachelor’s Students in Computer Science of the University of Molise—to describe classes taken from two software systems—JHotDraw [19] and eXVantage [20]—using at most ten words extracted from the source code and comments. Then, we analyzed to what extent the highest weighted words indexed using various IR techniques—i.e., VSMs, LSI, LDA, and a simple heuristic picking terms from class, attribute and method names only—overlap with those identified by humans.

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 40% and 80%—with the manually-generated labels. The highest overlap is obtained by using the heuristic considering only terms extracted from class name, method signatures and attribute names. LSI and LDA (which are based on artifacts clustering) generally provide the worst overlap. In particular, we observed that the high entropy of terms contained in the source code artifacts inhibits the capability of such 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 [19]. 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 analyzed the time required by subjects to identify the keywords that characterize the responsibilities of a class, to identify factors that could influence such an effort. We observed that, the higher the verbosity of the comments the lower the time required by subjects to identify the keywords. This result confirms the importance of having comments in comprehension tasks [1], [2]. In addition, we also observed that clustering-based approaches (i.e., LSI and LDA) are much more worthwhile to be used on source code artifacts having a high verbosity and that required more effort to be manually labeled.

The paper is organized as follows. Section II describes the empirical study definition and planning. Section III reports and discusses the results. Section IV discusses the threats to validity that could affect our study. Section V overviews related work, while Section VI concludes the paper and outlines directions for future work.

II. STUDY DEFINITION AND PLANNING

The goal of our study is to compare human-generated source code labeling with automatically generated ones. The quality focus concerns the quality of automatically generated source code labelings, measured as their overlap with the human-generated labels. The perspective is of researchers interested in understanding to what extent automatic source code labeling based on IR methods can be used, and in which circumstance each technique performs well or not.

A. Study Context and Research Questions

The context of our study consists of objects, i.e., classes extracted from two Java software systems, and subjects, i.e., undergraduate students from the University of Molise, Italy. The object systems used in our study are JHotDraw and eXVantage. The former is an open source UML modeling tool and the version 6.0 b1 used in our study is composed of 275 classes (29 KLOC). The latter is a novel testing and generation tool. It is an industrial system and the version used in our study is composed of 348 classes (28 KLOC). In the context of our study, we selected ten classes from JHotDraw and ten from eXVantage.

Regarding the subjects involved in our study, they are 17 2nd year Bachelor’s students attending the Software Engineering course. All subjects were from the same class with 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 the context of our study we detailed the overall research question mentioned in the introduction into the two following sub-questions:

- **RQ1**: How much is the overlap between the keywords identified by developers when describing a source code artifact and those identified by an automatic technique?

This research question aims at quantifying the performance of an automatic technique when used to identify the keywords that can be used to characterize a source code artifact. In presence of a high overlap, it can be argued that the automatic technique is able to approximate the mental model used by developers to extract keywords when comprehending a source code artifact.

- **RQ2**: Which are the characteristics of source code artifacts that affect the overlap of automatic labeling techniques with the human-generated labels? Our second research question aims at identifying specific characteristics of source code artifacts (such as verbosity or comment density) that could inhibit the capability of the automated technique to approximate the mental model of developers when comprehending a source code artifact. Such an analysis could also indicate techniques that are more useful for particular source code artifacts.

B. Study Procedure

The study was organized in three steps. In the first step, we asked developers to describe the selected source code classes with a set of 10 keywords. Then, we applied different techniques to automatically extract keywords from the selected classes. To this aim, we used three different IR techniques and a simple heuristic extracting words from specific source code elements. Once having the set of keywords identified by the developers and the set of keywords identified by the experimented technique, we compare them in order to compute the overlap and try to answer our research questions. In the next subsections we provide details on each of these steps.

1) Step 1: Human Labeling of Software Artifacts: Before asking subjects to label software artifacts, we made them familiar with the study objects. We provided to subjects access to both systems a month before sending the questionnaire. In addition, meetings made by students together with system experts were scheduled once a week with the aim to enrich the students’ domain knowledge, and one of the authors (instructor of the course) also participated to the meeting to check the learning progress. Then, we presented the experimental procedure to be followed, so that each subject knew exactly the sequence of steps to perform. However, to avoid any bias, we made sure subjects were not aware of the study research questions.

