

# What happens in my code reviews? An investigation on automatically classifying review changes

Enrico Fregnan · Fernando Petrulio ·  
Linda Di Geronimo · Alberto Bacchelli

Received: date / Accepted: date

**Abstract** Code reviewing is a widespread practice used by software engineers to maintain high code quality. To date, the knowledge on the effect of code review on source code is still limited. Some studies have addressed this problem by classifying the types of changes that take place during the review process (*a.k.a. review changes*), as this strategy can, for example, pinpoint the immediate effect of reviews on code. Nevertheless, this classification (1) is not scalable, as it was conducted manually, and (2) was not assessed in terms of how meaningful the provided information is for practitioners. This paper aims at addressing these limitations: First, we investigate to what extent a machine learning-based technique can automatically classify review changes. Then, we evaluate the relevance of information on review change types and its potential usefulness, by conducting (1) semi-structured interviews with 12 developers and (2) a qualitative study with 17 developers, who are asked to assess reports on the review changes of their project. Key results of the study show that not only it is possible to automatically classify code review changes, but this information is also perceived by practitioners as valuable to improve the code review process. Data and materials: <https://doi.org/10.5281/zenodo.5592254>

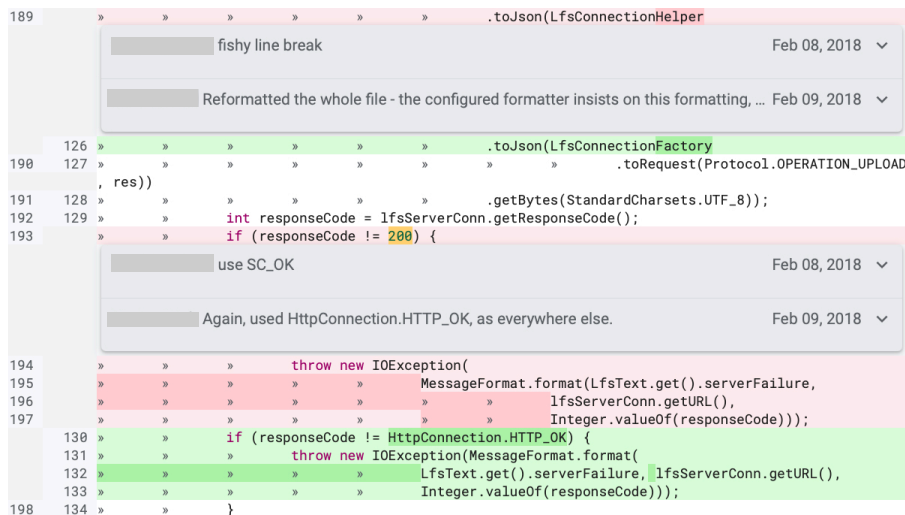
**Keywords** Modern Code Review · Empirical Software Engineering · Review Changes · Automatic Classification.

## 1 Introduction

Contemporary peer code review is a lightweight and informal process in which developers (*a.k.a.*, the reviewers) judge the quality of a newly proposed set of code changes, using a dedicated software tool [11].

---

Enrico Fregnan, Fernando Petrulio, Linda Di Geronimo, Alberto Bacchelli  
University of Zurich, Switzerland  
E-mail: fregnan@ifi.uzh.ch, fpetrulio@ifi.uzh.ch, digeronimo@ifi.uzh.ch, bacchelli@ifi.uzh.ch



**Fig. 1** Example of changes induced by a code review in JGit. For instance, at line 193 the reviewer asks to substitute the *magic number* "200" with a variable with a meaningful name. ([https://git.eclipse.org/r/c/jgit/jgit/+/111364/20..21/org.eclipse.jgit.lfs/src/org/eclipse/jgit/lfs/LfsPrePushHook.java](https://git.eclipse.org/r/c/jgit/jgit/+/)).

Despite the substantial practical adoption of code review [76], the knowledge on its effects—and how to measure them—is still limited. For instance, how can benefits such as knowledge transfer among developers be quantitatively assessed? Past research (*e.g.*, [8, 27]) found a strong mismatch between what developers and managers *think to obtain* from code review (*i.e.*, catching important defects early) vs. what they *actually obtain* (*i.e.*, catching very few, localized, low-level defects).

This mismatch is not surprising: Assessing the effect of the code review process is a complex task and existing evaluation tools provide basic information, at most. Nevertheless, a number of empirical software engineering studies (*e.g.*, [8, 17, 46, 58, 61, 62, 89, 91]) did manage to devise approaches able to assess certain aspects of the effect of code review and have provided valuable information to researchers on the topic.

In particular, three studies [8, 17, 58] looked at the code review process from a new perspective that investigates *review changes*: the changes to the code that are implemented at review time (Figure 1 shows an example of these review-induced changes).

Under this perspective, these studies could (1) determine that most code reviews do not affect the functionalities of source code [8, 17, 58], (2) find the aforementioned mismatch between expectations and outcome in code review [8], and (3) lay out a number of other, previously undocumented, effects of code review [8]. The key idea behind this perspective is an approach that evaluates the code review process by *classifying the types of review changes*. This approach answers questions such as: “Are most review changes fixing

functional defects or improving other aspects of the code?” “What kind of defects are found most frequently during review?”

Two critical advantages of studying review changes are: (1) It pinpoints exactly the immediate effect of code review on code and (2) it is not influenced by the external confounding factors (*e.g.*, change-proneness of artifacts or adjustments in the development process) affecting other long-term metrics, such as defect proneness [61].

In recent years, researchers focused on devising tools that leverage data from software development practices [10, 18, 96]. In particular, researchers at Microsoft developed a tool, *CodeFlow Analytics* (CFA), to collect and display to developers information about the code review process [19]. CFA was created to answer development teams’ requests to easily analyze and monitor their code review data. Bird et al. reported how CFA (1) successfully enabled development teams at Microsoft to monitor themselves and improve their processes and (2) having code review data at disposal stimulated further research to improve the code review process.

Based on the good welcome received by CFA, we believe that integrating information on review changes in a similar tool might be the next natural step to allow development teams to gain further insights in their code review process. This information could drive the team to reflect on the code review practices they have in place. For instance, if the majority of changes are documentation changes could not these be caught earlier in the development phase (*e.g.*, using static analysis tools or improving the style guidelines of the project). This could allow developers to allocate more time in looking for functional defects, critical for the success of the project.

Nevertheless, the application of this approach to help developers to assess their review process is currently limited by two factors. First, the review-induced changes have to be manually classified: This aspect clearly impacts the scalability of the approach, as it can not easily be applied at large. Second, the target of such effort in classifying review changes has mostly been the research community and it is still unclear whether and how its output is meaningful to practitioners.

In this paper, we present a two-step empirical investigation we conducted to address these limitations. First, we investigate whether review changes can be automatically classified using a supervised machine learning approach; this with the goal of solving the scalability issue. To this end, (1) we manually classify 1,504 review changes using Beller et al.’s taxonomy [17]; (2) we select 30 features based on the analysis of prior work [23, 34] as well as the insight acquired constructing the dataset; (3) we evaluate three machine learning algorithms to automatically classify review changes at two different levels of granularity. Our results show that the evaluated solution classifies code changes with an AUC-ROC beyond 0.91. This finding provides evidence that machine learning can be effectively employed to classify review-induced changes.

In the second part of our investigation, we focus on collecting feedback from developers. Our goal is two-fold: Analyzing developers’ perception concerning the classification of review changes as mean to support review practices as well

as evaluating our approach in a real-world scenario. To this aim, (1) we interviewed 12 developers with experience in code review, and (2) we sent reports generated with our approach to 20 open-source projects, receiving feedback from 17 developers. Practitioners positively assessed the novelty of the information on review changes and gave an initial indication of the potential usefulness of this information, particularly for project managers interested in improving the code review process.

## 2 Background and Related Work

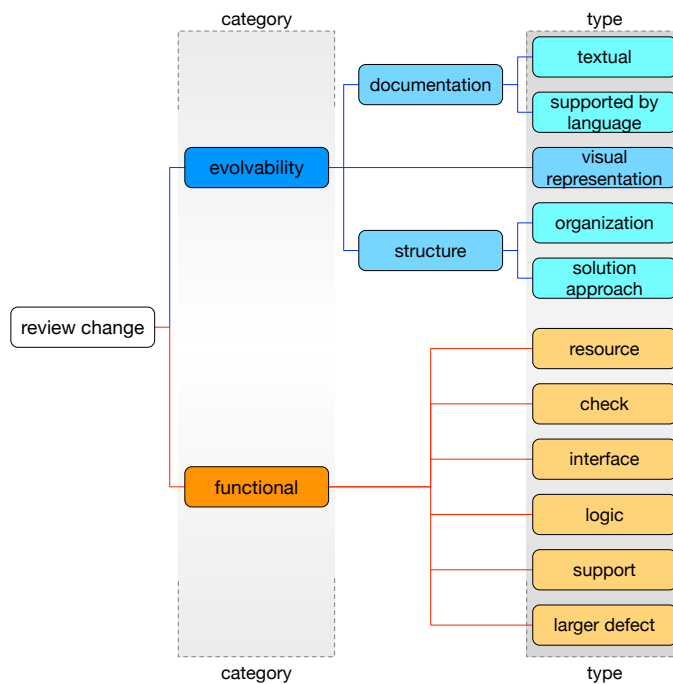
The topic of code inspection [32] has been largely explored in the past [12, 70, 95] and, in the last decade, both research community and practitioners switched to the more practical concept of modern code review [8]. In this respect, researchers have been investigating methodologies and techniques to provide support for developers during the code review process. For instance, researchers have devised automated solutions to identify proper reviewers for a new code change [49, 64, 90, 98], as well as investigated both the factors making the process more/less effective [13, 15, 21, 47, 60, 74, 83, 91] and the actual influence of code review on the resulting software quality [5, 46, 61, 62, 71, 76, 77, 81, 84, 93].

