Classifying code comments in Java software systems

Pascarella, Luca; Bruntink, Magiel; Bacchelli, Alberto

DOI
10.1007/s10664-019-09694-w

Publication date
2019

Document Version
Final published version

Published in
Empirical Software Engineering

Citation (APA)
Pascarella, L., Bruntink, M., & Bacchelli, A. (2019). Classifying code comments in Java software systems. *Empirical Software Engineering*, *24*(3), 1499-1537. https://doi.org/10.1007/s10664-019-09694-w

Empirical Software Engineering
https://doi.org/10.1007/s10664-019-09694-w

Classifying code comments in Java software systems

Luca Pascarella[1] · Magiel Bruntink[2] · Alberto Bacchelli[3]

Published online: 26 April 2019

Abstract
Code comments are a key software component containing information about the underlying implementation. Several studies have shown that code comments enhance the readability of the code. Nevertheless, not all the comments have the same goal and target audience. In this paper, we investigate how 14 diverse Java open and closed source software projects use code comments, with the aim of understanding their purpose. Through our analysis, we produce a taxonomy of source code comments; subsequently, we investigate how often each category occur by manually classifying more than 40,000 lines of code comments from the aforementioned projects. In addition, we investigate how to automatically classify code comments at line level into our taxonomy using machine learning; initial results are promising and suggest that an accurate classification is within reach, even when training the machine learner on projects different than the target one. Data and Materials [https://doi.org/10.5281/zenodo.2628361].

Keywords Code comments usage · Code comment classification · Dataset

1 Introduction

While writing and reading source code, software engineers routinely introduce code comments (Fluri et al. 2007). Several researchers investigated the usefulness of these comments, showing that thoroughly commented code is more readable and maintainable. For example, Woodfield et al. conducted one of the first experiments demonstrating that code comments improve program readability (Woodfield et al. 1981), then Tenny et al. confirmed these

Communicated by: Abram Hindle and Lin Tan

✉ Luca Pascarella
l.pascarella@tudelft.nl

Magiel Bruntink
m.bruntink@sig.eu

Alberto Bacchelli
bacchelli@ifi.uzh.ch

[1] Delft University of Technology, Delft, The Netherlands

[2] Software Improvement Group, Amsterdam, The Netherlands

[3] University of Zurich, Zurich, Switzerland

results with more experiments (Tenny 1985, 1988). Hartzman et al. investigated the economical maintenance of large software products showing that comments are crucial for maintenance (Hartzman and Austin 1993).

Jiang et al. found that comments that are not aligned with the annotated functions confuse authors of future code changes (Jiang and Hassan 2006).

Overall, given these results, having abundant comments in the source code is a recognized good practice (de Souza et al. 2005). Accordingly, researchers proposed to evaluate code quality with a metric based on code/comment ratio (Oman and Hagemeister 1992; Garcia and Granja-Alvarez 1996).

Nevertheless, not all the comments are the same. This is evident, for example, by glancing through the comments in a source code file[1]

from the Java Apache Hadoop Framework 2017. In fact, we see that some comments target end-user programmers (*e.g.*, Javadoc), while others target internal developers (*e.g.*, *inline* comments); moreover, each comment is used for a different purpose, such as providing the implementation rationale, separating logical blocks, and adding reminders; finally, the interpretation of a comment also depends on its position with respect to the source code. Defining a taxonomy of the source code comments is still an open research problem.

Haouari et al. (2011) and Steidl et al. (2013b) presented the earliest and most significant results in comments' classification. Haouari et al. investigated developers' commenting habits, focusing on the position of comments with respect to source code and proposing an initial taxonomy that includes four high-level categories (Haouari et al. 2011); Steidl et al. proposed a semi-automated approach for the quantitative and qualitative evaluation of comment quality, based on classifying comments in seven high-level categories (Steidl et al. 2013b). In spite of the innovative techniques they proposed to understand developers' commenting habits and to assess comments' quality, the classification of comments was not in their primary focus.

In the work presented in this article, we focus on increasing our empirical understanding of the types of comments that developers write in source code files. This is a key step to guide future research on the topic. Moreover, this increased understanding has the potential to (1) improve current quality analysis approaches that are restricted to the comment ratio metric only (Oman and Hagemeister 1992; Garcia and Granja-Alvarez 1996) and to (2) strengthen the reliability of mining approaches that use comments as input (*e.g.*, Tan et al. 2007; Padioleau et al. 2009).

To this aim, we conducted an in-depth analysis of the comments in the Java source code files of six major OSS systems and eight industrial projects. We set up our study as an exploratory investigation. We started without hypotheses regarding the content of source code comments, with the aim of discovering the comments' purposes and roles, their format, and their frequency. To this end, we (1) conducted three iterative content analysis sessions (involving four researchers) over 50 source files including about 250 comment blocks to define an initial taxonomy of code comments, (2) validated the taxonomy externally with 3 developers, (3) inspected 2,000 open source and 4,000 closed source code files and manually classified (using a new application we devised for this purpose) over 24,000 comment blocks comprising more than 40,000 lines, (4) used the resulting dataset to evaluate how effectively comments can be automatically classified, and (5) investigated how many comments from an unseen project should be manually classified to improve the performance of an automatic classification approach trained on other projects.

Our results show that developers write comments with a large variety of different meanings and that this should be taken into account by analyses and techniques that rely on

[1] https://tinyurl.com/zqeqgpq

code comments. The most prominent category of comments summarizes the purpose of the code, confirming the importance of research related to automatically creating these type of comments. Finally, our automated classification approach, based on supervised algorithms, reaches promising initial results, even when training on software projects that are different than the target project.

2 Motivating Example

Listing 1 shows an example Java source code file that contains both code and comments. In a well-documented file, comments help the reader with a number of tasks, such as understanding the code, knowing the choices and rationale of authors, and finding additional references. When developers perform software maintenance, the aforementioned tasks become mandatory steps that practitioners should attend. The fluency in performing maintenance tasks depends on the quality of both code and comments. When comments are omitted, much depends on the ability of developers and the complexity of the code; when well-written comments are present, the maintenance could be simplified.

2.1 Code/comment Ratio to Measure Software Maintainability

When developers want to estimate the maintainability of code, one of the simplest solutions is to compute the code/comment ratio, as proposed by Garcia and Granja-Alvarez (1996). By evaluating the aforementioned metric in the snippet in Listing 1, we find an overall indicator of quality, which—however—is inaccurate. The inaccuracy arises from the fact

```java
public class STSubscriptExpression extends STExpression {

    private static CSpellingService fInstance;

    /**
     * Returns the created expression, or null in case of error.
     * @deprecated Replaced by {@link #getExpression()}
     */
    @Deprecated
    public STExpression getSubscriptExpression(){
        if (fInstance == null) {
            fInstance = new Expression(ConsoleEditors.getPreferenceStore());
        }
        return fInstance;
    }

    /**
     * Handle terminated sub-launch
     * @param launch a terminable launch object.
     * @author Jesse MC Wilson
     */
    private void STLaunchTerminated(ILaunch launch) {
        // See com.vaadin.data.query.QueryDelegate#getPrimaryKeyColumns
        if (this == launch)
            return;
        // Remove sub launch, keeping the processes of the terminated launch to
        // show the association and to keep the console content accessible
        if (subLaunches.remove(launch) != null) {
            // terminate ourselves if this is the last sub launch
            if (subLaunches.size() == 0) {
                // TODO: Check the possibility to exclude it
                //monitor.exlude();
                monitor.subTask("Terminated"); //$NON-NLS-1$
                fTerminated = true;
                fireTerminate();
                // %%%
            }
        }
    }
}
```

Listing 1 Example of Java file

that this metric considers only one kind of comment. More precisely, Garcia et al. focus only on the presence or absence of comments, omitting the possibility of use comments with different benefits for different end-users. The previous sample of code represents a case where the author used comments for different purposes. The comment on line 31 represents a note that developers use to remember an activity, an improvement, or a fix. On line 20 the author marks his contribution on the file. Both these two comments represent real cases where the presence of comments increases the code/comment ratio without any real effect on code readability or maintanability. This situation hinders the validity of this kind of metric and indicates the need for a more accurate approach to tackle the problem.

2.2 An Existing Taxonomy of Source Code Comments

A great source of inspiration for our work comes from Steidl et al. who presented a first detailed approach for evaluating comment quality (Steidl et al. 2013b). One of the key steps of their approach is to first automatically categorize the comments to differentiate between different comment types. They define a preliminary taxonomy of comments that comprises 7 high-level categories: COPYRIGHT, HEADER, MEMBER, INLINE, SECTION, CODE, and TASK. They provide evidence that their quality model, based on this taxonomy, provides important insights on documentation quality and can reveal quality defects in practice.

The study of Steidl et al. demonstrates the importance of treating comments in a way that suits their different categories. However, the creation of the taxonomy was not the focus of their work, as also witnessed by the few details given about the process that led to its creation. In fact, we found a number of cases in which the categories did not provide adequate information or did not differentiate the type of comments enough to obtain a clear understanding. To detail this, we consider three examples from Listing 1:

Member category. Lines 5, 6, 7 and 8 correspond to the MEMBER category in the taxonomy by Steidl et al. In fact, MEMBER comments describe the features of a method or field being located near to definition (Steidl et al. 2013b). Nevertheless, we see that the function of line 6 differs from that of line 7; the former summarizes the purpose of the method, the latter gives notice about replacing the usage of the method with an alternative. By classifying these two lines together, one would lose this important difference.

IDE directives. Lines 33 does not belong to any explicit category in the taxonomy by Steidl et al. In this case, the target is not a developer, but the Integrated Development Environment (IDE). Similarly, line 23 does not have a category in the taxonomy by Steidl et al., but it is a possibly important external reference to read for more details.

Unknown. Line 36 represents a case of a comment that should be disregarded from any further analysis. Since it does not separate parts, the SECTION would not apply and an automated classification approach would try to wrongly assign it to one of the other categories. The taxonomy by Steidl et al. does not consider *unknown* as a category.

With our work, we specifically focus on devising an empirically grounded, fine-grained classification of comments that expands on the previous initial efforts by Steidl et al. Our aim is to get a comprehensive view of the comments, by focusing on the purpose of the comments written by developers. Besides improving our scientific understanding of this type of artifacts, we expect this work to be also beneficial, for example, to the effectiveness of the quality model proposed by Steidl et al. and other approaches relying on mining and analyzing code comments (*e.g.*, Oman and Hagemeister 1992; Tan et al. 2007; Padioleau et al. 2009).

3 Methodology

This section defines the overall goal of our study, motivates our research questions, and outlines our research method.

3.1 Research Questions

The ultimate goal of this study is to understand and classify the primary purpose of code comments written by software developers. In fact, past research showed evidence that comments provide practitioners with a great assistance during maintenance and future development, but not all the comments are the same or bring the same value.

We started by analyzing past literature searching for similar efforts on analysis of code comments. We observed that a few studies completed a taxonomy of comments, in a preliminary fashion. Indeed, most of past work focuses on the impact of comments on software development processes such as code understanding, maintenance, or code review and the classification of comments is only treated as a side outcome (*e.g.*, Tenny 1985; Tenny 1988).

RQ1. How can code comments be categorized?

Given the importance of comments in software development, the natural next step is to apply the resulting taxonomy and investigate on the primary use of comments. Therefore, we investigate whether some classes of comments are predominant and whether patterns across different projects or domains (*e.g.*, open source and industrial systems) exist. This investigation is reflected in our second research question:

RQ2. How often does each category occur in OSS and industrial projects?

Subsequently, we investigate to what extent an automated approach can classify unseen code comments according to the taxonomy defined in RQ1. An accurate automated classification mechanism is the first essential step in using the taxonomy to mine information from large-scale projects and to improve existing approaches that rely on code comments. This leads to our third research question:

RQ3. How effective is an automated approach, based on machine learning, in classifying code comments in OSS and industrial projects?