   The experiment was executed offline, i.e., we sent the experimental material to the subjects. We chose such an alternative to allow subjects having more time to perform the comprehension tasks. Also, a fully-controlled (online) setting is not strictly required in this case, as we are not interested to evaluate the subjects’ performances. Rather, we are interested in collecting human-generated artifact labelings to be able to compare them with the automatic generated ones.

   We sent to each subject 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 subject’s demographics, i.e., years of computer science schooling and years of programming experience with Java. We also asked the subjects to provide their confidence level with Java on a Likert scale of 1 to 5, where 1 indicates low confidence and 5 high confidence.

The other ten sheets aimed at collecting the keywords for each of the classes to be analyzed. In each sheet we reported the full class name and asked the subject 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 subjects to provide for each class the time spent to identify the keywords and rate the difficulty encountered to identify them on a scale of 1 to 5, where 1 indicates low difficulty and 5 high difficulty.

Subjects had two weeks to fill-in the questionnaire. 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 subjects to describe $C_i$. For each term $t_j \in T_{C_i}$, we computed its level of agreement ($LoA$) as follows:

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

where $f_{ij}$ represents the frequency of the term $t_j$, i.e., the number of subjects that used $t_j$ to label $C_i$, and $ns$ represents the number of subjects involved in our study, i.e., 17. The terms having a $LoA$ higher than 50% (i.e., the terms have been selected by at least half of the subjects) represent the set of keywords ($K_{C_i}$) identified by subjects to label the class $C_i$.

2) Step 2: Automatic labeling software artifacts: In the second step of our study, we automatically identify 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. In addition, we used a simple heuristic, considering only terms from class name, signature of methods, and attribute names.

VSM [11] 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 $u_{i,j}$ of this matrix denotes a measure of the weight (i.e., relevance) of the $i^{th}$ term in the $j^{th}$ document [11].

LSI [12] is an extension of the VSM. It was developed to overcome the synonymy and polysemy problems, which occur with the VSM model [12]. 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) [20] 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) [13] allows to fit 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 as topics. Basically, a document can be considered as a probability distribution of topics—fitting the Dirichlet prior distribution—and each topic consists in a distribution of words that, in some sense, represent the topic.

In order 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, morphological analysis of the extracted words was performed to bring back words to the same root, e.g., by removing plurals to nouns, verb conjugations. The simplest way to do morphological analysis is by using a stemmer, e.g., the Porter stemmer [21]. In the context of our study, we considered words from source code including comments and 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:

$$tf_{i,j} = \frac{r_{fi,j}}{\sum_{k=1}^m r_{fk,j}}$$

where $r_{fi,j}$ 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 $tf - idf_{i,j} = tf_{i,j} \cdot idf_i$, where $tf_{i,j}$ is the term frequency defined above and $idf_i$ (inverse document frequency) is defined as:

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

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, after weighting words using $tf$ or $tf-idf$, we just select, for each class, the $h$ words (we choose $h = 10$) having the highest weights.

As for LSI, we apply it on each single artifact (i.e., class) rather than on the corpus composed of all the classes of the system. This is done by considering the textual corpus
composing the body of each method as a document in the document-by-term space, then projecting the document-by-term space into a document-by-concept space, reducing the number of concepts to $k$. This is 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)\(^3\).

For the choice of $k$ we used the heuristic proposed by Kuhn et al. [16] that provided good results when labeling source code artifacts, i.e., $k = n \cdot m^{0.2}$. After that, we multiply 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 obtained the matrix $A_k$, we extracted the $h$ words having the highest weights in the LSI space (i.e., $A_k$).

LDA is applied similarly to 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. A crucial issue in the application of LDA is choosing the number of topics. We start 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. [18]), then we reduce it to half the number of methods, and finally we consider the extreme case of two topics only.