In the context of our work, we are mainly interested in *review-induced changes* (or simply *review changes*). These are the changes to which the code undergoes as result of the review process: *e.g.*, as consequence of a reviewer’s comment (as shown in Figure 1). However, changes to the code can also be triggered by other factors: *e.g.*, a spontaneous action from the code author or informal discussions among developers. In this paper, our main goal is to overcome the current limitations to classify review changes and evaluating how this data can be used to support developers during code review. For this reason, in this section we focus on the discussion of previous papers that previously faced the problem of classifying review changes. Surveying the literature, we found three main works.

In the first place, Bacchelli and Bird [8] analyzed the code review process in place at MICROSOFT, revealing a number of underestimated goals and benefits: among them, they pointed out that knowledge transfer and awareness of design solutions among team members are key aspects of modern code review. Bacchelli and Bird [8] also highlighted that the main challenge for practitioners while performing code review is the comprehension of the newly committed code changes. This finding motivates our work, as it suggests the need for solutions able to help in classifying and showing to reviewers which changes have been performed.

Other papers aimed at characterizing which defects can be identified during code review. In this direction, Mäntylä and Lassenius [58] were the first to empirically explore the types of defects discovered in code review sessions as well as their distribution in both industrial and academy contexts. To this aim, the authors assessed the outcome of over 100 code review sessions and defined a taxonomy of code changes. Their main finding reports that 75% of





**Fig. 2** Taxonomy [17] for review-induced changes with their *categories/types*.

the inspected reviews identify defects that do not affect the external behavior of source code.

A further step toward this research direction has been made by Beller *et al.* [17], who further explored what changes are performed in code reviews of open-source systems, with the aim of investigating what the real benefits provided by modern code review are. They considered over 1,400 code changes, manually labeling them using the taxonomy of Mäntylä and Lassenius [58], whenever possible. As a result, they came up with an updated taxonomy of code changes done in code review, which is shown in Figure 2.

**Taxonomy of review changes.** The taxonomy of Beller *et al.* presents two main categories, namely (1) *Evolvability* changes, which are related to all the modifications that do not have an effect of the functionalities implemented in the source code (e.g., those targeting documentation and refactoring) and (2) *Functional* changes, which are those that change the structure of the code, thus impacting on the inner-work of a program. These two main categories were then refined to report the specific types of evolvability and functional changes. A description of each type of change, together with an example from a real-world scenario, is available in Section 2.1 and Section 2.2. Moreover, each type can be further divided into subtypes. Two tables offering a complete

description of changes types and subtypes, respectively, can be found in our replication package [35].

Our work is clearly inspired by the paper by Beller et al. [17] and aims at studying deeper the value of their findings: We investigate a technique to overcome the scalability limitations of their classification approach and assess with developers the value such information on review changes.

## 2.1 Evolvability changes

Evolvability changes can be defined as maintainability changes, which, therefore, do not have an impact on the functionalities of a system. Based on the taxonomy by Beller et al. [17], they can be divided in five types:

**Textual:** Textual changes concern the use of textual information in the code. In particular, they impact logging messages, adding or updating comments, or renaming variables to be more consistent with the environment.

```

255 255  /**
256 256  * Create a primary index for the current bucket.
257 257  *
258 258  * A {@link CouchbaseException} can be propagated if the index already exists and ignoreIfExists
    is set to false.
    can/will ? (same wording with the other blocks) Dec 16, 2015
    done Dec 16, 2015
259 259  * The {@link Observable} can error under the following conditions:
260 260  * - {@link CouchbaseException} if the index already exists and ignoreIfExists is set to false.
261 261  *
  
```

**Fig. 3** Example of Textual change, where a comment was rephrased for consistency with similar comments. The real-world code review in which this change took place is available online.

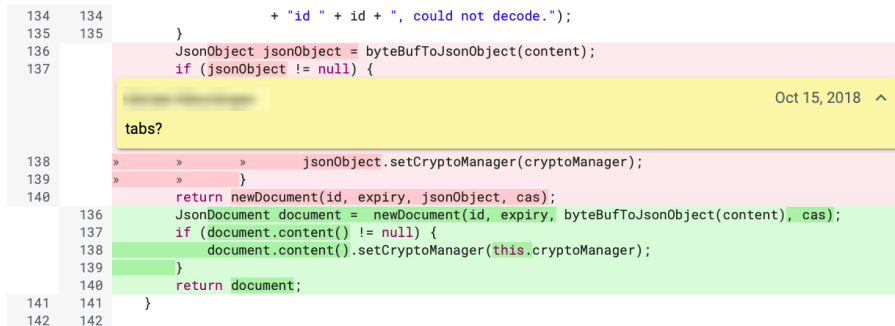
**Supported by language:** Supported by language changes impact programming language-specific features used to convey information: *e.g.*, the use of the *final* or *static* keywords in Java.

```

25 28 */
26 26 public class FtsServerOverloadException extends CouchbaseException {
    Does it make sense for this to extend `TemporaryFailureException`? Sep 25, 2018
27 27 public class FtsServerOverloadException extends TemporaryFailureException {
28 28     public FtsServerOverloadException(String payload) {
29 29         super("Search server is overloaded. Details: " + payload);
30 30     }
  
```

**Fig. 4** Example of Supported by language change, where the class that extends *FtsServerOverloadException* has been changed from *CouchbaseException* to *TemporaryFailureException*. The real-world code review in which this change took place is available online.

**Visual representation:** Visual representation changes are related to the style of the code: *e.g.*, they concern the wrong use of indentation in the code or the presence of unnecessary new lines and whitespaces.



```

134 134         + "id " + id + ", could not decode.");
135 135     }
136     JSONObject jsonObject = byteBufToJsonObject(content);
137     if (jsonObject != null) {
138         jsonObject.setCryptoManager(cryptoManager);
139     }
140     return newDocument(id, expiry, jsonObject, cas);
136     JsonDocument document = newDocument(id, expiry, byteBufToJsonObject(content), cas);
137     if (document.content() != null) {
138         document.content().setCryptoManager(this.cryptoManager);
139     }
140     return document;
141 }
142

```

**Fig. 5** Example of Visual representation change where a reviewer requested to change the code indentation. The real-world code review in which this change took place is available online.

**Organization:** Organization changes involve a re-arrangement of the code. For instance, an organizational change can remove unused portions of code or move a method into a different class to improve the structure of the project.



```

245 247     }
246
247     /**
248     * Infinitely loop processing IO.
249     */
250     @Override
251     public void run() {
252         while (running) {
253             if (!reconfiguring) {
254                 try {
255                     handleIO();
256                 } catch (IllegalStateException e) {
257                     logRunException(e);
258                 } catch (Exception e) {
259                     logRunException(e);
260                 }
261             }
262         }
263         getLogger().info("Shut down Couchbase client");
264     }
265
266     private void logRunException(Exception e) {
267         if (shutDown) {
268             // There are a couple types of errors that occur during the
269             // shutdown sequence that are considered OK. Log at debug.
270             getLogger().debug("Exception occurred during shutdown", e);
271         } else {
272             getLogger().warn("Problem handling Couchbase IO", e);
273         }
274     }
275 248 }
276 249

```

**Fig. 6** Example of Organization change where a portion of dead code has been removed. The real-world code review in which this change took place is available online.

**Solution approach:** Solution approach changes modify the way in which a system's functionality is implemented, without modifying the functionality itself: *e.g.*, the removal of magic numbers in favor of predefined constants.

```

278 279 @Test
279 280 public void testQuerySetGroupNoReduce() throws Exception {
280 281     Query query = new Query();
281 282     query.setGroup(true);
282 283     View view = client.getView(DSIGN_DOC_WO_REDUCE, VIEW_NAME_WO_REDUCE);
283 284     HttpFuture<ViewResponse> future = client.asyncQuery(view, query);
284 285     try {
285 286         ViewResponse response = future.get();
286 287         assert response != null: "Response was null";
287 288     } catch (ExecutionException e) {
288 289         assert true;
289 290     }
290 291     }
291 292     }

```

In this case, you should just do a fail("message") since you're catching the e... Jul 12, 2012 ▾

**Fig. 7** Example of Solution approach change. In this example the *assert* instructions has been changed for a more clear alternative without altering its functionality. The real-world code review in which this change took place is available online.

## 2.2 Functional changes

Functional changes are modifications that impact the functionality of a system to ensure the correct flow of the program. Based on the taxonomy proposed by Beller et al. [17], functional changes can be divided in six types:

**Resource:** Resource changes impact how the data are handled. They typically happen to solve issues in how the code manages data.

```

340 340 /**
341 341  * Remove the bucket (if it was adhoc) and disconnect from the cluster.
342 342  */
343 343 public void destroyBucketAndDisconnect() {
344 344     if (isAdHoc) {
345 345         bucket.close();
346 346         clusterManager.removeBucket(bucketName);
347 347     }
348 348     disconnect();
349 349 }
350 350 }

```

**Fig. 8** Example of Resource change where the data buffer is closed in order to avoid unexpected results in the computation. The real-world code review in which this change took place is available online.

**Check:** Check changes are introduced to verify conditions of variables and functions left previously unchecked (possibly leading to run-time errors).

```

34 36     public HashJoinHintElement(HashSide side) {
35 37         Objects.requireNonNull(side);
36 38         this.side = side;
37
38
39 }

```

NIT: requireNonNull to document and enforce the non-nullability of this field? Dec 06, 2018 ▾

Done Dec 06, 2018 ▾

**Fig. 9** Example of Check change: A check is introduced to enforce that the object *side* is not null. The real-world code review in which this change took place is available online.

**Interface:** Interface changes impact the way the code interacts with other parts of the system: *e.g.*, the use of incorrect parameters in a function call.

```

542 542 » » » public Heap(final int estimatedSize, final int limit) {
543 543 » » »     super(limit, estimatedSize);
544 544 » » » }

```

wrong arg order Mar 18, 2015 ▾

Done Mar 18, 2015 ▲

Sadly the test should have caused IAE in this case (est > limit), but the order in the test was wrong too :(

```

543 543 » » »     super(estimatedSize, limit);
544 544 » » » }

```

**Fig. 10** Example of Interface change that solve an issue with the incorrect order of parameters. The real-world code review in which this change took place is available online.