Finally, we expect that practitioners or researchers could benefit by applying our machine learning algorithm to an unseen real project. For this reason, we investigate how much the performance are improved by manually classifying an increasing number of comments in a new project and providing this information to our machine learning algorithm. This evaluation leads to our last research question:

RQ4. How does the performance of an automated approach improve by adding classified comments from the system under classification?

3.2 Selection of Subject Systems

To conduct our analysis, we focused on a single programming language (*i.e.*, Java, one of the most popular programming languages Diakopoulos and Cass 2016) and on projects that are either developed in an open source setting or in an industrial one.

OSS context: Subject systems. We selected six heterogeneous software systems: Apache Spark (Apache Spark 2016), Eclipse CDT, Google Guava, Apache Hadoop, Google Guice, and Vaadin. They are all open source projects and the history of the changes are controlled with GIT version control system. Table 1 details the selected systems. We select unrelated projects emerging from the context of different four software ecosystems (*i.e.*, Apache, Google, Eclipse, and Vaadin); the development environment, the number of contributors, and the project size are different: Our aim is to increase the diversity of comments that we find in our dataset.

Industrial context: Subject systems. We also include heterogeneous industrial software projects, which are clients of the company in which the second author works. Table 2 reports the anonymized characteristics of such projects, respecting their non-disclosure agreements.

3.3 RQ1. Categorizing Code Comments

To answer our first research question, we (1) defined the comment granularity we consider, we (2) conducted three iterative *content analysis sessions* (Lidwell et al. 2010) involving four software engineering researchers with at least three years of programming experience, and we (3) validated our categories involving other three professional developers.

Comment granularity Java offers three alternative ways to comment source code: *inline* comments, *multi-line* comments, and *JavaDoc* comments (which is a special case of multi-line comments). A comment (especially multi-line ones) may contain difference pieces of information with different purposes, hence belonging to different categories. Moreover, a comment may be a natural language word or an arbitrary sequence of characters that, for example, represent a delimiter or a directive for the preprocessor. For this reason, we conducted our manual classification at *character level*. The user specifies the starting and ending character of each comment block and its classification. For example, the user could categorize two parts of a single inline comment into two different classes. By choosing a fine-grained granularity at *character level* users are responsible for identifying comment's delimiters (*i.e.*, the text is not automatically split into tokens). Even if this choice may complicate the user's work, this flexibility, chosen during the manual classification, allowed us both to define the taxonomy precisely and to have a basis to decide the appropriate comment granularity for the automatic classification, *i.e.*, line granularity (see Section 3.5 – 'Classification granularity').

Definition phase This phase involved four researchers in software engineering (three Ph.D. candidates and one faculty member). Two of these researchers are authors of this paper. In the first iteration, we started choosing six appropriate OSS projects (reported in Table 1) and sampling 35 files with a large variety of code comments. Subsequently, together we analyzed all source code and comments. During this analysis we could define some obvious categories and left undecided some comments; this resulted in the first draft taxonomy defining temporary category names. In the course of the second phase, we first conducted

Table 1 Details of the selected open source systems

Project	Java source lines			Commits	Contributors	Sample sets		
	Code	Comment	Ratio			Files	Blocks of comments	Ratio
Apache Spark	753k	287k	38%	38k	1,351	61	465	7.7%
Eclipse CDT	1,239k	466k	38%	26k	211	799	6,009	7.5%
Google Guava	252k	88k	35%	4k	185	158	1,100	6.9%
Apache Hadoop	1,258k	396k	31%	15k	171	672	4,228	6.3%
Google Guice	9k	5k	56%	2k	32	59	718	12.1%
Vaadin	2,643k	1,101k	42%	91k	726	401	3,340	8.3%
Overall	6M	2.3M	38%	176k	2.7k	2k	16k	8%

Table 2 Details of the anonymize industrial systems

Project	Java source lines			Sample sets		
	Code	Comments	Ratio	Files	Blocks of comments	Ratio
P1	478k	12k	2.5%	159	1,386	8.7%
P2	69k	0.1k	0.1%	236	102	0.4%
P3	2,044k	19k	1.0%	503	1,673	3.3%
P4	1,065k	5k	0.5%	761	1,223	1.6%
P5	1,026k	7k	0.6%	506	1,073	2.1%
P6	3,088k	5k	0.2%	503	273	0.5%
P7	1,173k	13k	1.1%	290	1,058	3.6%
P8	1,208k	4k	0.3%	1,042	1,179	1.1%
Overall	10M	65k	0.7%	4k	8k	2%

an individual work analyzing 10 new files, in order to check or suggest improvements to the previous taxonomy, then we gathered together to discuss the findings. The second phase resulted in a validation of some clusters in our draft and the redefinitions of others. The third phase was conducted in team and we analyzed five files that were previously unseen. During this session, we completed the final draft of our taxonomy verifying that each kind of comments we encountered was covered by our definitions and those overlapping categories were absent.

Through this iterative process, we defined a hierarchical taxonomy with two layers. The top layer consists of six categories and the inner layer consists of 16 subcategories.

Validation phase We validated the resulting taxonomy externally with three professional developers who had three to five years of Java programming experience and were not involved in the writing of this work. We conducted one session with each developer. At the beginning of the session, the developer received a printed copy of the description of the comment categories in our taxonomy (similar to the explanation we provide in Section 4.1) and was allowed to read through it and ask questions to the researcher guiding the session. Afterwards, each developer was required to login into CommMean (a web application, described in Section 3.4) and classify—according to the provided taxonomy—each piece of comment (*i.e.*, by arbitrary specifying the sequence of adjacent characters that identify words, lines, or blocks belonging to the same category) in three Java source code files (the same files have been used for all the developers) that contained a total of 138 different lines of comments. During the classification, the researcher was not in the experiment room, but the printed taxonomy could be consulted. At the end of the session, the guiding researcher came back to the experiment room and asked the participant to comment on the taxonomy and the classification task. At the end of all the three sessions, we compared the differences (if any) among the classifications that the developers produced.

All the participants found the categories to be clear and the task to be feasible; however, they also reported the need for consulting the printed taxonomy several times during the session to make sure that their choice was in line with the description of the category. Although they observed that the categories were clear, the analysis of the three sets of answers showed differences. We computed the inter-rater reliability by using Fleiss' kappa value (Fleiss 1971) and found a corresponding k value above 0.9 (*i.e.*, very good) for the three raters and the 138 lines they classified. We individually analyzed each case of disagreement by asking the participants to re-evaluate their choices after better extrapolating the context. Following this approach, the annotators converged on a common decision.

Exhaustiveness by cross-license validation As a side effect of investigating the second industrial dataset (which covers different commercial licenses and market strategies) for RQ2, we cross-validated the exhaustiveness of the categories and subcategories present in our taxonomy in a different context.

In fact, we directly applied our definitions to the comments of 8 industrial projects producing a manually classified set of 8,000 block of comments. During the labeling phase, we adopted all definitions present in the proposed codebook and even though we found a significant different distribution of each category, we found that such definitions are sufficiently exhaustive to cover all types of comment present also in the new set of projects.

This second validation provided additional corroborating evidence that the proposed taxonomy (initially generated involving 4 software engineering researchers, which iteratively observed the content of a sample of 50 Java open source files) fits all the comments that we encountered in both open and closed source projects, adequately.

3.4 A Dataset of Categorized Code Comments

To answer the second research question about the frequencies of each category, we needed a statistically significant set of code comments classified accordingly to the taxonomy produced as an answer to RQ1. Since the classification had to be done manually, we relied on random sampling to produce a statistically significant set of code comments. Combining the sets of OSS and industrial project, we classified comments in a total of 6,000 Java files from six open source projects and eight industrial projects. Our aim is to give a representative overview of how developers use comments and how these comments are distributed.

To reach this number of files for which we manually annotate the comments, we adopted two slightly different sampling strategies for OSS and industrial projects; we detail these strategies in the following.

OSS Projects: Sampling files. To establish the size of statistically significant sample sets for our manual classification, we used as a unit the number of files, rather than the number of comments: This results in the creation of a sample set that gives an additional overview of how comments are distributed in a system. We established the size (n) of such set with the following formula (Triola 2006, pp. 328-331):

$$n = \frac{N \cdot \hat{p}\hat{q}\left(z_{\alpha/2}\right)^2}{(N-1)\,E^2 + \hat{p}\hat{q}\left(z_{\alpha/2}\right)^2}$$

The size has been chosen to allow the simple random sampling without replacement. In the formula, $\hat{p}$ is a value between 0 and 1 that represents the proportion of files containing a given category of code comment, while $\hat{q}$ is the proportion of files not containing such kind of comment (*i.e.*, $\hat{q} = 1 - \hat{p}$). Since the *a-priori* proportion of $\hat{p}$ is not known, we consider the worst case scenario where $\hat{p} \cdot \hat{q} = 0.25$. In addition, considering we are dealing with a small population (*i.e.*, 557 Java files for Google Guice project) we use the finite population correction factor to take into account their size (N). We sample to reach a confidence level of 95% and error (E) of 5% (*i.e.*, if a specific comment is present in $f\%$ of the files in the sample set, we are 95% confident it will be in $f\% \pm 5\%$ files of our population). The suggested value for the sample set is 1,925 files. In addition, since we split the sample sets in two parts with an overlapped chunk for validation, we finally sampled 2,000 files. This value does not change significantly the error level that remains close to 5%. This choice only validates the quality of our dataset as a representation of the overall population: It is not related to the *precision* and *recall* values presented later, which are actual values based on manually analyzed elements.

Industrial projects: Sampling files. As done in the OSS case we selected a statistically significant sample of files belonging to industrial projects. We relied on simple random sampling without replacement to select a sufficient amount of files representative of the eight industrial projects that we considered in this study. According to the formula (Triola 2006) used for the sampling in the OSS context, we defined a sample of 2,000 Java files with a confidence level of 95% and error of 5% (*i.e.*, if a specific comment is present in $f\%$ of the files in the sample set, we are 95% confident it will be in $f\% \pm 5\%$ files of our population). Since we expected a similar workload for both domains we started with the same number of files. However, during the inspection, we found out that we still had resources to conduct a deep investigation, because we found a less number of comments per file. Therefore, we decided to double the number of files to inspect manually. This condition led to the creation of a sample set of 4,000 Java files for the industrial study.

Manual classification Once the sample of files with comments was selected, each of them had to be manually classified according to our taxonomy. For the manual classification, we rely on the human ability to understand and categorize written text expressed in natural language, specifically, code comments. To support the users during this tedious works that may be error-prone due to the repetitiveness of the task (especially for large datasets), we developed a web application named ComMean to conduct the classification. ComMean (i) shows one file at a time, (ii) allows the user to save the current progress for further inspections, and (iii) highlights the classified instances with different colors and opacity. During the inspection, the user can arbitrarily choose the selection granularity (*e.g.*, s/he can select a part of a line, an entire line, or a block composed of multiple lines) by selecting the starting and ending characters. For the given selection, the user can assign a label corresponding to one of the categories in our taxonomy.

The first and last authors of this paper manually inspected the sample set composed of 2,000 open source files and 4,000 industrial files. One author analyzed 100% of these files, while another analyzed a random, overlapping subset comprising 10% of the files. These overlapped files were used to verify their agreement, which, similarly to the external validation of the taxonomy with professional developers (Section 3.3), highlighted only negligible differences. More precisely, every participant independently read and labeled his own set of comments. If labels matched we accepted those cases as resolved, otherwise, we discussed each unmatched case. During the discussion, we evaluated the reasons behind a certain decision. Then, we came to a conclusion of choosing a single label. In most cases, the opinions differed due to the ambiguous nature of the comments. In these cases, we analyzed the context and tried a second run. Finally, we resolved these comments by carefully analyzing comments and the code context.