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

- **core topic**: the class is labelled by the $h$ words of the topic having the highest probability in the obtained topic distribution;
- **core words**: we considered all the words characterizing the extracted topics and ranked them 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.

The last automatic labeling technique used in our study is a heuristic 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 considering only words composing (i) the class name, (ii) the signature of methods, and (iii) the attribute names. We select the most representative words by ranking them using their tf or 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.

3) **Addressing RQ1: Computing the Overlap between human- and automatic-generated labelings**: To address RQ1, once obtained the keywords identified by the subjects and the keywords identified by the experimented technique, we determined to what extent keywords identified by subjects

\(^3\)When applying LSI, a generic entry $(i, j)$ of the term-by-document matrix that is zero before the application of SVD (indicating that the term $i$ does not occur in the document $j$), can assume a value different to 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 $j$)

correspond to those generated by automatic techniques. To this aim, we compute the overlap between them using an asymmetric Jaccard overlap [11]. Formally, let $K(C_i) = \{t_1 \ldots t_m\}$ and $K_{m_i}(C_i) = \{t_1 \ldots t_h\}$ the sets of keywords identified by subjects and the technique $m_i$, respectively, to label the class $C_i$. The overlap is computed as follows:

$$\text{overlap}_{m_i}(C_i) = \frac{|K(C_i) \cap K_{m_i}(C_i)|}{K_{m_i}(C_i)}$$

It is worth noting that the size of $K(C_i)$ might be different from 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 subjects could be more or less than 10 (depending on the level of agreement). For this reason, we decided to use an asymmetric Jaccard to not penalize too much an automatic method when the size of $K(C_i)$ is higher than 10.

4) **Addressing RQ2: Factors Influencing the Accuracy of Automatic Labeling**: To address RQ2, we analyze whether various characteristics of the source code artifacts could influence the overlap. First, we evaluate whether techniques that perform clustering on software artifacts, thus shifting from the space of terms towards the space of concepts (LSI) or identifying topic distributions in documents (LDA) can be affected by the specific characteristics of the analyzed documents. To evaluate whether the difficulty of LSI and LDA for identifying the most important topic and applying consistent labeling on code classes can be due to their linguistic peculiarity, we measured the entropy for distribution of terms for each class. Formally, let $t = \{t_1, \ldots, t_m\}$ be the terms extracted from the class $C_i$. We can compute the entropy for class $C_i$ as follows:

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

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

Another factor we are interested to investigate is the difficulty for subjects to label a source code artifact. During the experiment, we collect the time spent by subjects to identify the keywords for each class. We therefore analyze whether the overlap between human- and automatic generated labelings can change for classes where subjects took more time. In addition, we further investigate whether such a time variation depends on:

- **class size**, measured as the lines of code; and
- **comment verbosity**, measured as the average number of words for each method’s comment.

To this aim, we used the Pearson product-moment correlation [22].

### III. Analysis of the Results

This section discusses the results of our experiments aiming at answering the research questions formulated in Section II-A.
A. RQ1: Overlap between Humans and Automatic Techniques

Figure 1 shows the average overlap between the human labeling and the automatic labeling obtained (i) using different IR methods (VSM, LSI, and LDA) and (ii) the heuristic that extracts words from specific source code elements. The analysis reveals that the overlap achieved by all the experimented methods for automatic labeling varies between 40% and 80%.

A more detailed analysis of the results indicates that LSI and LDA achieve the worst overlap as compared to VSM (which does not reduce the term space into a topic/concept space as LSI or LDA do) and the ad-hoc heuristic. In particular, the different variants of LSI and LDA achieved an average overlap with human labeling varies between 40% and 65%. VSM allows to obtain an average overlap 4% higher that obtained by LSI and LDA, while the simple heuristic obtains the best overlap (always higher than 70%). Such a result confirms our theoretical conjecture, i.e., that the words a developer would use to describe a class are those composing the class name itself and the signature of its methods.