**Logic:** Logic changes solve issues in the logic of the system: for instance, when an algorithm produces incorrect results.

```

187 187     @Override
188 188     public Boolean call(GetDesignDocumentResponse response) {
189 189         if (response.status() == ResponseStatus.NOT_EXISTS) {
190
191             throw new DesignDocumentDoesNotExistException();
192
193         boolean success = response.status().isSuccess();
194         if (!success) {
195             if (response.content() != null && response.content().refCnt() > 0) {
196                 response.content().release();
197             }
198         }
199         if (response.status() == ResponseStatus.NOT_EXISTS) {
200             throw new DesignDocumentDoesNotExistException();
201         }
202         return success;
203     }

```

this block needs to go before the return block since right now it will not release() the content buffer Dec 28, 2016 ▲

**Fig. 11** Example of Logic change. In the given example, a check on the response status needs to be moved to allow a correct release of the content buffer. The real-world code review in which this change took place is available online.

**Support:** Support changes solve issues related to support libraries (or systems) and their configuration: *e.g.*, defects caused by the use of a wrong version of an external library.

```

333 333  static const char* const kLibcoreModules[] = {
334 334      "core-oj",
335 335      "core-libart",
336 336      "core-simple",
337 337      "conscrypt",
338 338      "okhttp",
339 339      "bouncycastle",
337 340  };

```

**Fig. 12** Example of Support change, where new modules are added to the configuration settings to improve the security and stability of the system. The real-world code review in which this change took place is available online.

**Larger defect:** Larger defects changes are modifications that address major issues in the code, often spanning across several classes: for instance, when a class is left partly implemented.

```

261 258      decrypted = provider.decrypt(encrypted);
262 259  } catch (Exception ex) { }
259 260      String decryptedString = new String((byte[]) decrypted);
260 261      if (type.compareTo(String.class.getSimpleName()) == 0) {
261 262          return decryptedString;
262 263      } else if (type.compareTo(Boolean.class.getSimpleName()) == 0) {
263 264          return Boolean.parseBoolean(decryptedString);
264 265      } else if (type.compareTo(Integer.class.getSimpleName()) == 0) {
265 266          return Integer.parseInt(decryptedString);
266 267      } else if (type.compareTo(Long.class.getSimpleName()) == 0) {
267 268          return Long.parseLong(decryptedString);
268 269      } else if (type.compareTo(Double.class.getSimpleName()) == 0) {
269 270          return Double.parseDouble(decryptedString);
270 271      } else if (type.compareTo(JsonObject.class.getSimpleName()) == 0) {
271 272          return JsonObject.fromJson(decryptedString);
272 273      } else if (type.compareTo(JsonArray.class.getSimpleName()) == 0) {
273 274          return JsonArray.fromJson(decryptedString);
274 275      } else if (type.compareTo(Map.class.getSimpleName()) == 0) {
275 276          return JsonObject.fromJson(decryptedString).toString();
276 277      } else if (type.compareTo(Number.class.getSimpleName()) == 0) {
277 278          try {
278 279              return NumberFormat.getInstance().parse(decryptedString);
279 280          } catch (Exception ex) {
280 281              return null;
281 282          }
282 283      } else if (type.compareTo(List.class.getSimpleName()) == 0) {
283 284          return JsonArray.from(decryptedString).toList();
284 285      } else if (type.compareTo("null") == 0) {
285 286          return null;
286 287      }
287 288  } catch (Exception ex) {
288 289  }
263 289  return decrypted;

```

**Fig. 13** Example of Larger defect change where an entire missing piece of code was added to the method. The real-world code review in which this change took place is available online.

### 3 Research Questions

The *goal* of this study is to understand (1) whether it is possible to devise an approach to automatically classify code review changes and (2) to what extent developers perceive information about review changes as useful (*e.g.*, because these changes raise awareness about the outcome of the code review process in their projects). The *perspective* is of both researchers and practitioners. The first are interested in exploiting this new source of information to build analytics tools and devise new strategies to further support developers, while the latter aim to use this information to improve their processes and the overall quality of their software systems.

We start our investigation by focusing on whether a machine learning-based solution can automatically classify code review changes into the taxonomy proposed in prior work [17, 58]. A positive outcome would give us confidence that it is possible to overcome the scalability issues of this approach [8, 17, 58]. Hence, we ask:

**RQ<sub>1</sub>:** *How accurately can a machine learning approach classify review changes?*

To the best of our knowledge, no previous attempts have been made to devise approaches to automatically classify review changes. For this reason, in **RQ<sub>1</sub>** we could not compare our results with other existing approaches. In our investigation, we evaluated our approach in terms of AUC-ROC, using as baseline a random classifier. Having achieved promising results with the automated approach, we proceed to evaluate how information on review changes can be relevant for practitioners. To gather in-depth knowledge of the developers' perception of the meaningfulness of our approach as well as the usefulness of classifying review changes for review practices, we devise a two-step investigation. First, we conduct semi-structured interviews with developers to gather their opinions on data on types of review changes as a mean to support code review practices (*e.g.*, as a way to measure the effects of code review) and on the relevance of the taxonomy used for the classification. We ask:

**RQ<sub>2.1</sub>:** *How do developers perceive data on review changes and its use to support review practices?*

Informed by the feedback gathered during the interviews, we devise reports based on review change data and investigate how experienced developers from open-source software projects perceive these reports. Therefore, our last research question is:

**RQ<sub>2.2</sub>:** *How do project maintainers evaluate reports presenting review change data for their projects?*

Our study features a *mixed-method* research approach [42] that includes (1) quantitative analysis on the performance of the investigated automated approach [29], (2) semi-structured interviews [55], and (3) customized surveys with developers from open-source software systems [33]. The next sections describe methodology and results for each research question.

#### 4 RQ<sub>1</sub>: Classifying Types of Review Changes

The first necessary step to use information about the types of review changes as a mean to support practitioners is to overcome the existing scalability issues (*i.e.*, review changes have to be manually classified [8, 17, 58]). Here, we investigate an approach to *automatically* classify review changes and we evaluate its accuracy. In this study, we limit ourselves to the case of code changes in Java source code.

##### 4.1 The Evaluated Approach

We design the proposed review changes classifier as a machine learning-based solution (available in our replication package [35]). This choice is motivated by three main factors: (1) the amount of code review data available (*e.g.*, in projects using GERRIT [2]) that make supervised approaches practically suitable, (2) the restrictions of heuristics-based techniques, which may lead to approaches limited to certain programming languages, and (3) the ability of machine learning classifiers to identify the most suitable features to use and learn from previous changes applied by developers. To investigate the accuracy of a machine-learning based approach, we make the following steps:

**Dataset Selection.** To select the projects to build our review changes dataset, we start from CROP [65], a publicly available and curated code review dataset. CROP contains projects that use GERRIT as code review platform and those review data are publicly available. From the two communities (Eclipse and Couchbase) represented in CROP, we select JGIT and JAVA-CLIENT. These projects are (1) Java-based and (2) large open-source systems (ranging from 85k to 145k lines of code). Furthermore, the selected projects possess an active community of developers and reviewers, which makes them ideal targets of our investigation. We limited our selection to two projects from the CROP dataset to ensure that our classification covers a number of cases sufficient to report even the occurrences of uncommon types of changes. On the contrary, covering less cases from a more vast selection of projects might introduce bias in the types of changes contained in our dataset, under-reporting changes that occur less frequently. Moreover, we also consider ANDROID, one of the most widely used projects in code review research. ANDROID has a long development history with many active developers and reviewers and has been shown to be highly representative the code review practices of open-source projects [14, 60, 61, 68].



**Dataset Labeling.** To build and evaluate a supervised machine learning approach, we need reliable information on the actual labels to assign to the considered modifications (*i.e.*, an oracle of code change types). With the labeled data, the approach learns from a subset of the labeled changes and can be tested on the remaining ones. To this aim, the first author of this paper manually classified the changes according to the taxonomy proposed by Beller *et al.* [17], relying on (1) the source code of each modification and (2) the code comments left by developers on GERRIT. To each change was assigned both a *category* (Evolvability or functional) and a *type*. A description of each type is provided in Section 2. These changes were selected randomly choosing review IDs (*i.e.*, a unique identifier assigned to a review in Gerrit) among all merged reviews in the considered project. Then, we performed a second step randomly choosing a patch set among the ones contained in the review. We excluded from this selection the first patch since we were not interested in changes done before the beginning of the review process. The labeling process took place between March 2019 and August 2019.

**Labeling Granularity.** While analyzing the labeled dataset, we noticed that types of changes, such as *Larger defect* or *Support* are very uncommon. This observation was confirmed by previous studies: In their analysis Mäntylä and Lassenius found only eleven changes belonging to the larger defect type and no changes belonging to the support type [58]. Similar results have been obtained also by Beller *et al.* [17]. Moreover, the overall amount of functional changes is significantly lower compared to the number of evolvability changes: Only 6.89% of all changes in our dataset belong to the functional category. This might significantly impact the performance of the devised classification approach as it would not have enough instances of each type of change to be able to make accurate prediction at *type-level*. For this reason, we argue that considering all eleven *type* labels might negatively impact the performance of our machine-learning approach.