This large-scale categorization helped giving an indication of the exhaustiveness of the taxonomy created in RQ1 with respect to the comments present in our sample: None of the annotators felt that comments, or parts of the comments, should have been classified by creating a new category. Although promising, this finding is applicable only to our dataset and its generalizability to other contexts should be assessed in future studies. The annotations referring to open source projects as well as ComMean are publicly available (Pascarella and Bacchelli 2017); the dataset constructed with industrial data cannot be made public due to non-disclosure agreements.

3.5 Automated Classification of Source Code Comments

In the third research question, we set to investigate to what extent and with which accuracy source code comments can be automatically categorized according to the taxonomy resulting from the answer to RQ1 (Section 4.1).

Employing sophisticated classification techniques (*e.g.*, based on deep learning approaches Goodfellow et al. 2016) to accomplish this task goes beyond the scope of the current work. Our aim is twofold: (1) Verifying whether it is feasible to create an automatic classification approach that provides fair accuracy and (2) defining a reasonable baseline against which future methods can be tested.

Classification granularity We set the automated classification to work *at line level*. In fact, from our manual classification, we found several blocks of comments that had to be split and classified into different categories (similarly to the block defined in the lines 5–8 in Listing 1) and in the vast majority of the cases (96%), the split was at line level. In only less than 4% of the cases, one line had to be classified into more than one category. In these cases, we replicated the line in our dataset for each of the assigned categories, to get a lower bound on the effectiveness in these cases.

Classification technique Having created a reasonably large dataset to answer RQ2 (it comprises more than 15,000 comment blocks totaling over 30,000 lines in OSS and up to 8,000 comment blocks that correspond to 10,000 lines in industrial systems), we employ *supervised* machine learning (Friedman et al. 2001) to build the automated classification approach. This kind of machine learning uses a pre-classified set of samples to infer the classification function. In particular, we tested two different classes of supervised classifiers: (1) *probabilistic classifiers*, such as naive Bayes or naive Bayes Multinominal, and (2) *decision tree algorithms*, such as J48 and Random Forest.

These classes make different assumptions on the underlying data, as well as have different advantages and drawbacks in terms of execution speed and overfitting.

Data balancing Chawla et al. study the effect of an approach to limit the problem of data imbalance named Synthetic Minority Over-sampling Technique (SMOTE) (Chawla et al. 2002). Specifically, their method tries to over-sample the minority occurrences and under-sample the majority classes to achieve better performance in a classification task. Data imbalance, in fact, is a frequent issue in classification problems occurring when the number of instances that refer to frequent classes is higher than uncommon instances (in our case DISCARDED, UNDER DEVELOPMENT, and STYLE & IDE classes). To ensure that our results would not have been biased by confounding factors, such as data imbalance (Chawla et al. 2002), we adopt the SMOTE package available in WEKA toolkit[2] with the aim of balancing our training sets. In addition, we relied on the work of O'brien (2007) to mitigate the issues that can derive from the multi-collinearity of independent variables. To this purpose, we compared the results of different classification techniques. Specifically, in our study, we address this problem by applying the RANDOM OVER-SAMPLING algorithm (Chawla 2009) implemented as a supervised filter in the WEKA toolkit. The filter re-weights the instances in the dataset to give them the same total weight for each class maintaining unchanged the total sum of weights across all instances.

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

Classification evaluation To evaluate the effectiveness of our automated technique to classification code comments into our taxonomy, we use two well known Information Retrieval (IR) metrics for the quality of results (Manning et al. 2008), namely *precision* and *recall*:

$$Precision = \frac{|TP|}{|TP + FP|}$$

$$Recall = \frac{|TP|}{|TP + FN|}$$

TP, FP, and FN are based on the following definitions:

- True Positives (TP): elements that are correctly retrieved by the approach under analysis (*i.e.*, comments categorized in accord to annotators)
- False Positives (FP): elements that are wrongly classified by the approach under analysis (*i.e.*, comments categorized in a different way by the oracle)
- False Negatives (FN): elements that are not retrieved by the approach under analysis (*i.e.*, comments present only in the oracle)

The union of TP and FN constitutes the set of correct classifications for a given category (or overall) present in the benchmark, while the union of TP and FP constitutes the set of comments as classified by the used approach. In other words, *precision* represents the fraction of the comments that are correctly classified into a given category, while *recall* represents the fraction of *relevant comments* in that category, where the *relevant comments* definition includes both true positive and false negative.

Effort/performance estimation With the fourth research question, we aim at estimating how many new code comments a researcher or developers should manually classify from an unseen project, in order to obtain higher performance when using our classification algorithm on such a project. Since the presence of classified comments from a project in the training set positively influences the performance of the classification algorithm on other comments from the same project, we want to measure how many instances are required in the training set to reach the knee of a hypothetical effort/performance curve.

To this aim, we quantify the exact extent to which each new manually classified block of comments contributes to produce better results, if any. We consider the project for which we obtained the worst results in cross-project validation when answering RQ3; then, we progressively add to the training set a fixed number of randomly selected comments belonging to the subject project and for each iteration we measure the performance of the model.

This evaluation starts from a cross-project setting (*i.e.*, we train only on different projects and test on an unseen one) and slowly gets to a within-project validation setting (*i.e.*, we train not only from different projects, but also from comments in the project that we are going to test with on unseen comments). We investigate what is among the lowest reasonable amount of comments that one should classify to get as close as possible to a full within-project setting.

3.6 Threats to Validity

Taxonomy validity To ensure that the comments categories emerged from our content analysis sessions were clear and accurate, and to evaluate whether our taxonomy provides an exhaustive and effective way to organize source code comments, we conducted a validation session that involved three experienced developers (see Section 3.3) external to content analysis sessions. These software engineers held an individual session on three unrelated Java source files. They found the categories to be clear and the task feasible, and the analysis of the three sets of answers showed a few minor differences. We counted the number of lines of comments classified with the same label by all participants and the number of lines of comments that at least two experts were in conflict. Finally, considering these two values we could calculate the percentage of comments that were classified with the same label by all participants. We measured that only 8% of the considered comments in the first run led to mismatches. Moreover, we individually analyzed each case by asking the participants to re-evaluate their choices after better extrapolate the context. Following that approach, the annotators converged on a common decision.

In addition, we reduce the impact of human errors during the creation of the dataset by developing ComMean, a web application to assist the annotation process.

External validity One potential criticism of a scientific study conducted on a small sample of projects is that it could deliver little knowledge. In addition, the study highlights the characteristics and distributions of 6 open source frameworks and 8 industrial projects mainly focusing on developers practices rather than end-users patterns. Historical evidence shows otherwise: Flyvbjerg gave many examples of individual cases contributing to discoveries in physics, economics, and social science (Flyvbjerg 2006). To answer to our research questions, we read and inspected more than 28,000 lines of comments belonging to 2,000 open source Java files and 12,000 lines of comments belonging to 4,000 closed source Java files (see Section 3.4) written by more than 3,000 contributors in a total of 14 different projects (in accord to Tables 1 and 2). We also chose projects belonging to different ecosystems and with different ment environments, number of contributors, and size of the project. To have an initial assessment of the generalizability of the approach we tested our results simulating this circumstance using the cross-project validation and cross-license validation (*i.e.*, training on OSS systems and testing on industrial systems, and viceversa) involving both open and closed source projects. Similarly, another threat concerning the generalizability is that our taxonomy refers only to a single object-oriented programming language *i.e.*, Java. However, since many object-oriented languages descend to common ancestor languages, many functionalities across object-oriented programming are similar and it is reasonable to expect the same to happen for their corresponding comments. Further research can be designed to investigate whether our results hold in other programming paradigms.

After having conducted the entire manual classification and the experiment, we realized that the exact location and the surrounding context of a code comment may be a valuable source of information to extract the semantic of the comment. Unfortunately, our tool ComMean did not record this information, thus we could not investigate how the performance of a machine learner would benefit from it. Future work should take this feature into account when designing similar experiments.

Moreover, RandomForest can be prone to overfitting, thus provide results that are too optimistic. To mitigate this threat, we use different training and testing mechanisms that create conditions that should decrease this problem (*e.g.*, within-project and cross-project).

Finally, a line of comment may have more than one meaning. We empirically found that this was the case for 4% of the inspected lines. We discarded these lines, as we considered this effect marginal, but this is a limitation of both our taxonomy and automatic classification mechanism.

4 Results and Analysis

In this section, we present and analyze the results of our research questions aimed at understanding what developers write in comments and with which frequency, as well as at evaluating the results of an automated classification approach and how manually classified comments from a project help improving the performance of a classifier trained on different projects.

4.1 RQ1. How Can Code Comments be Categorized?

Our manual analysis led to the creation of a taxonomy of comments having a hierarchy with two layers (Section 3.3). The top level categories gather comments with similar overall purpose, the internal levels provide a fine-grained definition using explanatory names. Figure 1 outlines all categories. The top level categories are composed of 6 distinct groups and the second level categories are composed of 16 definitions. We now describe each category with the corresponding subcategories.

A. **PURPOSE**

The PURPOSE category contains the code comments used to describe the functionality of linked source code either in a shorter way than the code itself or in a more exhaustive manner. Moreover, these comments are often written in a natural language and are used to describe the purpose or the behavior of the referenced source code. The keywords *'what'*, *'how'* and *'why'* describe the actions that take place in the source code in SUMMARY, EXPAND, and RATIONALE groups, respectively, which are the subcategories of PURPOSE:

A.1 **SUMMARY:** This type of comment contains a brief description of the behavior of the referenced source code. More generically, this class of comments represents answers to the question word *'what'*. Intuitively, this category incorporates comments that represent a sharp description of what the code does. Often, this kind of comments is used by developers to provide a summary that helps to understand the behavior of the code without reading it.

A.2 **EXPAND:** As with the previous category, the main purpose of reading this type of comment is to obtain a description of the associated code. In this case, the goal is to provide *more details* on the code itself. The question word *'how'* can be used to easily recognize the comments belonging to this category. Usually, these comments explain in detail the purpose of short parts of the code, such as details about a variable declaration.

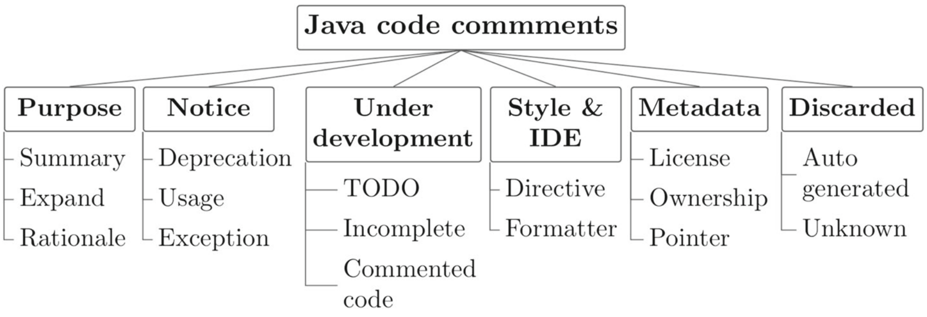

Fig. 1 Taxonomy of code comments

A.3 **RATIONALE:** This type of comment is used to explain the rationale behind some choices, patterns, or options. The comments that answer the question *'why'* belong to that category (*e.g.*, “Why does the code use that implementation?” or “Why did the developer use this specific option?”).

B. **NOTICE**

The NOTICE category contains the comments related to the description of warning, alerts, messages, or in general, functionalities that should be used with care. It covers the description of deprecated artifacts, as well as, the adopted strategies to move to new implementations. Further, it includes the use case examples giving to developer additional advice over parameters or options. Finally, it covers examples of use cases or warnings about exceptions.

B.1 **DEPRECATION:** This type of comment contains explicit warnings used to inform the users about deprecated interface artifacts. This subcategory contains comments related to alternative methods or classes that should be used (*e.g.*, ”do not use [this]”, ”is it safe to use?” or ”refer to: [ref]”). It also includes the description of the future or scheduled deprecation to inform the users of candidate changes. Sometimes, a tag comment such as *@version*, *@deprecated*, or *@since* is used.