Figure 1 also shows that the contribution of comments for class labeling varies over the two software projects. For eXVantage, indexing only comments and using LDA allows to obtain an higher overlap with manual labeling than considering source code also. For JHotDraw, we obtain different results, i.e., the best overlap is always achieved when considering the whole source code corpus (code plus comments). Such a different result can be due to the quality of identifiers and to the different verbosity of comments over the two software systems. JHotDraw identifiers are very good and appropriate for labeling (as the system was conceived for pedagogical purposes, i.e., explaining design patterns). In eXVantage the identifiers are generally less meaningful than those used in JHotDraw and developers used very often abbreviated identifiers that are not appropriate for labeling. However, in eXVantage the bad quality of the identifiers is compensated by good comments. In particular, while for eXVantage comments contains an average number of 14 terms for each method, for JHotDraw the average verbosity of comments is around 6 terms for each method.

The achieved results suggest that the different elements of a class (e.g., class name, method signatures, local variables, and comments) play different roles during the automatic labeling of source code. Basically, the most important words are included in the class name, in the method signature, and in the attribute names, i.e., in the high-level structure of the class. Comments also play an important role, but only when they have a good verbosity. Finally, the local variables seem to bring only a poor contribution when labeling source code.

B. RQ2: Factors Influencing the Accuracy of Automatic Labeling

Figure 1 indicates that LSI and LDA provide the worst overlap as compared to the other techniques. This could be due to the fact that they are techniques based on clustering. Indeed, both LSI and LDA are often used for such purposes (see for example the work of Kuhn et al. [16]). Even if such techniques are an appealing solution for labeling and clustering documents, its outcome depends on several strong assumptions [23]. Such assumptions are usually true in traditional IR contexts, while do not necessarily apply for source code. Specifically, approaches based on clustering were designed for analyzing heterogeneous collections, where documents can contain information about multiple topics, while they are pretty different from each other [19]. This means that documents likely contain dominating terms (terms having an higher frequency as compared to the others) that can be used to characterize and distinguish the topics discussed in the documents.

In source code artifacts, heterogeneity is not always present, especially when considering single classes. 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. This means that when analyzing the
textual content of classes it is difficult to identify dominating terms that characterize the topics discussed in the class. To verify such a conjecture, we computed the entropy of the terms contained in the classes used in our study (see Section II-B4). High entropy indicates that the probability distribution of the terms (based on their frequency) is quite uniform, limiting the ability of clustering-based techniques to identify dominating terms that can be used to characterize the class topics.

Figure 2 shows boxplots of the entropy of the terms computed on the classes selected in our study. Data are grouped by system and corpus type (source code including comments and comments only). In all cases, the average entropy 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 (i.e., frequency), then it is hard to identify dominating terms that can be used to label such a class. These values justify the worst overlap of the two clustering-based IR methods, namely LSI and LDA. In addition, it is worth noting how the influence of the high entropy of the terms has a higher impact on LDA (that is a probabilistic model) than on LSI (that is a deterministic clustering-based method).

The analysis of Figure 2-(b) further emphasizes the relationship between the entropy and the capability of LSI and LDA to approximate the mental model of the subjects when labeling source code artifacts. In particular, the entropy of JHotDraw classes is higher when indexing only comments than when considering source code too. This has a negative impact on the overlap between the keywords identified by both LSI and LDA and those identified by subjects. In particular, the achieved overlap is generally lower when considering comments only than when indexing source code including comments (see Figure 1). On eXVantage, even if the entropy of classes when considering comments only is still higher than when considering source code including comments, the difference is not so evident as in JHotDraw. This justifies the almost comparable overlap achieved by both LSI and LDA for the two possible corpus (i.e., comments only and code plus comments).

To provide further evidence of the relationship between entropy and the overlap achieved by LDA, Table I shows two examples of classes with high and low entropy, respectively. In the table the terms identified by LDA that overlap with those identified by humans are in bold face. In the former case, the entropy is very high (0.97), thus it turns out to be difficult identifying the most important topic. This results in a very low overlap between automatic and human labelings. 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. Such terms are selected by LDA to label the class and, as indicating by the high overlap, the same terms were also selected by subjects.