Restricting our classification approach to work only at *category-level* might also not be an effective solution: We argue that the information offered by the categories might be too coarse-grained to offer valuable insight in the code review process. For these reasons, we devise the concept of *group* as an intermediate level of granularity for labels between categories and types. We define changes *groups* as follows. Referring to the taxonomy presented in Figure 2, we aggregate types into their corresponding higher classes: ‘textual’ and ‘supported by language’ changes were considered as ‘documentation’ changes, while ‘organization’ and ‘solution approach’ changes as ‘structure’ ones. Therefore, a change group can be one of the following: ‘documentation’, ‘visual representation’, ‘structure’, or ‘functional’. Based on this consideration, we assigned to each change in our dataset an additional label reflecting the *group* to which it belongs. Then, we tested the performance of our model at both category and group level.

```

1041 > /**
1042 > * Receive a list of commands from the input.
1043 > *
1044 > * @throws IOException
1045 > */
1046 > protected void recvCommands() throws IOException {
1047 >     for (;;) {
1048 >         String line;
1049 >         try {
1050 >             line = pckIn.readStringRaw();
1051 >         } catch (EOFException eof) {
1052 >             if (commands.isEmpty())
1053 >                 return;
1054 >             throw eof;
1055 >         }
1056 >         if (line == PacketLineIn.END)
1057 >             break;
1041 > /**
1042 > * Receive a list of commands from the input.
1043 > *
1044 > * @throws IOException
1045 > */
1046 > protected void recvCommands() throws IOException {
1047 >     for (;;) {
1048 >         String rawLine;
1049 >         try {
1050 >             rawLine = pckIn.readStringRaw();
1051 >         } catch (EOFException eof) {
1052 >             if (commands.isEmpty())
1053 >                 return;
1054 >             throw eof;
1055 >         }
1056 >         if (rawLine == PacketLineIn.END) {
1057 >             break;

```

**Fig. 14** Example of logically linked review changes, therefore, labeled with the same ID. These changes gave origin to three modifications. The real-world code review in which this change took place is available online.

```

190 >     }
191 >
192 >     @Override
193 >     protected void doClose() {
194 >         configChangeListenerHandle.remove();
195 >         super.doClose();
196 >     }
197 >
198 >     private void loadUserConfig() throws IOException {
199 >         try {
185 >     }
186 >
187 >     private void loadUserConfig() throws IOException {
188 >         try {

```

**Fig. 15** Example of a *modification* containing only an old code chunk. The real-world code review in which this change took place is available online.

```

411 > #Test(expected = IllegalArgumentException.class)
412 > public void smallWhenSourceOnlyAndWildcard() {
413 >     assertNotNull(new RefSpec("refs/heads/*"));
414 > }
415 > #Test(expected = IllegalArgumentException.class)
416 > public void smallWhenDestInlinedNewWildcard() {
417 >     assertNotNull(new RefSpec("refs/heads/*:refs/heads/*"));
418 > }
419 > }
420 >
421 > #Test(expected = IllegalArgumentException.class)
422 > public void invalidWhenSourceOnlyAndWildcard() {
423 >     assertNotNull(new RefSpec("refs/heads/*"));
424 > }

```

**Fig. 16** Example of a *modification* containing only a new code chunk. The real-world code review in which this change took place is available online.

**Instance Unit.** When labeling the review changes, we assigned the same ID to multiple changes when they were logically linked together: *e.g.*, multiple changes involving the renaming of the same variable (as shown in Figure 14). However, the granularity of our approach is the individual code change, which we call *modification* (an example of *modification* is shown in part ① in Figure 14). More precisely, a modification is composed of a pair of code chunks (*i.e.*, a group of continuous modified code lines): an old code chunk reporting the code removed in a change, and a new one representing the added code. However, a modification might not include (1) an old code chunk in case the code was only removed (Figure 15) or (2) a new code chunk if the code was added without removing anything (Figure 16).

We mined GERRIT to extract the related code chunks for each of the review changes in the labeled dataset. However, the GERRIT API does not allow to extract information on the link between the old and new code chunks. Furthermore, although GERRIT UI displays related old and new code chunks together, this link is made based on the changes line number and not their

**Table 1** Modification groups in our dataset.

Group	Number of changes	Percentage of changes
Documentation	1276	48.33%
Visual representation	402	15.23%
Structure	780	29.55%
Functional	182	6.89%

content. Old and new code chunks impacting the same lines of code are reliably linked together. However, this is not the case when the old and new code chunk impact different code lines: *e.g.*, when a function declaration is moved in a class. For these reasons, we need to link related old and new code chunks by ourselves. Keeping the old and the new code chunks separated, the designed approach computes the Levenshtein distance [97] among each old chunk and all the new ones. This process leads to the construction of a weighted bipartite graph, where each code chunk is a node and the computed distance represents the weight of the link. Then, it selects the pair that has the lowest distance, links them into a modification, and removes them from the graph. Finally, our linking approach proceeds iteratively among the remaining nodes in the graph. If multiple links have the same weight, it relies on the assumption that related code chunks are likely to have similar positions in the file. We tested our linking approach against all modifications contained in our dataset, after removing changes containing only *import* statements. The proposed approach reached a precision of 86.07% and an accuracy of 89%. Furthermore, we removed modifications containing only *import* statements as they might introduce potential bias in the classification approach. Chunks containing only *import* statements are logically linked to the chunks where the imported entities are used, sharing a common classification (category and type). Once grouped into modifications, these import chunks might have similar characteristics but different types, potentially reducing the classifier performance. Overall, this leads to 2,641 modifications. Each modification has two corresponding labels reporting its *category* (evolvability or functional) and its *group* (documentation, visual representation, structure or functional), respectively.

From Table 1, we see that functional changes represent less than the 7% of all modifications considered, while ‘documentation’ changes constitute 48.33% of the whole dataset, ‘visual representation’ the 15.23%, and ‘structure’ the remaining 29.55%. Such a distribution of changes confirms the findings previously reported in literature [8, 17, 58].

**Labeling validation.** To evaluate the reliability of the manually assigned labels, a second author performed an independent labeling of a statistically significant subset of the labeled dataset (306 changes, leading to a confidence level of 95% and a margin of error of 5%). Comparing the labels given by the two authors, 90% of the categories labels matched perfectly as well as 75% of the type labels. In the other cases, the two inspectors opened a discussion to reach a consensus and the dataset was adapted accordingly. To measure

**Table 2** Code Metrics employed in the evaluated approach at **code chunk** level.

Metric	
Description	Rationale
LOC	
Number of lines of code contained in the code chunk	The considered four groups of changes involve chunks of code of different sizes. For instance, visual representation changes often influence few lines of code, while a structure change might require to move an entire function (leading to a change with a high LOC). We further analyze this aspects by looking at LOC Comments, LOC Exec, and LOC Blank separately.
LOCComments	
Number of code comments lines contained in the code chunk.	Documentation review changes often involve comments [58]. Therefore, changes with high LOCComments are likely to belong to the documentation group.
LOCExec	
Number of lines of executable code in the code chunk.	Structure changes often require to modify large portions of code ( <i>e.g.</i> , removing unused functions), while changes in the other groups generally impact only few lines of code.
LOCBlank	
Number of blank lines in the code chunk.	Visual representation changes improve the layout of the code without affecting its functionality. For instance, a change containing only blank lines is likely to belong to the visual representation group.
First character	
First character of the code chunk.	Specific initial characters in Java might indicate the nature of the change: <i>e.g.</i> , the character “@” is likely to introduce a Java doc comment. Similarly, a “\” character might indicate that the change is a documentation change.
End character	
Last character of the code chunk.	Specific ending characters in Java indicate the nature of the change: <i>e.g.</i> , the character “\” is likely to conclude a comment. If a comment is added at the end of a code instruction, this line might not be reflected in the LOCComments. We introduced this metric as a way to deal with such particular scenarios.
Cyclomatic complexity	
McCabe Cyclomatic complexity of the code chunk.	Documentation and visual representation changes possess low cyclomatic complexity, while structural and functional changes often involve the addition/removal of, for instance, functions. This leads to a higher cyclomatic complexity.

the inter rater agreement between the two authors we computed Krippendorff’s alpha coefficient [51] achieving a measure of 0.447 for the categories labels and 0.673 for the type labels. Although the alpha value for categories labels reports only a moderate agreement, we argue that this reflects the intrinsic unbalance in our dataset: Evolvability changes happen significantly more often during code review compared to functional changes [17, 58].

**Machine-learning Features.** As features of the machine learning algorithm, we selected metrics with three different scopes: the code in the old chunk, the code in the new chunk, and the difference between them. Table 2, Table 3, and Table 4 show a summary of the metrics, grouped by their scope (the single code chunk or the whole modification) and the rationale behind them.

**Table 3** Code Metrics employed in the evaluated approach at **modification** level.

Metric	
Description	Rationale
LOC diff	
Difference between the LOC of the old and new code chunk.	A structure modification often involve the addition or removal of a high number of lines of code between the old and new code chunk. On the contrary, a documentation modification might involve a simple variable renaming, therefore having a low LOC diff.
LOCExec diff	
Difference between the LOCExec of the old and new code chunk.	Similarly to what stated for <i>LOCExec</i> , a high difference in LOCExec between the old and new chunk in a modification indicates that a vast portion of code was added or removed, pointing towards structure or, rarely, functional modifications.
LOCComments diff	
Difference between the LOCComments of the old and new code chunk.	Similarly to what stated for <i>LOCComments</i> A high LOCComments diff between the old and new code chunk might a strong indicator of a documentation change.
Cyclomatic diff	
Difference between the Cyclomatic complexity of the old and new code chunk.	The difference in cyclomatic complexity between the old and new code chunk in a modification reflect the addition/deletion of, for instance, code cycles. The inclusion of this feature might support the classifier in identifying functional and structure changes. Documentation and visual representation changes generally do not impact the cyclomatic complexity of a code chunk.
#Added words	
Number of words added in the new code chunk.	Some modifications involve the <i>addition</i> of only few instructions in the same line of code. For instance, <i>Interface</i> functional changes often only involve the addition of a missing parameter, leaving the rest of the code line unmodified. A similar situation can occur with <i>Logic</i> changes, where a change often impact a single element of a comparison statement.
#Deleted words	
Number of words removed from the old code chunk.	Some modifications involve the <i>removal</i> of only few instructions in the same line of code. Similarly to <i>#Added words</i> , this metric allows to capture functional changes that modify only few words in a line of code.
#Added characters	
Number of characters added in the new code chunk.	Some changes impact only few characters in a change. For instance, check or logic functional changes might involve the modification of few characters (e.g., a numerical variable) in a conditional statement.
#Deleted characters	
Number of characters deleted in the old code chunk.	Similarly to <i>#Added characters</i> , functional changes (e.g., check or logical) might require the removal of only few characters in a line of code.
#Methods diff	
Difference between the number of methods in the old and new code chunk.	Structure and functional modifications might remove unused or wrong function calls and/or add new function calls. Therefore, we included this feature in our model.
#Methods changed	
Number of methods changed in the modification.	Similarly to <i>#Methods diff</i> , based on our observations while constructing the dataset, a high number of methods changed might indicate a structure of functional modification.
#Methods added	
Number of methods added in the new code chunk.	Similarly to <i>#Methods diff</i> , based on our observations while constructing the dataset, a high number of methods changed might indicate a structure of functional modification.
#if diff	
Difference in the number of <i>if statements</i> between the old and new code chunk.	The two most prominent types of functional changes, <i>check</i> and <i>logic</i> [17], involve modification or addition of conditional statements. Our hypothesis is that the difference in the number of <i>if statements</i> in a modification might help in identifying functional changes.
#Added if	
Number of added <i>if statements</i> in the new code chunk.	Similarly to <i>#if diff</i> , we argue that the difference in the number of <i>if statements</i> added in the new code chunk of a modification might help in identifying functional changes.