B.2 **USAGE:** This type of comment regards explicit suggestions to users that are planning to use a functionality. It combines pure natural language text with examples, use cases, snippets of code, etc. Often, the advice is preceded by a metadata mark *e.g.*, *@usage*, *@param* or *@return*

B.3 **EXCEPTION:** This category describes the reasons for the occurred exception. Sometimes it contains potential suggestions to prevent the unwanted behavior or actions to do when that event arise. Some tags are used also in this case, such as *@throws* and *@exception*.

C. **UNDER DEVELOPMENT**

The UNDER DEVELOPMENT category covers the topics related to ongoing and future development. In addition, it envelopes temporary tips, notes, or suggestions that developers use during development. Sometimes informal requests of improvement or bug correction may also appear.

C.1 **TODO:** This type of comment regards explicit actions to be done or remarks both for the owners of the file and for other developers. It contains explicit fix notes about bugs to analyze and resolve, or already treated and fixed. Furthermore, it references to implicit TODO actions that may be potential enhancements or fixes.

C.2 **INCOMPLETE:** This type comprises partial, pending or empty comment bodies. It may be introduced intentionally or accidentally by developers and left in the incomplete state for some reason. This type may be added automatically by the IDE and not filled in by the developer *e.g.*, empty ”@param” or ”@return” directives.

C.3 **COMMENTED CODE:** This category is composed of comments that contain source code commented out by developers. It envelopes functional code in a comment to try hidden features or some work in progress. Usually, this type of comment represents features under test or temporarily removed. The effect of this kind of comments is directly transposed to the program flow.

D. **Style & IDE**

The Style & IDE category contains comments that are used to logically separate the code or provide special services. These comments may be added automatically by the IDE or used to communicate with it.

D.1 **Directive:** This is an additional text used to communicate with the IDE. It is in form of comments to be easily skipped by the compiler and it contains text of limited meaning to human readers. These comments are often added automatically by the IDE or used by developers to change the default behavior of the IDE or compiler.

D.2 **Formatter:** This type of comment represents a simple solution adopted by the developers to separate the source code in logical sections. The occurrence of patterns or the repetition of symbols is a good hint at the presence of a comment in the formatter category.

E. **Metadata**

The Metadata category aims to classify comments that define meta information about the code, such as authors, license, and external references. Usually, some specific tags (*e.g.*, *"@author"*) are used to mark the developer name and its ownership. The license section provides the legal information about the source code rights or the intellectual property.

E.1 **License:** Generally placed on top of the file, this types of comments describes the end-user license agreement, the terms of use, the possibility to study, share and modify the related resource. Commonly, it contains only a preliminary description and some external references to the complete policy agreement.

E.2 **Ownership:** These comments describe the authors and the ownership with different granularity. They may address methods, classes or files. In addition, this type of comment includes credentials or external references about the developers. A special tag is often used *e.g.*, *"@author"*.

E.3 **Pointer:** This types of comments contains references to linked resources. The common tags are: *"@see"*, *"@link"* and *"@url"*. Other times developers use custom references such as *"FIX #2611"* or *"BUG #82100"* that are examples of traditional external resources.

F. **Discarded**

This category groups the comments that do not fit into the categories previously defined; they have two flavors:

F.1 **Automatically generated:** This category defines auto-generated notes (*e.g.*, "Auto-generated method stub"). In most case, the comment represents the skeleton with a comment's placeholder provided by the IDE and left untouched by the developers.

F.2 **Unknown:** This category contains all remaining comments that are not covered by the previous categories. In addition, it contains the comments whose meaning is hard to understand due to their poor content (*e.g.*, meaningless because out of context).

Finding 1: The manual inspection of a sample of representative Java files from diverse open source software systems has led to the creation of a taxonomy for code comments composed by two layers with 6 top coarse-grained categories and 16 inner fine-grained categories.

4.2 RQ2. How Often Does Each Category Occur in OSS and Industrial Projects?

The second research question investigates the occurrence of each category of comments in the 6,000 source files that we manually classified from our six OSS systems and eight industrial projects. We first describe the results separately, then we contrast how the comments are distributed in the two settings.

OSS projects: Distribution of comments Figure 2 shows the distribution of the comments across the categories in the considered OSS systems. The figure reports the cumulative value for the top level categories (*e.g.*, NOTICE) and the absolute value for the inner categories (*e.g.*, EXCEPTION). For each category, the top red bar indicates the number of *blocks of comments* in the category, while the bottom blue bar indicates the number of non-blank *lines of comments* in the category.

Comparing blocks and lines, we see that the longest type of comments is LICENSE, with more than 11 lines on average per block. The EXPAND category follows with a similar average length. The SUMMARY category has only an average length of 1.4 lines, which is surprising, since it is used to describe the purpose of possibly very long methods, variables, or blocks of code. The other categories show negligible differences between number of blocks and lines.

We consider the quality metric code/comment ratio, which was proposed at line granularity (Oman and Hagemeister 1992; Garcia and Granja-Alvarez 1996), in the light of our results. We see that 59% of lines of comments should not be considered (*i.e.*, categories from C to F), as they do not reflect any aspect of the readability and maintainability of the code they pertain to; this would significantly change the results. On the other hand, if one considers blocks of comments, the result would be closer to the original code/comment metric purpose. In this case, a simple solution would be to only filter out the METADATA category, because the other categories seem to have a more negligible impact.

Considering the distribution of the comments, we see that the SUMMARY subcategory is the most prominent one. TThis is in line with the value of research efforts that attempt to generate summaries for functions and methods automatically, by analyzing the source code (Sridhara et al. 2010). In fact, these methods would alleviate developers from the burden of writing a significant amount of the comments we found in source code files. On the other hand, the SUMMARY accounts for only 24% of the overall lines of comments, thus suggesting that they only give a partial picture on the variety and role of this type of documentation. The second most prominent category is USAGE. Together with the prominence of SUMMARY, this suggests that the comments in the systems we analyzed are targeting end-user developers more frequently than internal developers. This is also confirmed by the low occurrence of the UNDER DEVELOPMENT category. Concerning UNDER DEVELOPMENT, the low number of comments in this category may also indicate that developers favor other channels to keep track of tasks to be done in the code.

Finally, the variety of categories of comments and their distribution underlines once more the importance of a classification effort before applying any analysis technique on the content and value of code comments. The low number of discarded cases corroborates the completeness of our taxonomy.

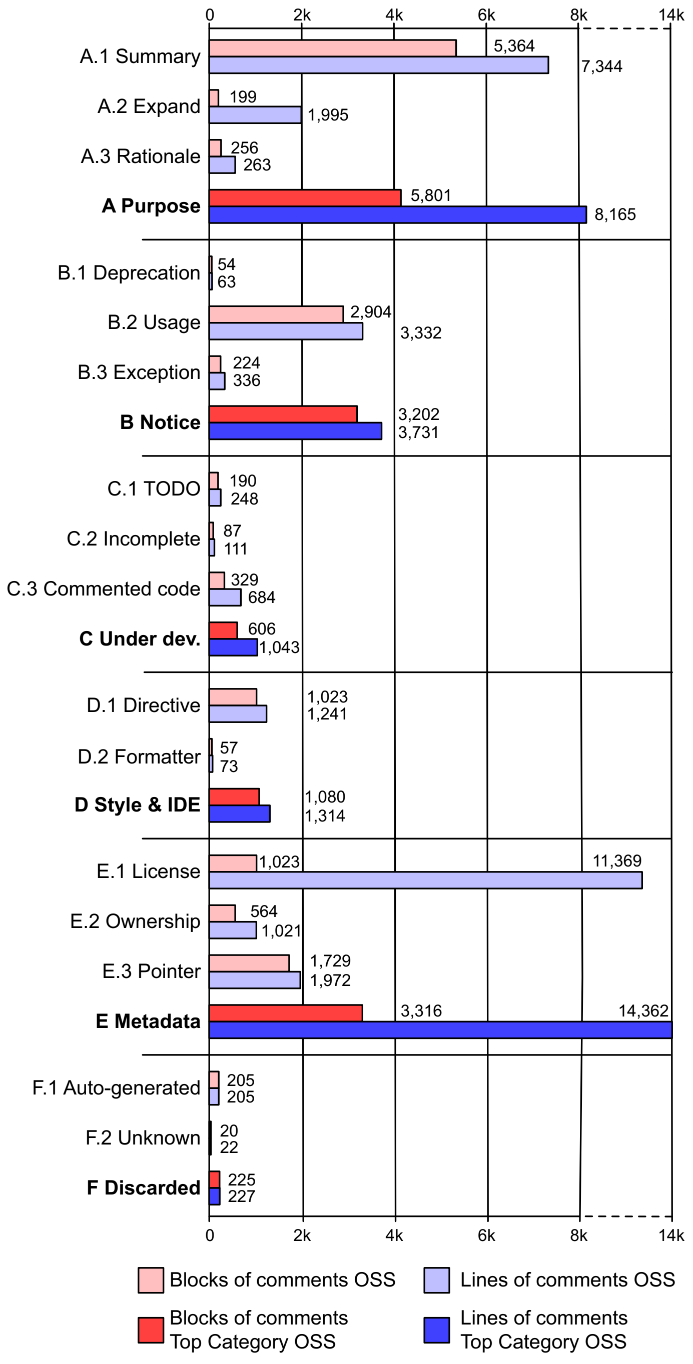

Fig. 2 Frequencies of comments per category in open source projects. Top, red bars show the occurrences by blocks of comments and bottom, blue bars by lines

Industrial projects: Distribution of comments Figure 3 shows the distribution of the comments across the categories in the considered industrial systems. To differentiate from the case of OSS systems, we use other colors: The top *green* bar indicates the number of blocks of comments, while the bottom *yellow* bar indicates the number of non-blank lines of comments.

Comparing number of blocks and number of lines, we see that most categories show a negligible differences between the two granularities. The largest, yet unremarkable difference is in the PURPOSE category (this is expected since this category includes both the SUMMARY and the EXPAND subcategories), in which we found 4,167 lines distributed over 3,436 blocks, with an average of 1.21 lines per block.

Considering the quality metric code/comment ratio in the industrial context, we see that 31% of the lines of comments should not be considered (*i.e.*, categories from C to F). This percentage is significantly lower than the case of OSS systems, whose distribution of comment lines is skewed by the LICENSE category. Past research has shown that these types of comments, which are especially structured, can be detected with high precision and recall even in free form documents (Bacchelli et al. 2010).

The SUMMARY subcategory is the most prominent one, thus corroborating the importance of research investigating ways to automatically generate this kind of comments (*e.g.*, Sridhara et al. 2010), also in the industrial setting. Matching the case of OSS systems, the second most prominent subcategory is USAGE, immediately followed by INCOMPLETE. This indicates that most comments target internal developers in the system, which is to be expected in a close source setting.

Finally, also in the industrial setting, the taxonomy was extensive enough to allow us to categorize all the source code comments without dropping any instance, even though we created this taxonomy from comments in OSS projects.

OSS vs. Industrial: Comparison of the distributions Figure 4 shows a comparison of the distribution of comments for the considered OSS systems and industrial projects, as a proportion of the total number of lines/blocks of comments in each context. The large difference in the frequency of LICENSE lines is evident, while we see that the categories PURPOSE, NOTICE, STYLE & IDE, and DISCARDED have substantially similar distributions. Another large difference regards the UNDER DEVELOPMENT category: The industrial projects we analyzed use source code comments for commenting code and leave incomplete comments far more frequently than OSS systems. This could be an indication that, if we exclude the LICENSE category, using code comments as an indicator for quality could be more appropriate for OSS systems. In fact, INCOMPLETE and COMMENTED CODE subcategories could be an indication of bad practices and low readability and maintainability of code, thus hindering the value of a comment/code metric. Investigating this hypothesis is beyond the scope of our current work, but studies can be devised and conducted to verify to which extent some types of comments indicate *problems* in the code, rather than a higher quality.