It is important to note that the classes selected in our study are not the only ones having a high entropy. We also computed the entropy for all the classes of the two systems. We obtained an average entropy of 0.89 for ExVantage, and 0.87 for JHotDraw. 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, Figure 3 shows the classical weighted graph for visualizing all pairwise document distances achieved using LDA aiming at clustering source code artifacts of eXVantage at two different levels of granularity: Figure 3-(a) shows the results achieved by clustering all the classes of the system, while in Figure 3-(b) the clustering is performed at class level, i.e., LDA is used to cluster the methods of the class EventThread.java. Each node of the graph represents a class, while the weight of the edges measures the distance between topic distributions of each pair of classes. Thus, if two classes (or methods) have different
distribution of topics (i.e., they belong to different topics), then the correspondent nodes are far from each other in the graph. As we can notice from the figures, in both cases all classes (or methods) are concentrated in a small area. In such a scenario is quite difficult to discriminate between classes (or methods), and consequently efficiently clustering them.

We also analyzed two factors that could affect the time required by subjects to label the selected classes. We investigated whether such a time variation depends on two factors, namely class size and comment verbosity using the Pearson product-moment correlation. Table II reports the achieved results. The correlation analysis shows that the comment verbosity has a statistically significant negative correlation with time needed to label the selected classes for both eXVantage and JHotDraw. In particular, such a negative correlation means that, if the source code is not well commented, then it turns out to be very difficult for a human identify the responsibilities of a class and thus the keywords to characterize them. For the class size factor, there is no correlation with the time for eXVantage while such a factor has a high positive correlation (statistically significant) for JHotDraw. This result can be due the different verbosity of comments in the two software systems. Indeed, while for eXVantage the comments contains 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, if the classes are not well commented, the time needed to understand their responsibility (and then identify the labeling keywords) increases with the number of source code lines.

To identify whether the time needed to label classes correlates with the overlap achieved by the automated labeling techniques, we grouped the classes in two groups, based on the time spent by subjects to label them. In particular, a first group contains all classes for which the average time spent by subjects to identify the keywords is higher than the average, and a second cluster contains the remaining ones. Partitioning the classes in these two groups allowed us to analyze the overlap between the human labeling and the labeling provided by the automatic techniques on two different groups, i.e., classes requiring a high and a low effort.

Table III shows the achieved results grouped by software system, the time needed to label classes, and the type of corpus

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

TABLE II
INFLUENCE OF CODE SIZE AND COMMENT VERBOSITY ON THE TIME NEEDED TO LABEL THE SELECTED CLASSES. PEARSON PRODUCT-MOMENT CORRELATIONS.

<table>
<thead>
<tr>
<th>System</th>
<th>Class size</th>
<th>Comment verbosity</th>
</tr>
</thead>
<tbody>
<tr>
<td>JHotDraw</td>
<td>0.62</td>
<td>-0.25</td>
</tr>
<tr>
<td>eXVantage</td>
<td>0</td>
<td>-0.13</td>
</tr>
</tbody>
</table>

Fig. 3. Distance between source code elements of eXVantage.

(a) Distance between classes

(b) Distance between methods of EventThread.java
used to automatically identify the sets of keywords (source code including comments or comments only). A detailed analysis of results indicates that LSI using tf as weighting mechanism always achieves better overlap for classes requiring more time to be labeled by humans, while performance are lower for classes requiring less effort to be labeled. In particular, the different variants of LSI achieved an average overlap—for classes requiring a high time to be labeled—of 12% higher than those requiring less time for labeling. LDA allows to obtain less robustness with respect of increasing effort-proneness of the labeling task: in the 50% of the cases, the overlap achieved by LDA increases with the time spent, while it decreases in the remaining cases. VSM achieves better performance for classes having a low verbosity, while performs worse for most (6 cases out of 8) classes requiring a high effort (and thus likely to be more difficult to label). In the end, the heuristic based on the class name seems to be highly robust across different complexity levels: the difference in terms of overlap consists in an increasing about 4%–5% on highly complex classes.