**Table 4** Code Metrics employed in the evaluated approach at **modification** level.

Metric	
Description	Rationale
#Cycles diff	
Difference in the number of cycles ( <i>for</i> and <i>while</i> ) between the old and new code chunk.	Similarly to <i>#if diff</i> , the addition of a high number of cycles might indicate the presence of logic functional change.
Levenshtein Distance	
Levenshtein distance between the old and new code chunk.	The Levenshtein distance might quantify the impact of a modification. Structure modifications might have a high distance between the old and new code chunk since they involve the addition/removal of vast portions of code. On the other hand, modifications such as variable renaming, often impact only few characters in the code chunk, therefore leading to a small distance between the old and new chunk.
#Keywords	
Number of Java keywords removed or added in the modification.	We looked for specific Java keywords to understand if a modification impacted variables or method definitions. For instance, this might show the presence of <i>supported by language</i> changes. We consider the following keywords: <i>private</i> , <i>public</i> , <i>protected</i> , <i>static</i> , <i>final</i> , <i>volatile</i> , <i>this</i> , and <i>void</i> .
Keyword	
If <i>#Keywords</i> equals 1, it contains the specific keyword modified.	The presence of a specific Java keyword can indicate if the change impacted a variable or method declaration.
#New diff	
Difference in the number of new objects created between the old and new code chunk.	Organizational modification might remove unused objects. On the contrary, the presence of new objects declarations in the new chunk might point to a functional or solution approach modification.
#New added	
Number of new objects added in the new code chunk.	Similarly to <i>#New diff</i> , the presence of new objects might point to a functional or solution approach modification.
#Assignment diff	
Difference in the number of assignments between the old and new code chunk.	Functional or solution approach modification might introduce new variable assignments in the new code chunk. On the contrary, documentation or visual representation modification keep the number of variable assignments unaltered between the old and new code chunk.
#Sum/difference diff	
Difference in the number of sums/differences operations in the old and new code chunk.	Logic changes, (in particular their sub-type <i>compute</i> changes [58]) can fix computational mistakes in the code. Considering the removal or addition of sum/difference operations in the code might help identify these kind of changes. We did not consider multiplication or division operations as the symbols used to denote them in Java is also used for other purposes: <i>e.g.</i> , in combination with “\” to introduce multi-line comments.
#Comma diff	
Difference in the number of commas between the old and new code chunk.	Commas in Java are often used in assignment statements. We introduced this metric to work together with <i>#Assignment diff</i> to cover cases left unaddressed by it. Differences in the number of variable assignments might indicate <i>resource</i> changes (a type of functional changes).
#Round brackets diff	
Difference in the number of round brackets between the old and new code chunk.	We use this metric as a proxy for the number of loops and method calls contained in the old and new code chunk. In particular, the introduction of a method call in the new code chunk might reveal the presence of a functional change.
Brackets	
The code changed in the modification is included in round brackets.	Functional changes (for instance, <i>check</i> and <i>interface</i> changes) modify a comparison, a check statement or a method call. In Java, these operations are included between round brackets.

**Table 5** Metrics derived from the analysis of review change comments.

Metric	Words
comment word	comment; style; messag; string; log; error; read
method word	method; scope; enum; tag; call
return word	return
change word	chang; remov; miss; order; delet; sort
final word	final
test word	test; bug
add word	add; implement; ad

The rationale behind each of the metrics selected for our investigation based on the literature and our observations during the creation of the dataset. We combined common code analysis metrics (*e.g.*, LOC, LOCExec, or Cyclomatic Complexity) with a selection of code readability metrics: number of commas or number of cycles [23]. The selection of these metrics is based on an analysis of the literature on what can characterize the type of a modification performed by developers [34] and what can capture the nature of source code under different perspectives. The metrics that work at code chunk level are computed for both the old and the new code chunk in a modification. To compute the selected metrics, we extracted from GERRIT the code snippet of each review change contained in our dataset using the java implementation of the Gerrit REST API.<sup>1</sup> Based on the retrieved code snippets, we then computed the selected code metrics. The code developed to extract and compute the metrics is available in our replication package [35] as part of the devised machine-learning approach.

We also consider the comments associated with a modification (if any). The first of these metrics is *words in comment*, a metric that counts the number of words included in a comment. In computing this metric, we considered the main comment and all its replies as a unique entity. Our hypothesis is that modifications belonging to different categories or types involve a different amount of discussion (*e.g.*, the request to change an algorithm might require more explanations than adding a missing comment). We collected the comments related to each code change using the Java implementation of the Gerrit REST API, in a similar fashion to what done for the code of each change. Subsequently, we analyzed the content of comments. To this aim, we first follow a typical Information Retrieval normalization process [9] involving tokenization, lower-casing, stop-word removal, and stemming [72]. After these pre-processing steps, we computed the 50 most frequent words taking into account the whole corpus of comments of our dataset. Based on this analysis, we cluster them into seven groups grouping together words indicating a similar operation on the code (*e.g.*, *remove* and *delete*): We define groups of words that could help the classifier to distinguish between different change types. The defined group of words are reported in Table 5

<sup>1</sup> Java Gerrit REST API: <https://github.com/uwolfer/gerrit-rest-java-client>

**Table 6** Gain Ratio of the ten most relevant features to predict change categories or groups, respectively. When a metric is computed at code chunk-level, we specify between brackets if it is related to the new or the old code chunk in a modification.

Category		Group	
Feature	Gain Ratio	Feature	Gain Ratio
#Added if	0.0938	LOCExec (new chunk)	0.1523
#if diff	0.0782	#if diff	0.1299
return word	0.0564	#Added if	0.129
#Assignment diff	0.0483	#Cycles diff	0.122
Cyclomatic diff	0.0396	LOCBlank (old chunk)	0.1192
LOCComments (new chunk)	0.0320	LOCComments (new chunk)	0.118
LOCComments diff	0.0267	LOCExec diff	0.1118
#New diff	0.0232	#Sum/difference diff	0.1089
LOCExec (new chunk)	0.0218	#Assignment diff	0.1086
#Comma diff	0.0199	LOC Blank (new chunk)	0.1074

**Table 7** Pearson’s correlation value of the ten most relevant features to predict change categories or groups, respectively. When a metric is computed at code chunk-level, we specify between brackets if it is related to the new or the old code chunk in a modification.

Category		Group	
Feature	Corr. value	Feature	Corr. value
#Added if	0.2776	#Methods added	0.2128
#Methods added	0.0636	Levenshtein distance	0.1349
Levenshtein distance	0.0616	#New added	0.1293
Comments words	0.0406	#Added if	0.1115
LOCComments diff	0.0388	End character (new chunk)	0.1078
First character (old chunk)	0.0375	Cycl. complexity (new chunk)	0.1048
#Sum/difference diff	0.0362	First character (new chunk)	0.1002
#Added words	0.0307	End character (old chunk)	0.0865
#Deleted words	0.0282	First character (old chunk)	0.0863
#Comma diff	0.0280	#Methods diff	0.0855

**Machine-learning Algorithm.** We experiment with three different classifiers: RANDOM FOREST, J48, and NAIVE BAYES. We selected these classifiers as they have been successfully applied to solve similar problems in the software engineering domain: *e.g.*, defect prediction [37, 53, 85] or refactoring recommendation [52, 67]. These classifiers make different assumptions on the underlying data, as well as having different advantages and drawbacks in terms of execution speed and overfitting [20]. In building our approach, we deal with two different classification scenarios: (1) a binary classification to classify a modification in one of the two categories (evolvability or functional); (2) a multi-class classification to assign each modification to one of the four groups (documentation, visual representation, structure, and functional). In the multi-class classification scenario, we estimate multi-class probabilities directly. We do not use techniques such as *one-vs-one* or *one-vs-rest*.

**Feature selection:** To identify relevant features for our classification problem, we apply two different techniques: (1) Gain Ratio attribute selection [45] and (2) Correlation-based feature selection [39]. Gain ratio attributes selection



evaluates the importance of an attribute computing the gain ratio with respect to the class. Table 6 reports the ten most relevant features in terms of Gain Ratio to predict *categories* or *groups*, respectively. The complete list of metrics is available in our replication package [35]. Correlation-based feature selection measures instead the value of Pearson’s correlation between it and the class to evaluate the importance of a feature. Table 7 shows the ten most relevant features based on their correlation value (the complete list of values is available in our replication package [35]).

Our analysis revealed how the number of *if statements* (*#Added if* and *#if diff*) is one of the most prominent variable in our model (at both category and group-level). The vast majority of *check* and *logic* changes among the functional changes in our dataset (respectively, 34.95% and 24.73% of the functional changes) might explain the high importance of these variables in our model. Both *check* and *logic* changes involve fixing issues with variable checks and comparisons: *e.g.*, adding a missing check on the variable returned by a method.)

Variables related to the number of lines of comments are also evaluated as particularly important (especially the number of the LOC in the new code chunk of a modification). We argue that high number of comments LOC might be a strong indication of a documentation change. Overall, documentation changes constitute 48.33% of all the modifications in our dataset.