Finding 2: By comparing the distribution of comments in open and closed source software projects, we found that on average the former class of projects uses 4 times the number of comments compared to second set. The METADATA category is the most popular category in OSS and PURPOSE is the most popular category in closed source.

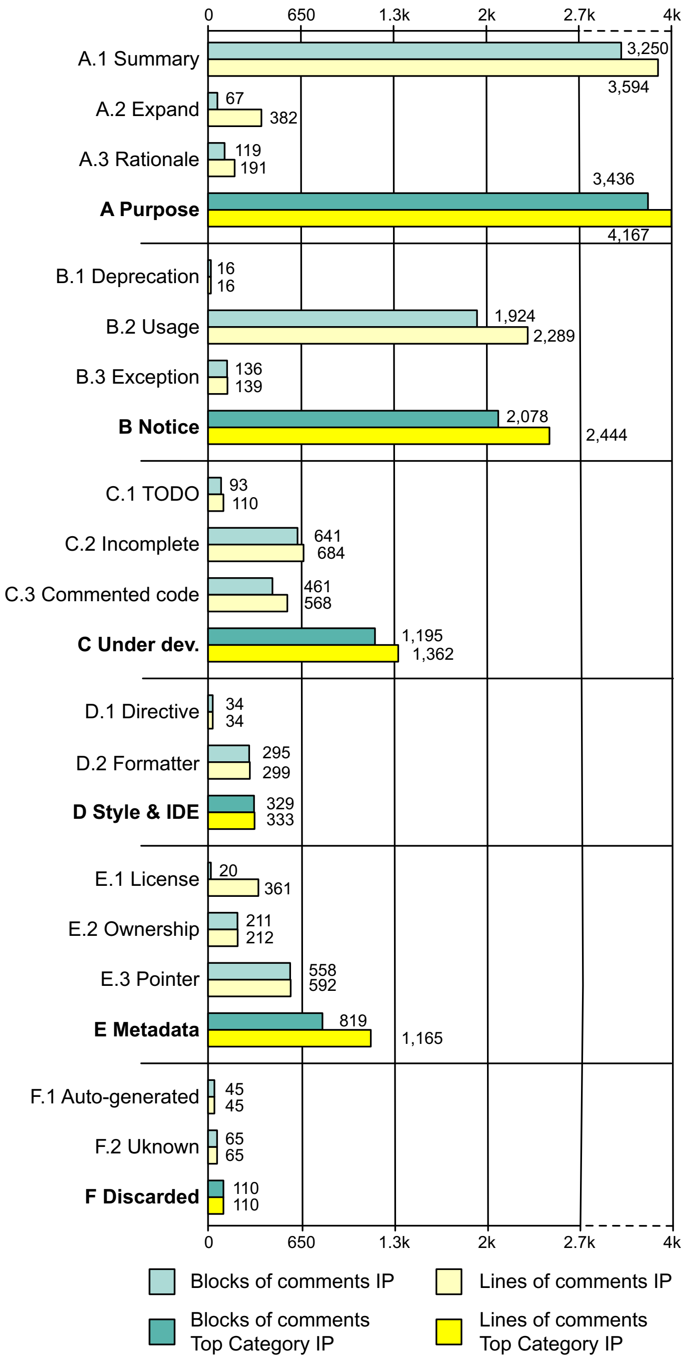

Fig. 3 Frequencies of comments per category industrial projects. Top, green bars show the occurrences by blocks of comments and bottom, yellow bars by lines

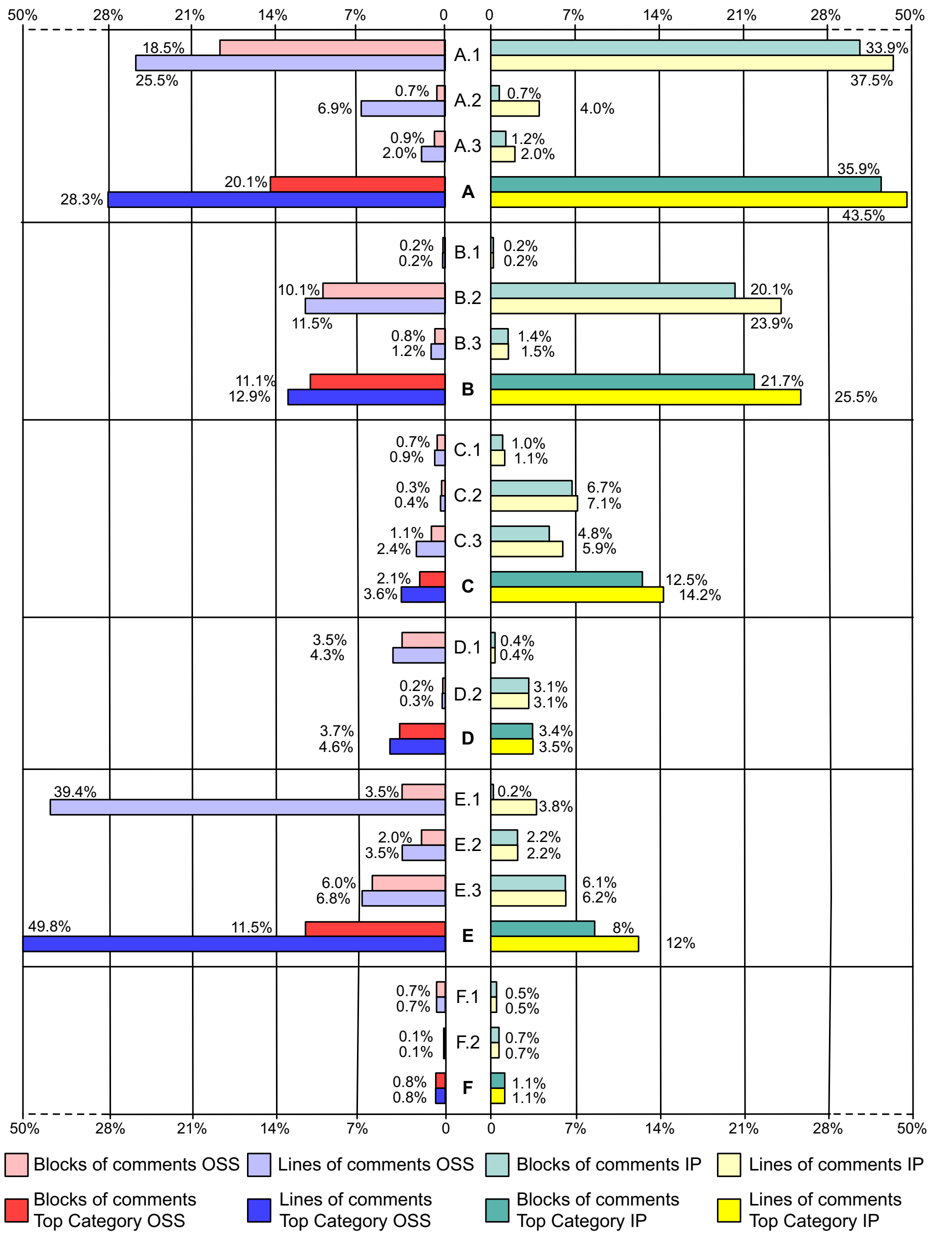

Fig. 4 Frequencies of comments per category. Top, red bars show the occurrences by blocks of comments and bottom, blue bars by lines

4.3 RQ3. How Effective is an Automated Approach, Based on Machine Learning, in Classifying Code Comments in OSS and Industrial Projects?

To evaluate the effectiveness of machine learning in classifying code comments we employed a supervised learning method. Supervised machine learning bases the decision evaluating on a pre-defined set of features, training on a set of labeled instances. Since we set to classify *lines* of code comments, we computed the features at line granularity.

Text preprocessing We preprocessed the comments by doing the following in this order: (1) tokenizing the words on spaces and punctuation (except for words such as '@usage' that would remain compounded), (2) splitting identifiers based on camel-casing (*e.g.*, 'ModelTree' became 'Model Tree'), (3) lowercasing the resulting terms, (4) removing numbers and rare symbols, and (5) creating one instance per line.

Feature creation Table 3 shows all the features that appear in the final model (these features are a subset of all that we initially devised). Since the optimal set of features is not known *a priori*, we started with some simple, traditional features and iteratively experimented with others more sophisticated features, in order to improve precision and recall for all the projects we analyzed.

A set of features commonly used in text recognition (Sebastiani 2002) consists in measuring the occurrence of the words; in fact, words are the fundamental tokens of all languages

Table 3 Machine learning features for comments classification

Feature	Type	Description
words	numeric	counts the occurrence of each word in the bag of unique words
punctuation	boolean	used in combination of a regular expression to distinguish source code from natural language *e.g.*, object.method(par1, par2);
words count	numeric	measures the length of the comment, using the words as unit size
unique words count	numeric	measures the length of the comment, only unique words are counted
row position	numeric	detects the absolute position of the comment
adjacent rows	numeric	recognizes the nature of the adjacent rows *e.g.*, comments or code
deprecation	boolean	true if a comment contains special tags like @deprecation
usage	boolean	true if a comment contains special tags such as @usage, @return or @value
exception	boolean	true if a comment contains special tags such as @exception or @throws
TODO	boolean	true if a comment contains keywords such as todo or fix or a link to a bug is detected
incomplete	boolean	true if a comment contains an empty body
commented code	boolean	true if a comment contains code snippets
directive	boolean	true if a comment contains special sequence of symbols used by IDE
formatter	boolean	true if a comment is composed of patterns of symbols or characters
xlicense	boolean	true if a comment contains words such as license, copyright, legal or law
ownership	boolean	true if a comment contains tags such as @author or @owner
pointer	boolean	true if a comment contains a reference to an external linkable resource
automatic generated	boolean	true if a comment contains text automatically inserted by IDE *e.g.*, Auto-generated method stub

we want to classify. To avoid overfitting to words too specific to a project, such as code identifiers, we considered only words above a certain threshold t. We found this value experimentally starting with a minimum of 3 and increasing up to 10, in one-unit steps. Since the values above 7 do not change the precision and recall quality, we chose that threshold.

In addition, other features consider the information about the *context* of the line, such as the text length, the comment position in the whole file, the number of rows, the nature of the adjacent rows, *etc.*

The last set of features is category specific. We defined regular expressions to recognize specific patterns. We report three detailed examples:

– This regular expression is used to match comments in single line or multiple lines with empty body.

```
^\s*\/(\*|\s)*(\/|\*\s*\*\/)\n*
```

– This regular expression matches the special keywords used in the *Usage* category.

```
(?i)@param|@usage|@since|@value|@return
```

– The following regular expression is used to find patterns of symbols that may be used in *Formatter* category.

```
([^{*}\s])(\1\1)|^{\s}*\/\/\/\s*\S*
|\$\S*\s*\S*\$
```

Machine learning validation with 10-fold We tested both probabilistic classifiers and decision tree algorithms. When using probabilistic classifiers, the average values of precision and recall were usually lower those obtained using decision tree algorithms. While, using decision tree algorithms, the percentage value associated with the correctly classified instances is improved. Particularly, with Random Forest we obtain up to 98.4% correctly classified instances. Nevertheless, in the latter case, many comments belonging to classes with a low occurrence were wrongly classified. Since the purpose of our tool is to best fit the aforementioned taxonomy we found that the best classifier is based on a probabilistic approach.

In Table 4 we report only the results (precision, recall, and weighted average TP rate) for the naive Bayes Multinominal classifier that on average, considering whole categories, achieves a better result accordingly to the aforementioned considerations. In Table 4 we intentionally leave empty cells that correspond to categories of comments that are not present in related projects. For the evaluation, we started with a standard 10-fold cross validation. Tables 4 and 5 show these results in the columns '10-fold' for open and closed source, respectively. In both cases, we obtain promising performance for the six high-level categories. Generically, the performance in the open source case is slightly higher than the closed source one. For OSS systems, precision and recall are always above 93%; for closed source projects, we have a drop of the performance up to 70% of precision for the DISCARDED category. This difference is most likely attributable to the smaller number of instances available for the training set of closed source projects. Indeed, the same trend is also visible in fine-grained categories. The precision for inner categories is in average better for OSS projects (with a minimum of 50% in the case of the RATIONALE category). In the closed source projects, both precision and recall for inner categories reach high value up to 100% for several categories, however, there are categories (RATIONALE DEPRECATION, and UNKNOWN) where the performance is below 70%.