IV. Threats to Validity

This section describes threats to validity that can affect 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, we rely on a widely used IR technique, i.e., the Jaccard overlap score. Time measurements annotated by subjects could be affected by imprecision as the experiment was conducted offline: however we clearly explained subjects to carefully record such a time, and there is no reason why would have cheated, also because the experiment was entirely on a voluntary basis and they were not evaluated on their performances during the experiment. Also, the pre-experiment code understanding would have unlikely influenced such a time, since subjects did not exactly which classes were the objects of the study.

Threats to internal validity are related to factors that can influence our results. To mitigate the effect of labeling variability across subjects, 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. [16] for their labeling. Furthermore, we investigate whether other factors, such as the class entropy and the similarity among classes in the systems could have influenced the results—in terms of set of selected topics—of the various techniques.

Threats to external validity concern the generalization of results. We tried to investigate as many IR-based heuristics as possible to perform automatic labeling (VSM, LDA, LSI, and a simple VSM applied on method signature, class name and attributes names only), different weighting scheme (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 vary if replicating the study on other systems and in particular on objects developed with different programming languages. last, but not least, we are aware that our labelings was 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 terms other than those in the method signatures/class names, thus decreasing the performances of the simple heuristic.

V. 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, a lot of effort in the software engineering community (both research and commercial) has been devoted to the analysis of textual information contained in the artifacts of software project repositories to support activities such as impact analysis [4], clone detection [5], feature location (e.g., [6]), to define new cohesion and coupling metrics (e.g., [7], [8]), to assess software quality (e.g., [1], [2], [9], [24]), to recover traceability links between software artifacts (e.g., [3], [25], [26], [27], [28], [29], [30]).

Related to our work is the study of textual analysis, and in particular topic modeling techniques, to mine and understand topics within source code. Such approaches aiming at providing support to program comprehension deriving a snapshot of the system easier to understand. Clustering semantically related classes or labeling can be considered as a way to summarize the responsibilities of source code artifacts aiming at aiding developers in comprehension tasks.
Maletic and Marcus [31] 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. [32] 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 functionalities. The authors also used LSI to label the identified clusters. Finally, a visual notation is provided aiming at giving an overview of all the clusters and their semantic relationships.

Baldi et al. [33] applied LDA to source code to automatically identify concerns. In particular, they used LDA to identify topics in the source code. Than, 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. [34] 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. [35] 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.

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 automatic generate summaries of bug reports [36] and software concerns [37]. Buse and Weimer proposed an approach to automatically generate human-readable documentation for arbitrary code differences [38]. Textual analysis have also been recently applied for the automatic summarization of source code artifacts with the purpose of aiding developers in comprehension tasks [39]. The evaluation of these approaches [40] revealed that they capture some important aspects of the code, but they also need improvement in order to satisfy the developer needs.

Murphy [41] 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 [42], and comments for high-level actions [43]. The automatic generation of comments can be considered as a kind of summarization of source code components.

VI. CONCLUSION AND FUTURE WORK

In recent years researchers have been 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. In this paper we presented an empirical study aimed at investigating to what extent a source code labeling based on IR technique would identify relevant words in the source code, compared to the words a human developer would have selected during a program comprehension task.

In the context of our study we asked 17 Bachelor students to describe ten classes of two Java systems, namely JHotDraw (an open source system) and eXVantage (an industrial system), with a set of 10 keywords. Then, we applied different techniques to automatically extract keywords from the selected classes. We used different IR techniques (several variants of VSM, LSI, and LDA) and a simple heuristic that consider the terms contained in the class name, in the signature of the methods, and in the instance variables. Once having the set of keywords identified by students and the set of keywords identified by the experimented technique, we compare them in order to compute the overlap.

Results show that—despite overall there is a relatively high overlap between automatic and human-generated labels—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. In particular, the high entropy of terms in the classes inhibits the capability of topic models techniques 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 [19]. Unfortunately, such an heterogeneity is always present in source code artifacts. Thus, for labeling source code ad-hoc heuristics can be used to better approximate the mental model used by developers when identifying class keywords.

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.

ACKNOWLEDGEMENT

We would like to thank all the students that participated in our study as subjects.

REFERENCES