We test our approach using either gain ratio or correlation-based feature selection. We notice that using correlation-based feature selection leads to better performance. For this reason, we select this technique and we remove the reported non-relevant features. We test different correlation thresholds to remove non-relevant features to predict modification *categories* and *groups*. When evaluating our model at category-level, we notice that its performance slowly increase until a threshold of 0.005 of correlation is selected to exclude non-relevant features. Afterwards, the performance of the model remains stable until decreasing again for even smaller threshold. Based on these observations, we select 0.005 as our feature selection threshold at category level. This leads us to exclude the following features from our model: *method word* (corr. value 0.0044); *#Round brackets diff* (corr. value 0.0031); *change word* (corr. value 0.0030); *#Assignment diff* (corr. value 0.0009); *#Methods changed* (corr. value 0.0001); *Brackets* (corr. value 0). Concerning groups, we apply a correlation threshold of 0.01. This leads to include almost all the computed features in our model, with the exception of *Brackets* since our analysis shows that is uncorrelated to the class variable (corr. value 0).

**Data Preprocessing and Training Strategy.** We consider three typical aspects for the use of machine learning models: (1) data normalization [48], (*i.e.*, the reduction of the feature space to the same interval) (2) removal of non-relevant features [24], and (3) balancing of minority classes [50]. We use the `normalize` function available in the WEKA toolkit to scale data [38], while we employed and Synthetic Minority Oversampling (SMOTE) [25] for bal-

ancing the data, respectively. To implement SMOTE in our machine-learning approach, we relied on the *SMOTE* class offered by the WEKA Java API.<sup>2</sup> Afterward, we run the selected machine learning classifiers using all combinations of settings (*e.g.*, without data normalization but including feature selection), so that we can identify the most performing solution. Moreover, we test the performance of the models considering both within- and cross-project strategies. In the first case, we train models only using the data of single projects and validate their performance using the 10-fold cross validation method [7]; to mitigate the possible negative effect of random fold splittings [87] and have a more reliable interpretation of the results, we repeat the validation ten times. In the second case, the data of two projects are used to train the models, while the remaining project is used as test set; we run the models multiple times to allow each project to be the test set once.

**Evaluation Metrics.** We assess the goodness of the experimented models by computing Precision, Recall, F-Measure, AUC-ROC, and Matthew’s Correlation Coefficient (MCC) [75]. These metrics provide different perspectives on the performance of the investigated approach.

#### 4.2 Analysis of the results

The final manually labeled dataset contains 1,504 changes with an assigned label. We classified 227 additional review changes, but these are borderline cases that could not be properly assigned to a label with the available data, thus we excluded them from the dataset to not reduce potential errors.

Table 8 shows the results achieved by the best of the experimented models in terms of F-Measure. A report on the performance of the other experimented models and configurations is available in our replication package [35]. This configuration uses RANDOM FOREST to predict both the category and group labels, normalizing the data and oversampling the minority class using SMOTE. We kept the standard parameter settings offered by the WEKA implementation of SMOTE. In particular, the default percentage parameter of SMOTE was left to 100: This parameter specifies the percentage of SMOTE instances the algorithm creates. To avoid introducing bias in the classifier performance, we apply class-rebalancing with SMOTE only to the training data and never to the test data, following the guidelines of Santos et al. [80]. The results have been computed combining the three projects in our dataset and applying 10-fold cross validation ten times. At category level, our model reached promising results, showing an AUC-ROC of 0.91 and MCC of 0.61. Furthermore, the model could classify *evolvability* changes with a precision and recall of 0.97 and 0.98, respectively, resulting in an F-Measure of 0.97. As for the *functional* class, we achieved an F-measure of 0.63, with precision and

<sup>2</sup> SMOTE class in the WEKA JAVA API: <https://weka.sourceforge.io/doc/packages/SMOTE/weka/filters/supervised/instance/SMOTE.html>

**Table 8** Performance achieved by the three considered Machine-Learning algorithms (Naive Bayes, J48, and Random Forest) for categories and groups using SMOTE.

Naive Bayes					
Class	Precision	Recall	F-Measure	AUC	MCC
Evolvability	0.99	0.12	0.21	0.66	0.09
Functional	0.08	0.99	0.14	0.66	0.09
Documentation	0.68	0.36	0.47	0.69	0.23
Visual representation	0.23	0.85	0.36	0.72	0.25
Structure	0.71	0.12	0.20	0.63	0.21
Functional	0.18	0.35	0.24	0.65	0.17
J48					
Class	Precision	Recall	F-Measure	AUC	MCC
Evolvability	0.96	0.97	0.97	0.83	0.54
Functional	0.60	0.53	0.57	0.83	0.54
Documentation	0.85	0.86	0.85	0.90	0.71
Visual representation	0.77	0.71	0.74	0.89	0.70
Structure	0.74	0.74	0.74	0.85	0.63
Functional	0.54	0.56	0.55	0.80	0.52
Random Forest					
Class	Precision	Recall	F-Measure	AUC	MCC
Evolvability	0.97	0.98	0.97	0.91	0.61
Functional	0.70	0.57	0.63	0.91	0.61
Documentation	0.88	0.89	0.88	0.95	0.77
Visual representation	0.84	0.77	0.80	0.95	0.77
Structure	0.77	0.81	0.79	0.94	0.70
Functional	0.68	0.62	0.65	0.91	0.63

recall of 0.70 and 0.57, respectively. At group level, the AUC-ROC computed for each of the four classes was above 0.91, giving a first indication of the feasibility of the task using the proposed model.

Considering these results, the selected features seem to be good predictors for the types of review changes, thus suggesting that it is possible to overcome the scalability issues of previous approaches to classify review changes.

***Finding 1.*** *The investigated approach reaches an AUC-ROC beyond 0.91, suggesting that it is possible to overcome the scalability limitations of previous work by using supervised machine learning.*

At category level, we selected a feature selection threshold of 0.005 (as reported in Section 4.1). This value was identified as a key point in the classifier performance: The classifier performance slowly increase until reaching this threshold, after which they begin to decrease. Despite this, the selected value might be considered too small and, thus, lead to overfitting. To mitigate this risk, we explored how the performance of our classifier changed for higher thresholds, 0.005 and 0.01, respectively. Table 9 reports the results achieved by our classifier when these feature selection threshold are selected. We used

**Table 9** Performance achieved by our approach (using Random Forest and SMOTE) with different correlation thresholds.

Class	Precision	Recall	F-Measure	AUC	MCC
<b>Threshold: 0.005</b>					
Evolvability	0.97	0.98	0.97	0.90	0.60
Functional	0.69	0.56	0.62	0.90	0.60
<b>Threshold: 0.01</b>					
Evolvability	0.97	0.98	0.97	0.89	0.57
Functional	0.64	0.56	0.60	0.89	0.57

the best performing classifier (Random Forest with SMOTE) based on the results reported in Table 8. Even for higher thresholds, the performance of our approach remain stable and in-line with the previously reported results. This, combined with the other measures taken to reduce possible overfitting, contributes to strengthen our confidence in the reported findings. Nonetheless, further work might be conducted to further reduce this possible bias: *e.g.*, by using Recursive Feature Elimination with Cross Validation (RFECV).

Despite the good overall results, we make some further observations on the performance of the model when it classifies the *functional* class. Table 8 shows that this category is the most problematic. To better understand the reasons behind this result, we conducted a further qualitative analysis. First, the limited amount of functional changes naturally limits the capabilities of the machine learner, as it lacks enough examples to fully learn how to classify them [69]. Our dataset contains 6.89% of functional review changes. This is in line with previous findings that reported how functional changes constitute only a very low percentage of all the code changes that happen during code review [17, 58].

Furthermore, the ‘functional’ class includes changes with very different characteristics: indeed, a functional change might range from a missing `if` statement to the modification of a whole algorithm. This large variety of changes for one category is a further challenge for machine learning solutions. Nevertheless, the performance achieved indicates that the features selected can discriminate most of the functional changes in our dataset, thus representing a promising baseline that other researchers can improve through further investigations. Finally, we performed an additional analysis aimed at improving the classification of functional changes. In particular, we assessed the suitability of cost sensitive learning [31]: we assigned different cost weights to false positive and negatives of functional instances, thus testing the model with different cost matrix configurations. However, we did not achieve any significant improvement, confirming that further research is required to improve the classification of functional changes.

**Table 10** Pearson’s correlation value of the ten most relevant features to predict if a modification belongs to the functional or structure. When a metric is computed at code chunk-level, we specify between brackets if it is related to the new or the old code chunk in a modification.

Feature	Correlation value
#Added if	0.2833
#New added	0.0748
End character (new chunk)	0.0686
#Methods added	0.0669
#Comma diff	0.0658
#New diff	0.0657
#Methods diff	0.0653
Levenshtein distance	0.0616
# Sum/difference diff	0.0606
LOCComments diff	0.0597

**Finding 2.** *The distribution of changes in our dataset confirms findings of existing literature [17, 58]. Furthermore, classifying functional changes is challenging because of (1) their limited presence in the dataset and (2) their large variability in their content.*

Our results showed that Random Forest achieved the best performance among the considered models. This is in line with findings of previous studies that reported how Random Forest outperforms Naive Bayes and other decision tree-based machine learning algorithms [36, 57]: For instance, in the field of software engineering, Random Forest was shown to perform better than other algorithms when used to build bug-prediction approaches [43]. Random forest is a machine learning algorithm that builds multiple decision trees and trains them with the “bagging” method. Compared to a decision tree, random forest randomly selects observations and features to create multiple trees. Then, the final performance is computed by averaging the results of each tree. We argue these characteristics of random forests might be what allowed it to outperform the other models we considered in our investigation: J-48 (an implementation of C4.5) and Naive Bayes.

To gain further insight into the differences between different *groups* of review modifications, we investigated which features are the most relevant to distinguish between a *functional* or a *structure* modification. To this aim, we built a binary classifier with these two groups (functional and structure) of modifications and excluding documentation or visual representation modifications. Then, we analyzed the importance of each feature in the model using correlation-based feature analysis. We report the ten most relevant features in Table 10.