Table 4 Results of the classification with naive bayes multinomial classifier in OSS

			Validation						
	P = Precision R = Recall			Cross project					
Top categories	Inner categories		10-fold	CDT	Guava	Guice	Hadoop	Vaadin	Spark
		P	0.88	0.96	0.68	0.61	0.72	0.62	0.65
	Summary	R	0.82	0.99	0.61	0.69	0.56	0.69	0.84
		P	1.00	0.84	0.00	0.00	0.09	0.00	0.00
	Expand	R	0.98	0.64	0.00	0.00	0.05	0.00	0.00
		P	0.50	0.56	0.15	0.00	0.10	0.03	0.67
	Rationale	R	0.69	0.84	0.23	0.00	0.41	0.17	0.17
		P	0.99	0.77	0.77	0.81	0.80	0.83	0.68
Purpose	Purpose	R	0.99	0.98	0.98	0.81	1.00	1.00	1.00
		P	0.74	0.75	0.22			0.14	
	Deprecation	R	0.78	0.81	1.00			1.00	
		P	0.86	0.85	0.50	0.43	0.67	0.90	0.56
	Usage	R	0.90	0.87	0.45	0.64	0.61	0.65	0.15
		P	0.76	0.75	0.43	0.00	0.58	0.69	0.13
	Exception	R	0.98	0.95	0.87	0.00	0.88	0.97	0.29
		P	1.00	0.50	0.50	0.36	0.60	1.00	1.00
Notice	Notice	R	0.98	0.50	0.50	1.00	0.41	0.33	0.17
		P	0.61	0.97	0.57	0.29	0.03	0.19	
	TODO	R	0.52	0.96	0.83	1.00	0.16	0.11	
		P	0.91	0.92			0.11	0.95	
	Incomplete	R	0.96	1.00			0.88	0.91	
		P	0.91	0.91			0.05	0.92	
	Commented code	R	0.91	0.95			0.06	0.50	
		P	0.98	1.00	0.00	0.00	0.00	0.00	
Under dev.	Under development	R	0.93	0.67	0.00	0.00	0.00	0.00	

Table 4 (continued)

			Validation						
	P = Precision R = Recall			Cross project					
Top categories	Inner categories		10-fold	CDT	Guava	Guice	Hadoop	Vaadin	Spark
		P	0.96	0.96				0.00	
	Directive	R	1.00	1.00				0.00	
		P	0.81	0.93	0.00			0.00	
	Formatter	R	0.77	0.28	0.00			0.00	
		P	0.97	1.00	1.00			0.00	
Style & IDE	Style & IDE	R	0.99	1.00	1.00			0.00	
		P	0.99	1.00	0.98	1.00	0.99	0.99	1.00
	License	R	0.98	0.99	1.00	0.95	0.99	1.00	1.00
		P	0.80	1.00	1.00	0.57	0.00	1.00	
	Ownership	R	0.96	1.00	0.08	0.27	0.00	0.98	
		P	0.84	0.80	0.82	0.81	0.79	0.97	1.00
	Pointer	R	0.94	0.74	0.52	0.54	0.70	0.85	0.60
		P	1.00	1.00	1.00	1.00	1.00	1.00	0.89
Metadata	Metadata	R	1.00	0.68	0.68	0.57	0.57	0.95	1.00
		P	0.90	0.91			0.13	0.84	
	Auto generated	R	1.00	1.00			1.00	1.00	
		P	0.65	1.00		0.00	0.00	0.00	
	Unknown	R	0.77	0.39		0.00	0.00	0.00	
		P	0.96	0.00		0.00	0.00	0.00	
Discarded	Discarded	R	0.98	0.00		0.00	0.00	0.00	
Weighted average TP rate			0.85	0.88	0.77	0.79	0.74	0.80	0.83

Table 5 Results of the classification with random forest classifier in industrial projects

			Validation								
	P = Precision R = Recall			Cross project							
Top categories	Inner categories		10-fold	P1	P2	P3	P4	P5	P6	P7	P8
		P	0.97	0.52	0.64	0.84	0.73	0.63	0.78	0.83	0.68
	Summary	R	1.00	0.86	0.87	0.89	0.86	0.97	0.69	0.89	0.77
		P	1.00	0.75	1.00	0.86	0.57	0.96	0.13	0.60	
	Expand	R	1.00	0.14	0.29	0.90	0.57	0.53	0.10	0.53	
		P	1.00	0.18	0.50	0.13	0.71	0.00	0.54	0.53	0.00
	Rationale	R	0.70	0.30	0.10	0.83	0.71	0.00	0.43	0.16	0.00
		P	0.98	0.56	0.92	0.92	0.95	0.72	0.88	1.00	0.70
Purpose	Purpose	R	0.99	0.99	0.97	0.91	0.84	0.94	0.79	0.72	0.83
		P	0.83	0.00		1.00	0.67	0.85			
	Deprecation	R	0.45	0.00		0.86	0.54	0.72			
		P	1.00	0.40	0.00	0.77	0.90	0.81	0.97	0.84	0.67
	Usage	R	1.00	0.91	0.00	0.86	0.69	0.98	0.85	0.84	0.81
		P	1.00	1.00		1.00	1.00	1.00	0.72	1.00	1.00
	Exception	R	0.96	0.94		0.25	1.00	1.00	0.70	0.31	0.96
		P	1.00	1.00	0.00	0.73	0.65	0.80	0.87	1.00	0.69
Notice	Notice	R	1.00	0.58	0.00	0.88	0.99	0.97	0.85	0.58	0.89
		P	0.99	0.56			0.00			1.00	0.15
	TODO	R	0.86	0.55			0.00			0.53	0.08
		P	1.00				1.00			0.50	1.00
	Incomplete	R	1.00				0.24			0.75	0.79
		P	0.98	0.68		1.00	1.00	0.27	1.00	0.70	0.37
	Commented code	R	0.98	0.62		1.00	1.00	0.25	1.00	0.58	0.41
		P	0.99	0.96	0.95	0.11	0.67	0.27	0.75	0.95	0.37
Under dev.	Under development	R	0.97	0.78	0.96	0.34	0.69	0.33	0.88	1.00	0.45

Table 5 (continued)

			Validation								
	P = Precision R = Recall			Cross project							
Top categories	Inner categories		10-fold	P1	P2	P3	P4	P5	P6	P7	P8
		P	0.94	0.00		0.78		0.00		0.71	
	Directive	R	0.92	0.00		0.53		0.00		0.42	
		P	1.00		0.90						0.88
	Formatter	R	0.84		0.55						0.68
		P	1.00	0.00	0.99	0.54	0.00	0.00	0.00	0.81	0.87
Style & IDE	Style & IDE	R	0.85	0.00	0.08	0.34	0.00	0.00	0.00	0.52	0.55
		P	1.00	0.00	0.63	0.90		0.00		1.00	1.00
	License	R	1.00	0.00	0.69	0.87		0.00		1.00	1.00
		P	1.00	0.54	1.00	1.00		1.00	1.00	1.00	
	Ownership	R	1.00	0.51	0.97	0.78		1.00	1.00	0.81	
		P	1.00	0.51	1.00	1.00	1.00	1.00	0.89	0.57	0.57
	Pointer	R	0.98	0.70	0.97	0.32	0.71	0.45	0.73	0.63	0.31
		P	1.00	1.00	0.84	0.91	0.98	1.00	0.96	0.96	0.97
Metadata	Metadata	R	0.99	0.32	0.61	0.87	0.26	0.47	0.79	0.81	0.98
		P	0.94				0.10	0.00			
	Auto generated	R	1.00				0.56	0.00			
		P	0.60		0.25						
	Unknown	R	0.99		0.20						
		P	0.70	0.78	0.28		0.00	0.00			
Discarded	Discarded	R	0.99	1.00	0.15		0.00	0.00			
Weighted average TP rate			0.95	0.75	0.68	0.73	0.77	0.70	0.78	0.76	0.71

Cross-project validation Different systems have comments describing different code artifacts and are likely to use different words and jargons. Thus, term-features working for the comments in one system may not work for others. To better test the generalizability of the results achieved by the classifier, we conduct a *cross-project validation*, as also previously proposed and tested by Bacchelli et al. (2012). In practice, cross-project validation for OSS case consists in a 6-fold cross validation, in which folds are neither stratified nor randomly taken, but correspond exactly to the different systems: In the open source case, we train the classifiers on five systems and we try to predict the classification of the comments in the remaining system. We do this six times rotating the test system. Similarly, in the industrial context we divided the dataset in eight folds corresponding to the eight industrial projects, then we used one fold as test dataset and the remaining folds to train the model. We repeated this process eight times to evaluate the performance for each project. The right-most columns (*i.e.*, 'cross-project') in Tables 4, 5, and 6 show the results by tested systems.

Cross-license validation The different development setting, *i.e.*, OSS or industrial, may have an impact on software development (Paulson et al. 2004). In line with the hypothesis of Paulson et al. (2004), we indeed found a difference in the comments usage between these two different categories of development processes. We found that open source projects have on average up to 8 blocks of comments per file, while the industrial projects decrease have an average of 2 blocks of comments per file.

Therefore, these differences during the training of a machine learning classifier can become crucial, as they may impact on the performance of the model.

To evaluate the impact of the different development setting on an automated approach to classify code comments, we define and conduct a *cross-license validation*. We define cross-license validation when the training set differs from the testing set for the license/setting of the file to which the comment pertain, *i.e.*, OSS or industrial. In our study, we conduct a *2-fold* cross-license validation, in which we train on projects from one setting (*e.g.*, OSS) and we test on projects from the other setting (*e.g.*, industrial). In this validation, we alternate OSS and industry as test and training sets. Table 7 contains the results, in terms of precision and recall, obtained by evaluating our model on the top categories. The first row represents the results obtained training the model on the OSS projects and testing it on the industrial ones, instead the second row refers to the opposite situation where we trained the model with the industrial comments and tested it with OSS ones. Even though the differences are not major (*e.g.*, 0.73 of precision for both DISCARDED categories), training the model with the open source data achieves better results on average (*e.g.*, the precision is up to 10% higher for the category UNDER DEVELOPMENT using open source training set); this result may be due to the higher number of comments in the OSS dataset or more diverse distributions of the features across the data. Overall, the within-project performance is marginally better than the cross-project one, when the training is accomplished with open-source data. Indeed, cross-project validation achieves performance above 0.73 in terms of weighted average TP rate, while within-project validation conducted only on open-source projects is up to 0.88 in terms of weighted average TP rate. Based on our experience gained through the manual classification, we argue that many comments in OSS systems are written with a different purpose than comments in closed-source projects. For example, OSS programmers rely on code comments to communicate their development strategies, vice-versa, industrial driven developers seem to rely on alternative channels to communicate with their team. This observation is also reflected the different number of comments present in the two domains as well as the different distribution across found categories. Moreover, this difference would

Table 6 Results of the classification with random forest classifier in cross-project validation

			Cross project validation													
	P = Precision R = Recall		Open source						Closed source							
Top categories	Inner categories		CDT	Guava	Guice	Hadoop	Vaadin	Spark	P1	P2	P3	P4	P5	P6	P7	P8
		P	0.95	0.85	0.93	0.65	0.79	0.74	0.84	0.69	0.89	0.88	0.84	0.85	0.75	0.93
	Summary	R	0.98	0.96	1.00	1.00	0.96	1.00	1.00	0.97	1.00	1.00	0.96	0.99	0.90	0.94
		P	0.85	0.70	1.00	0.15	0.15	1.00	1.00	0.42	1.00	1.00	1.00	0.24	0.57	
	Expand	R	0.64	0.85	1.00	0.13	0.75	0.25	1.00	1.00	1.00	1.00	1.00	0.19	0.57	
		P	0.55	0.00	1.00	0.15	0.00	0.00	1.00	0.40	1.00	1.00	0.32	0.56	0.72	0.36
	Rational	R	0.68	0.00	1.00	0.51	0.00	0.00	0.86	0.29	0.86	0.87	0.62	0.36	0.71	0.56
		P	0.99	0.85	0.79	.98	0.70	0.70	0.97	0.86	0.90	0.89	0.90	0.99	0.89	0.74
Purpose	Purpose	R	1.00	1.00	1.00	1.00	1.00	0.99	1.00	0.99	0.98	0.99	0.99	0.97	0.84	0.78
		P	0.75	0.00			0.00		1.00		1.00	1.00	1.00			
	Deprecation	R	0.81	0.00			0.00		1.00		1.00	1.00	1.00			
		P	0.88	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.84
	Usage	R	0.90	0.85	0.90	1.00	0.35	0.33	0.92	0.38	0.92	0.93	0.84	0.92	1.00	0.81
		P	0.72	1.00		0.63	1.00	1.00	1.00		1.00		1.00	0.74	0.68	1.00
	Exception	R	0.98	1.00		0.88	1.00	1.00	0.55		0.65		0.50	0.75	0.43	0.87
		P	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.74	0.96
Notice	Notice	R	0.99	0.76	0.73	0.97	0.24	0.65	0.95	0.81	0.87	0.91	0.81	0.73	0.86	0.85
		P	0.59	1.00		0.00	0.21		0.96			0.00			0.77	0.15
	TODO	R	0.50	0.65		0.00	0.32		0.67			0.00			0.67	0.08
		P	0.90				0.87					0.78			0.89	1.00
	Incomplete	R	0.88				0.73					0.63			0.10	0.25
		P	0.89	1.00			1.00		1.00		1.00	1.00	1.00	1.00		0.17
	Commented code	R	0.97	1.00			1.00		1.00		1.00	1.00	1.00	0.99		0.41
		P	0.99	1.00	0.76	0.98	1.00		1.00	1.00	1.00	1.00	1.00	1.00	0.87	1.00
Under dev.	Under development	R	0.95	0.55	0.85	0.86	0.67		0.99	0.86	0.85	0.86	0.87	0.86	0.87	0.99

Table 6 (continued)

			Cross project validation													
	P = Precision R = Recall		Open source						Closed source							
Top categories	Inner categories		CDT	Guava	Guice	Hadoop	Vaadin	Spark	P1	P2	P3	P4	P5	P6	P7	P8
		P	1.00	0.00					1.00		1.00		1.00		1.00	
	Directive	R	1.00	0.00					1.00		1.00		0.75		1.00	
		P	0.90							0.87						0.98
	Formatter	R	1.00							0.42						0.57
		P	1.00	0.00	1.00				1.00	1.00	1.00		1.00		1.00	0.99
Style & IDE	Style & IDE	R	1.00	0.00	0.79				1.00	0.89	0.89		0.82		0.90	0.84
		P	1.00	1.00	1.00	0.99	1.00	1.00	1.00	0.93	1.00		1.00		1.00	1.00
	License	R	1.00	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00		1.00		1.00	1.00
		P	1.00	1.00			1.00		1.00	1.00	1.00		0.99	1.00	1.00	
	Ownership	R	0.98	1.00			0.97		1.00	1.00	1.00		0.97	1.00	0.91	
		P	0.97	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.97	1.00	0.95	0.57	0.67
	Pointer	R	0.99	0.82	1.00	0.68	1.00	0.45	0.42	0.99	0.71	0.85	0.86	0.99	0.57	0.40
		P	1.00	1.00	0.99	0.99	0.94	0.90	1.00	0.99	0.91	0.99	0.99	0.99	1.00	0.99
Metadata	Metadata	R	0.95	0.94	0.89	0.99	1.00	1.00	1.00	1.00	0.97	0.95	0.98	0.98	0.91	0.95
		P	0.00			0.23	0.78					0.10	0.00			
	Auto generated	R	0.00			1.00	0.89					0.78	0.00			
		P	1.00							0.21						
	Unknown	R	0.40							0.17						
		P	0.38		0.55	0.00	0.10			0.00		0.00	0.00			
Discarded	Discarded	R	0.15		0.56	0.00	0.15			0.00		0.00	0.00			
Weighted average TP rate			0.90	0.92	0.90	0.98	0.76	0.86	0.94	0.94	0.95	0.91	0.83	0.90	0.86	0.88

Table 7 Results of the cross-license validation in terms of precision (P) and recall (R), using a random forest classifier

	Top categories											
	Purpose		Notice		Under development		Style & IDE		Metadata		Discarded	
Testing on:	P	R	P	R	P	R	P	R	P	R	P	R
Industrial systems	0.75	0.98	0.96	0.49	0.87	0.69	0.78	0.13	0.99	0.69	0.73	0.91
OSS systems	0.68	0.99	0.88	0.67	0.77	0.50	0.63	0.66	0.98	0.55	0.73	0.30

have an impact on the creation of new tools aimed at helping developers to increase their productivity and to improve software reliability.

Summary The values for 10-fold cross validation reported in Table 4 show accurate results (mostly above 95%) achieved for top-level categories. This means that the classifier could be used as an input for tools that analyze source code comments of the considered systems. For inner-categories, the results are lower; nevertheless, the weighted average TP rate remains 0.85. Furthermore, we do not see large effects due to the prominent class imbalance. This suggests that the amount of training data is enough for each class.

As expected, testing with cross-project validation, the classifier performance drops. However, this is a more reliable test for what to expect with JAVA comments from unseen projects. The weighted average TP rate that goes as low as 0.74. This indicates that project-specific terms are key for the classification and either an approach should start with some supervised data or more sophisticated features must be devised.

The last analysis (*i.e.*, the cross-license validation), where we divided the dataset in two parts gathering in the same dataset all projects with the same commercial license (*i.e.*, OSS and industrial projects), shows that results are higher when using open source dataset to train the model (up to 15% of PRECISION for STYLE & IDE category). Even though cross-license shows a negative performance when using industrial comments to train the model, the differences are in average below 7% for PURPOSE and METADATA categories in terms of PRECISION. In the end, DISCARDED achieves the same performance for both categories (0.73 in PRECISION). These results suggest that the proposed open source dataset may be used by both open source organization and industrial companies to categorize Java code comments. The higher performance of training on OSS may be due to the higher number of manually classified instances from the OSS projects; a further study could investigate whether a higher number of training instances from the industrial context would lead to similar results.

Finding 3: The within-project validation achieves the best performance (up to 0.95 for weighted average TP rate) compared to cross-project or cross-license validation where the performance is typically 15% lower. Although this, it remains above 0.74 in terms of weighted average TP for unseen files that may be due to the use a project-specific set of terms. In addition to differences in the terminology used in the comments, this difference may be increased by the different communication strategies used in the inspected developmental domains.

4.4 RQ4. How Does the Performance of an Automated Approach Improve by Adding Classified Comments from the System Under Classification?

The answer to our previous research question shows that it is possible to create an automatic classifier for code comments. However, when such a classifier is tested on an unseen project, it achieves lower results, compared to testing it on a project for which some of the comments are part of the training set. This is expected, since words used in the text are parts of the training features. In this research question, we investigate how many instances should one classify from an unseen system to make the classification algorithm reach higher results.

To this aim, we selected the industrial project that achieved the worst performance in cross-project validation, then, we progressively added to the training set a fixed number of manual classified comments (*i.e.*, in steps of 5 comments). For each iteration, we evaluated the performance of a Random Forest classifier and computed precision and recall.

Figure 5 shows the classifier's results by progressively including new manually classified comments. The *blue* line indicates the evolution of the precision curve by progressively adding 5 random selected manual classified comments of the subject system; the *red* line indicates the trend of the recall values. The lines show that the classifier starts from a minimum of 0.65 and 0.74 for precision and recall, respectively. This is the scenario in which no comments belonging to the unseen project are included in the training set. The maximum performance corresponds to 0.89 of precision and 0.94 of recall, and it is reached when at least 100 manually classified comments of the subject system are added to the training set. The trend shows that the performance reaches a plateau after 100 manually classified instances.

Finally, we observe that in the starting phase (left side of the chart) the performance of the model remains stably below 0.80 of precision and recall until the comments contribution is below 30 threshold; instead it improves rapidly in the interval included between 30 and 70 blocks of comments. This observation seems to indicate the presence of an optimal interval of comments that a human classifier should manually classify to boost the performance of the proposed solution for a novel project.

The investigation highlights that the performance of the proposed model can be easily and significantly increased by manually classifying a small sample of new comments (*e.g.*, in our case the manual classification of just 60 blocks of comments boosted the *prediction* of 30%). We empirically found this sample size to be between 40 and 80 block of comments, which corresponds to about 10 Java open source files (or about 3 hours of labeling efforts).

Finding 4: The improvement achieved by manual classifying a sample between 40 and 80 code comment from an unseen project is up to a 37% and 27% gain in precision and recall, respectively. This indicates that it is possible to significantly improve the accuracy of the automatic classification for an unseen project with a minor effort.

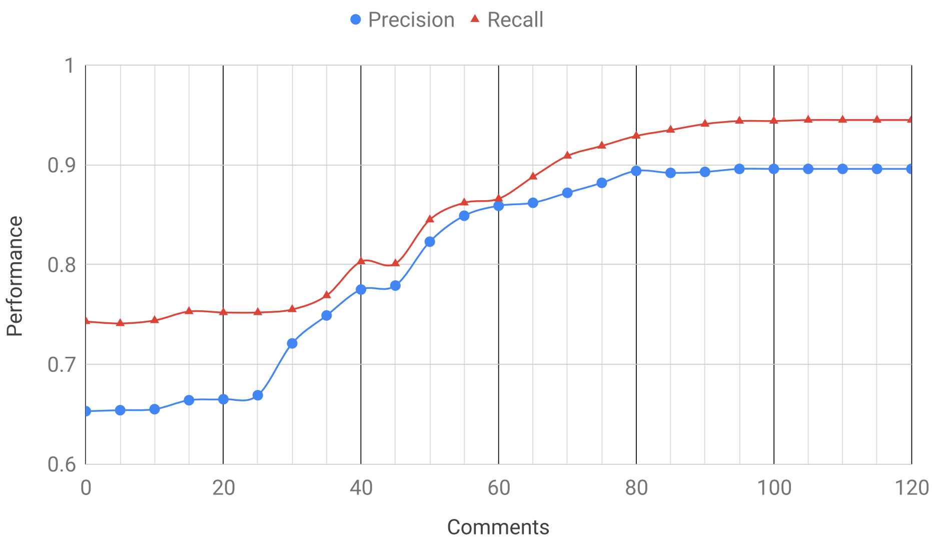

Fig. 5 Number of comments required to increase the performance for an unseen project

5 Related Work

5.1 Information Retrieval Technique

Lawrie et al. (2006) use information retrieval techniques based on cosine similarity in vector space models to assess program quality under the hypothesis that *"if the code is high quality, then the comments give a good description of the code"*. Marcus and Maletic propose a novel information retrieval techniques to automatically identify traceability links between code and documentation (Marcus and Maletic 2003). Similarly, de Lucia et al. focus on the problem of recovering traceability links between the source code and connected free text documentation. They propose a comparison between a probabilistic information retrieval model and a vector space information retrieval (Lucia and et al 2000). Even though comments are part of software documentation, previous studies on information retrieval focus generally on the relation between code and free text documentation.

5.2 Comments Classification

Several studies regarding code comments in the 80's and 90's concern the benefit of using comments for program comprehension (Woodfield et al. 1981; Tenny 1985, 1988). Stamelos et al. suggest a simple ratio metric between code and comments, with the weak hypothesis that software quality grows if the code is more commented (Stamelos et al. 2002). Similarly, Oman and Hagemeister propose a tree structure of maintainability metrics that also consider code comments (Oman and Hagemeister 1992) and Garcia et al. also use lines of comments to measure the maintainability of a module (Garcia and Granja-Alvarez 1996).