The most important feature is the number of if added in a modification: Most of the functional defects involve changes in the logic of the algorithm or add checks on the variables. On the contrary, modifications that instantiate new objects or remove/replace method calls are likely to belong to the structure group: e.g., modifications belonging to the “solution approach” type (a

**Table 11** Mean and standard deviation of *#if added*, *#new added*, and *#methods added* for each of the four *groups* of review modifications.

Group	#Added if		#New added		#Methods added	
	Mean	St. dev.	Mean	St. dev.	Mean	St. dev.
Documentation	0.0196	0.1386	0.0321	0.1850	0.1747	0.3901
Visual Rep.	0.0796	0.2710	0.0696	0.2548	0.2512	0.4342
Structure	0.0729	0.2615	0.1498	0.3677	0.4942	0.5252
Functional	0.3296	0.5371	0.0824	0.2757	0.4065	0.4925

subgroup of structure modifications) often involve removing unused methods to increase the maintainability of the code.

Finally, informed by the results of our feature importance analysis (reported in Table 6 and Table 7) and by the comparison between functional and structure modifications (whose results are illustrated in Table 10), we select a subset of features that might be particularly relevant to highlight the difference among the four different *groups* of modifications. At the same time, we aim to verify previous hypotheses formulated based on the results of the feature importance analysis. To this aim, we selected the following three features: *#if added*, *#new added*, and *#methods added*. We report in Table 11 the mean and standard deviation of these features for each group of review modifications.

The mean of the number of added *if* statements (*#Added if*) is significantly higher for the functional modifications compared to the other three groups (documentation, visual representation, and structure). This confirms our previous hypothesis: A functional change often requires the addition of a check on a variable and, therefore, the number of added *if* statements can significantly help to distinguish between functional and evolvability changes. Considering the amount of added new objects declarations and method calls added, we noticed that the mean values of both these features are significantly different when comparing documentation and visual representation changes to structure and functional changes. Looking at only structure and functional changes, our results still confirm our hypothesis: Structure modifications have a higher number of added objects and methods, but the difference with functional modifications is not so large as originally expected.

## 5 RQ<sub>2,1</sub>: Evaluation of Developers' Perceptions

To carry out an evaluation on using information on the types of review changes as a mean to inform practitioners about their code review process and evaluate it, we conduct semi-structured interviews involving twelve developers with code review experience (whose characteristics are summarized in Table 12).

After verifying that is possible to automatically classifying review changes (therefore overcoming the scalability issue of previous works [8, 17, 58]), our aim is to understand how this information is perceived by practitioners. For this reason, the goals of this investigation are the following: (1) collecting

developers' feedback on review changes classification and the potential use of this information to support and improve code review practices, (2) conducting a qualitative evaluation on the used taxonomy, and (3) gathering feedback on how to display the generated review changes data. In this investigation, we did not validate our machine learning approach. We focused instead on how information on review changes can be effectively used and shown to reviewers.

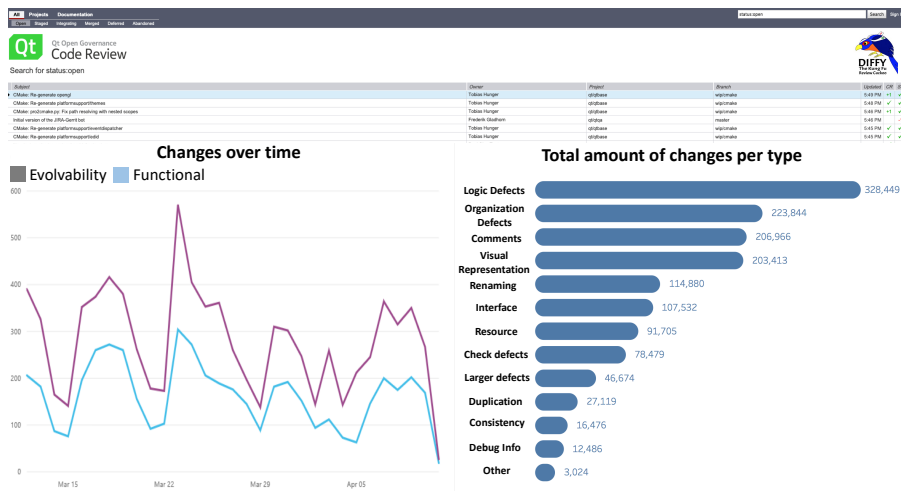
## 5.1 Design and Methodology

Each interview has three parts: (1) A general discussion on code review, focusing on the interviewee's experience; (2) a discussion on the concept of automatically classify review changes as well as the information it offers; (3) the evaluation of the taxonomy [17]. All interviews have been conducted as semi-structured interviews [55, 63]. Starting from the three above-mentioned general topics, the use of semi-structured interviews allowed us to dynamically adjust the structure of the interview to ask follow up questions when needed or address unforeseen points of discussion raised by the participants. We employed a set of slides to guide participants through the structure of the interview as well as to illustrate key concepts: *e.g.*, the review changes taxonomy. The complete set of slides is available in our replication package [35].

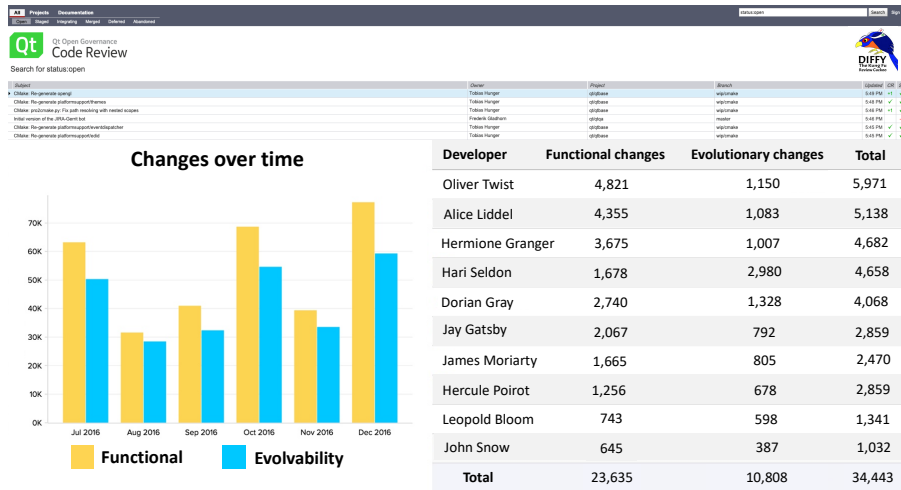
In the first part, we open a discussion about interviewees overall **experience with code reviews**. This discussion allows us to gather participants background experience on code reviews and set the context for the concept testing. Afterwards, we present to participants **the concept of classifying review changes**. First, we explain orally the general idea of classifying review changes, then we show this information in four POWERPOINT slides. Using code review data from the QT project [4], the slides show two possible UIs as GERRIT extensions: (1) a general overview of the changes of the entire project, and (2) a detail view on a specific commit.

For the general overview, two alternative representations are shown in two different slides (Figure 17 and Figure 18). The UIs differ only from the charts used to represent the types of changes (pie-chart, histograms, *etc.*). We show more than one option to display alternative representations of the same data. Both slides show a list of the latest changes, and three charts representing the type of the changes over time totally, and for each developer. One additional slide of the general overview shows the evolution of review changes by developer, accessible by clicking on the developers' name in the list of changes. Finally, one slide shows the categorization of changes for a specific commit as a pie-chart.

After presenting the concept of classifying review changes, we ask interviewees to express their feedback on this information and how it is displayed; they are also allowed to provide additional ideas and suggestions on what data/information might be useful to developers (*e.g.*, to build a tool based on review changes data).



**Fig. 17** First view of a UI that leverages information on review changes. For readability reasons, this view contains only two charts (as opposed to the three in the original slides) and has a different layout compared to the one shown to the interview participants. The original slides are available in our replication package [35].



**Fig. 18** Second view of a UI that leverages information on review changes, as shown to the participants during the interviews. For readability reasons, this view contains only two charts (as opposed to the three in the original slides) and has a different layout compared to the one shown to the interview participants. The original slides are available in our replication package [35].



In the third step, we discuss with the participants the review changes **taxonomy** [17]. To mitigate possible biases, we introduce the details of the taxonomy only in this last phase. We aim to validate if this taxonomy matches developers' expectations. In particular, we focus on assessing if all the types in the taxonomy are understandable and match the interviewee's experience. Before starting this discussion, we show the taxonomy to the participants and provide them with a brief explanation of each category and type. Because of the complexity of the taxonomy, we do not introduce the changes subtypes. To assess the taxonomy, we ask participants to place in the taxonomy each of the changes mentioned in the first interview phase. We finally discuss how difficult they perceive this task and whether they face any issues.

Interviews are conducted through video conference or in person and lasted approximately from 45 minutes to one hour. However, we do not define a strict maximum duration limit: We consider an interview concluded once all pre-determined topics are covered and the discussion naturally comes to an end. The study is held by one or two researchers and all interviews are recorded and transcribed. The second researchers participated in the first interview to ensure the goodness of the procedure. Before conducting the interviews, the authors obtained the approval of the ethics committee of their home institution. Moreover, interviewees were asked to sign a consent form to consent to take part to the interview. The form described the scope of the study.

To analyze the results of the interviews, we employ *thematic analysis* [56]. As a first step, we create *codes*: Brief descriptions of the content of the interview to summarize an idea presented by the interviewee. Subsequently, we identify common themes across different codes. To generate *themes*, we group together codes referring to the same topic. Finally, we review the identified themes to ensure their goodness: e.g., to avoid overlaps between different themes. The results are first analyzed by one of the authors, then checked by a second one.