New recent studies add more emphasis to the code comments in a software project. Fluri et al. present a heuristic approach to associate comments with code investigating whether developers comment their code. Marcus and Maletic propose an approach based on information retrieval technique (Marcus et al. 2005). Maalej and Robillard investigate API reference documentation (such as javadoc) in Java SDK 6 and .NET 4.0 proposing a taxonomy of knowledge types. They use a combination of grounded and analytical approaches to create such taxonomy (Maalej and Robillard 2013). Instead Witte et al. used Semantic Web Technologies to connect software code and documentation artifacts (Witte et al. 2007). However, both approaches focus on external documentation and do not investigate evolutionary aspects or quality relationship between code and comments, *i.e.*, they do not track how documentation and source code changes together over time or the combined quality factor. More in focus is the work of Steidl et al. where they investigate the quality of the source code comments (Steidl et al. 2013a). They proposed a model for comment quality based on different comment categories and use a classification based on machine learning technique tested on Java and C/C++ programs. They define 7 high-level categories that are generically available in both Java and C/C++ programming languages. Moreover, they evaluated the quality of their taxonomy with a survey by involving 16 experienced software developers. Despite the quality of the work, they found only 7 high-level categories of comments based mostly on comment syntax, *i.e.*, inline comments, section separator comments, task comments, *etc.* This limitation may be a consequence of the aggregation of different programming languages such as Java and C/C++. In our study, we refine such categories by including a fine-grained taxonomy composed of 16 categories. Our taxonomy is tailored specifically for Java programming language, indeed, the study involved only Java sources.

Padioleau et al. (2009) also conducted an extensive evaluation and classification of source code comments. They had a different aim than ours: They focused on understanding to what extent developers' needs can be derived from code comments (*e.g.*, how comments are used for code annotations or to communicate intentions behind the software development paradigm); moreover they considered a different context (*i.e.*, code comments from three Unix-like operating systems (*i.e.*, LINUX, FREEBSD, and OPENSOLARIS) written in C). They conducted a classification along four dimensions: *content*, *people involved*, *code location*, and *time and evolution*. The 'content' dimension is the most aligned with our work. Although their focus (developers' needs and software reliability) and data sources (C systems) were different, some of their categories along the 'content' dimension share strong similarities with our taxonomy (*e.g.*, 'PastFuture' includes TODOs, as our 'C. Under Development' does). Especially if we account for the subjective differences that are common in manual classification studies, these similarities seem to indicate that it would be possible to derive a taxonomy of code comments that goes beyond the boundaries of a single programming language (indeed they also performed a preliminary investigation trying to classify comments of a Java system according to their taxonomy that further suggests this possibility Padioleau et al. 2009). Interestingly, Padioleau et al. also showed how more than 50% of the comments can be *exploited* by existing or to be proposed tools. We did not consider this aspect in our work, but future studies can be devised to investigate it using our publicly available dataset. An additional difference between our work and that of Padioleau et al. is that we studied how our manually analyzed code comments can be automatically classified according to our taxonomy using machine learning.

6 Conclusion

Code comments contain valuable information to support software development, especially during code reading and code maintenance. Nevertheless, not all the comments are the same: For accurate investigations, analyses, usages, and mining of code comments, this has to be taken into account. By recognizing different kinds of code comments, as well as different meaning brought by code comments we want to simplify developers' activities and provide better data for metrics. Following this direction, a developer may exclude comments that do not involve a specific task and focus only on a subset (*e.g.*, a developer that plans a maintainability task may be not interested in comments related to metadata information such as license and ownership). Relying on our findings, a developer may dynamically choose which categories highlight during a specific development phase. Moreover, our classification system can be used to improve techniques that use comments as a way to measure the maintainability of a software system. In fact, in this work, we investigated how comments can be categorized, also proposing an approach for their automatic classification.

The contributions of our work are:

- A novel, empirically validated, hierarchical taxonomy of code comments for Java projects, comprising 16 inner categories and 6 top categories.
- An assessment of the relative frequency of comment categories in 6 OSS Java software systems.
- An estimation of the relative frequency of comment categories in 8 industrial Java software systems.
- An exhaustiveness by cross-license validation of proposed taxonomy in a different context.
- A publicly available dataset of more than 2,000 source code files with manually classified comments, also linked to the source code entities they refer to.

- An empirical evaluation of an enhanced machine learning approach to automatically classify code comments according to the aforementioned taxonomy.
- An empirical evaluation aimed at understanding how many instances should be manually classified from an unseen system to make the classification algorithm perform similarly to an already seen project.

Acknowledgments This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 642954. Bacchelli gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

References

Apache Spark (2016) http://spark.apache.org. [Online; accessed 03-Feb-2016]

Apache Software Foundation (ASF) - Apache Hadoop software library (2017). http://hadoop.apache.org/. [Online; accessed 10-02-2017]

Bacchelli A, Lanza M, Robbes R (2010) Linking e-mails and source code artifacts. In: Proceedings of ICSE 2010 (32nd International Conference on Software Engineering). ACM Press, pp 375–384

Bacchelli A, dal Sasso T, D'Ambros M, Lanza M (2012) Content classification of development emails. In: Inproceedings of ICSE 2012 (34th ACM/IEEE International Conference on Software Engineering), pp 375–385

Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) Smote: synthetic minority over-sampling technique. Journal of artificial intelligence research 16:321–357

Chawla NV (2009) Data mining for imbalanced datasets: An overview. In: Data mining and knowledge discovery handbook. Springer, pp 875–886

de Souza SCB, Anquetil N, de Oliveira KM (2005) A study of the documentation essential to software maintenance. In: Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information. ACM, pp 68–75

Diakopoulos N, Cass S (2016) The top programming languages IEEE Spectrum, vol http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016

Fleiss JL (1971) Measuring nominal scale agreement among many raters. Psychological Bulletin 76(5):378

Fluri B, Wursch M, Gall HC (2007) Do code and comments co-evolve? on the relation between source code and comment changes. In: 2007. WCRE 2007. 14th Working Conference on Reverse Engineering. IEEE, pp 70–79

Flyvbjerg B (2006) Five misunderstandings about case-study research. Qualitative Inquiry 12(2):219–245

Friedman J, Hastie T, Tibshirani R (2001) The elements of statistical learning, vol 1. Springer series in statistics Springer, Berlin

Garcia MJB, Granja-Alvarez JC (1996) Maintainability as a key factor in maintenance productivity: a case study. In: icsm, pp 87

Goodfellow I, Bengio Y, Courville A (2016) Deep Learning. Adaptive Computation and Machine Learning series. The MIT Press, Cambridge

Haouari D, Sahraoui H, Langlais P (2011) How good is your comment? a study of comments in java programs. In: 2011 International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, pp 137–146

Hartzman CS, Austin CF (1993) Maintenance productivity: Observations based on an experience in a large system environment, pp 138–170

Jiang ZM, Hassan AE (2006) Examining the evolution of code comments in postgresql. In: Proceedings of the International Workshop on Mining Software Repositories, MSR '06. ACM, New York, pp 179–180

Lawrie DJ, Feild H, Binkley D (2006) Leveraged quality assessment using information retrieval techniques. In: 14th IEEE International Conference on Program Comprehension ICPC 2006. IEEE , pp. 149–158

Lidwell W, Holden K, Butler J (2010) Universal Principles of Design, Revised and Updated: 125 Ways to Enhance Usability. Influence Perception. Increase Appeal, Make Better Design Decisions, and Teach through Design. Rockport Publishers, 2nd ed.

Lucia D et al (2000) Information retrieval models for recovering traceability links between code and documentation. In: 2000. Proceedings. International Conference on Software Maintenance. IEEE, pp 40–49

Maalej W, Robillard MP (2013) Patterns of knowledge in api reference documentation. IEEE Trans Softw Eng 39:1264–1282

Manning CD, Raghavan P, Schütze H et al (2008) Introduction to information retrieval. Cambridge University Press, Cambridge, vol 1

Marcus A, Maletic JI (2003) Recovering documentation-to-source-code traceability links using latent semantic indexing. In: 25th International Conference on Software Engineering Proceedings. IEEE, pp 125–135

Marcus A, Maletic JI, Sergeyev A (2005) Recovery of traceability links between software documentation and source code. Int J Softw Eng Knowl Eng 15(5):811–836

O'brien RM (2007) A caution regarding rules of thumb for variance inflation factors. Quality & Quantity 41(5):673–690

Oman P, Hagemeister J (1992) Metrics for assessing a software system's maintainability. In: Conference on Software Maintenance Proceerdings. IEEE, pp 337–344

Padioleau Y, Tan L, Zhou Y (2009) Listening to programmers taxonomies and characteristics of comments in operating system code. In: Proceedings of the 31st International Conference on Software Engineering, pp 331–341, IEEE Computer Society

Pascarella L, Bacchelli A (2017) Manually classified dataset of source code comments. https://doi.org/10.5281/zenodo.2628361

Paulson JW, Succi G, Eberlein A (2004) An empirical study of open-source and closed-source software products. IEEE Trans Softw Eng 30(4):246–256

Sebastiani F (2002) Machine learning in automated text categorization. ACM Computing Surveys (CSUR) 34(1):1–47

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

Stamelos I, Angelis L, Oikonomou A, Bleris GL (2002) Code quality analysis in open source software development. Inf Syst J 12(1):43–60

Steidl D., Hummel B, Jürgens E (2013a) Quality analysis of source code comments. In: IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, pp 83–92

Steidl D, Hummel B, Juergens E (2013b) Quality analysis of source code comments. In: IEEE 21st International Conference on Program Comprehension (ICPC). IEEE, pp 83–92

Tan L, Yuan D, Krishna G, Zhou Y (2007) /* icomment: Bugs or bad comments?*. In: ACM SIGOPS Operating Systems Review. ACM, vol 41, pp 145–158

Tenny T (1985) Procedures and comments vs. the banker's algorithm. SIGCSE Bull 17:44–53

Tenny T (1988) Program readability: Procedures versus comments. IEEE Trans Softw Eng 14(9):1271–1279

Triola MF (2006) Elementary statistics, 10th edn. Pearson/Addison-Wesley, Reading

Witte R, Zhang Y, Rilling J (2007) Empowering software maintainers with semantic web technologies. In: The Semantic Web: Research and Applications, 4th European Semantic Web Conference, ESWC 2007. Proceedings, Innsbruck, pp 37–52

Woodfield SN, Dunsmore HE, Shen VY (1981) The effect of modularization and comments on program comprehension, pp 215–223

Luca Pascarella is a Ph.D. student at the Delft University of Technology, The Netherlands. He is also collaborating as a researcher at the Software Improvement Group (SIG), in Amsterdam. He received his B.Sc. and M.Sc. in Software Engineering from the University of Sannio, Italy. Since he started his research career in 2016, he focused on ancillary aspects of the source code such as code comments to understand their role in software quality. His research aims at improving modern code review tools with augmented defect prediction techniques.

Magiel Bruntink is Head of Research of the Software Improvement Group (SIG), a consultancy that focuses on the automatic analysis of software quality and related decision making. He has a scientific background in program analysis and empirical study. Most of his work consists of mixed industry-academic projects, positioned within the software engineering domain.

Alberto Bacchelli is an SNSF Professor in Empirical Software Engineering in the Department of Informatics in the Faculty of Business, Economics and Informatics at the University of Zurich, Switzerland. He received his B.Sc. and M.Sc. in Computer Science from the University of Bologna, Italy, and the Ph.D. in Software Engineering from the Università della Svizzera Italiana, Switzerland. Before joining the University of Zurich, he has been assistant professor at Delft University of Technology, The Netherlands where he was also granted tenure. His research interests include peer code review, empirical studies, and the fundamentals of software analytics.