**Participants.** Twelve people participated in our semi-structured interviews on review changes. They are selected through snowballing of the professional network of the authors. Table 12 summarizes the background of the participants involved. Their age range from 18 to 44 years, with the majority (9) ranging from 25 to 35. The participants come from eight different countries with three from the US, and the remaining from the Czech Republic, France, Germany, Italy, and other European countries. Eleven participants identified themselves as males, while one identified herself as female. Currently, nine participants work in companies, among whom one also works as a Ph.D. student, two also contribute to open-source projects and one studies at the university. Two users work exclusively for open source projects, and one is a student. Overall, our participants have an average of 11 years of coding experience and 5.3 years in code reviews (std. 4.7 and std. 2.5, respectively). On average, 7.7 years of coding experience were acquired during academic studies (std. 3) and 9 in industry (std. 8.6). Participants reported to spend, on average, 17.2 hours

**Table 12** Interviewees' background (N = 12).

Part.	Gender	Current position	Coding experience (years)		Review Experience
			Academic	Industry	
P1	Male	Industrial Dev. & PhD	8	14	9
P2	Male	Industrial & O.S. Dev.	7	7	5
P3	Male	Industrial Dev.	6	18	9
P4	Male	Industrial Dev.	7	8	5
P5	Male	Industrial Dev.	13	7	6
P6	Male	Industrial Dev. & Student	5	3	1.5
P7	Male	Industrial Dev.	10	1	0.5
P8	Male	Industrial & O.S Dev.	5	25	8
P9	Female	Student	5	3	5
P10	Male	O.S. Developer	8	4	4
P11	Male	Researcher	14	13	7
P12	Male	Industrial Dev.	5	6	4

a week doing code reviews (std. 10.2). Nine participants use code reviews tools such as GitHub [3] and Crucible [1].

## 5.2 Analysis of the Results

This section overviews the main findings achieved in **RQ**<sub>2.1</sub>.

**Information on review changes to support review.** All participants found valuable and interesting the concept of using the information obtained by classifying review changes as mean to increase the understanding of code review. For instance, P1 stated that our proposed approach might help software teams to improve their code review processes and understand where and which problems are happening in the project. Similarly, P12 said that with such features developers may be able to avoid the same errors in the future. P3 also suggested that a tool that displays analytics on review changes might allow companies to understand where resources have been spent. Most developers (ten) mentioned managers as the perfect suit for the information generated by our approach; they commented that managers and CTOs could use these data for a better understanding of the project evolution. They reported how the information on the kind of review changes could be useful to project manager or tech-leads to assess the goodness of their review practices. For instance, P3 explained that this information could reveal if the team is paying attention to the right kind of defects. This is in line with the findings of Bird et al. [19]: their code review analytics tool became a valuable instrument for Microsoft developers to monitor themselves and improve their

code review process. To increase the actionability of this information, interview participants suggested to compare the distribution of review changes in their project with benchmarks, created, for instance, by looking at similar projects or the history of the project under analysis. Eight participants acknowledged the potential benefits of this kind of functionality.

Moreover, our participants found that showing the type of changes for each developer may be problematic, for example creating competition [P12]. To avoid this issue, P11 suggested limiting access to this information. Seven participants stated that the most interesting information is the evolution of changes over time. P4 also stated that an additional feature might allow developers to change the level of granularity (year, month, and week). P11 and P7 suggested adding a chart, visible to the single team member, that compares the developer and to an ideal standard reviewer, obtained averaging the type of changes found by all reviewers in the project.

***Finding 3.** Participants found information on review changes valuable, especially for leads and managers.*

**Evaluation of the taxonomy.** After the participants were briefed on Beller *et al.*'s taxonomy [17] and knew the definition of each category, we asked them to place some of the examples of review changes they mentioned during their interview within the taxonomy. All developers were able to fully or partially associate their changes into the various classes. During the task, all users could see a graphical representation of the taxonomy on the screen (Figure 2). Four participants had no issues performing this task, while five had some initial doubts on where to put a change in the taxonomy. However, they were then able to decide before proceeding.

When asked to comment on the taxonomy, five developers stated that it could have some overlapping issues. Three participants suggested renaming some of the types (i.e., 'Interface', 'Larger defects') to lower confusion. Despite these comments, participants were able to associate most changes in the taxonomy. Although participant P9 recognized some overlapping issues, he stated that, overall, everything was fairly clear. Similarly, P7 found that classes may not be mutually exclusive, but stated that the taxonomy had enough descriptive power.

***Finding 4.** Despite reporting some possible overlapping issues, participants acknowledged the descriptive power of the review changes taxonomy.*

Overall, this evaluation phase brought positive feedback to motivate further research on classifying review changes to support practitioners (*e.g.*, as an analytics tool) and improve Beller *et al.*'s taxonomy.

### 5.3 Follow-up interviews

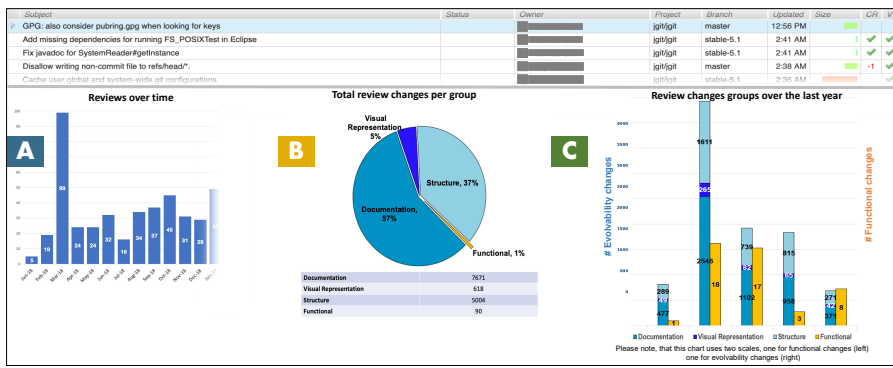
We perform follow-up interviews with 7 developers from the original interviews. Our goal is two-fold: (1) Assessing the concept of *group*, introduced in Section 4.1, as intermediate layer in the review changes taxonomy; (2) Conducting an initial evaluation of the goodness of our approach with developers. Each interview was conducted by the first author in the form of semi-structured interviews and lasted approximately 30 minutes. All interview were conducted remotely. Moreover, to clarify the concepts discussed in the interviews and show examples to the participants, we use a set of slide (an example is available in our replication package [35]). Note that in the following section, we refer to the follow-up interviews participants with the code assigned to them in the original interviews and reported in Table 12.

**Evaluation of review changes groups:** To evaluate the goodness of review changes *groups*, we discuss them with developers. First of all, we introduce again the concept of review changes as well as all the change types reported in Beller et al.’s taxonomy [17] (reported in Figure 2). To support our explanation, we show to the interviewees a slide with the taxonomy.

All of the interviewees positively assessed the reduction of the original taxonomy into the defined four *groups*. P10 reported how the reduction into groups was based on the original taxonomy and did not arbitrarily introduced new categories. Five participants mentioned that the level of granularity of the classification depends on the goal of our approach. Participants agreed that the level of information offered by our approach is enough, for instance, to inform project managers about the kind of changes that happen in their code review process. For instance, P1 reported how *groups* are appropriate to offer an overview of the review process of a project, but a finer level of granularity is required to make this information fully actionable.

**Assessing our approach with developers:** In the second part of the interviews, we conduct an initial evaluation of the performance of our classification approach with developers. To this aim, we show in random order to each participant 20 review changes and ask them to classify each change in one of the four *group* (documentation, visual representation, structure, and functional). To select the changes, we applied our classification approach to randomly selected reviews in the code review history of JGIT. Among the resulting classified changes, we randomly choose five changes per *group*. Each participant is asked to evaluate a different set of review changes: We do not show the same change to multiple participants. Before asking participants to complete this task, we discuss with them the review changes taxonomy and introduce the concept of review change *groups*.

We compare the classification performed by the interview participants with the one of our approach. Overall, developers agreed with the approach classification in 79.41% of the cases. Despite this investigation constitutes only an initial assessment of the performance of our approach, the obtained results are encouraging. We notice that the majority of the disagreements be-



**Fig. 19** General overview shown to participants. The images and charts were cropped for readability purposes. A) Number of reviews over times. B) Total review changes per group. C) Review changes group over time. The original report is available in the replication package [35].

tween developers and our approach was caused by *structure* and *functional* changes. This might be caused by the relatively similar appearance of these two groups of changes: without having an in-depth knowledge of the project, it is not always possible to immediately understand if a change is impacting the functionality of the code (therefore, being a functional change) or not.

## 6 RQ<sub>2.2</sub>: Evaluation with open-source projects

In RQ<sub>2.2</sub>, we verify the meaningfulness of the information on review changes on 20 open source projects. The main goal is to analyze the developers’ perception of such data when related to their own codebase and, therefore, evaluate the impact of automatically classifying review changes for active software projects.

### 6.1 Design and Structure

**Design.** To analyze how information obtained by automatically classifying review changes is perceived by open-source developers, we consider 20 different projects and send the results of the classification to developers in the form of an *online report*, with some attached questions to gather feedback. We devised our reports following a methodology similar to the one used in previous studies [94].

The target population of our investigation is represented by the main developers of each project (*e.g.*, core developers or product owners): to give valuable feedback on the data generated by our approach is indeed necessary to have a high-level view and understanding of the whole code review process of the project. The respondents are contacted by sending the link to the report to the developers’ mailing lists or project forums, as these channels mostly attract the interaction of core contributors [82]. Among the 20 projects (reported in

**Table 13** List of projects for which a report was created (N = 20) grouped by environment.

Environment	Project
Android [6]	Android
Eclipse [30]	JGit Eclipse platform text Eclipse platform ui Egit Equinox Etrice JDT-core Linuxtools M2e Pde Rap Rcptt Scout Sirius Trace-compass Tycho Viatra
Couchbase [26]	Java-client JVM-core

Table 13), three are the initial systems used for the training of our tool (*i.e.*, Android, JGit, and Java-client). The remaining projects are randomly selected among the ones belonging to the same communities as JGit (Eclipse) and Java-client (Couchbase). We focus on these communities because of the high review activity of their developers. Furthermore, all the selected projects respect the following criteria: they are mainly written in Java and use GERRIT as code review tool.

To create the reports, we extract information on the code review history of each project using the Java implementation of the GERRIT REST API<sup>3</sup>. We use these data as input for our review changes classification approach (presented in Section 4). The information returned by the devised classification approach is used to create the graphs shown in the online reports.

**Structure of the reports.** In the online report, after a brief explanation on the meaning of review changes and their selected types (*i.e.*, evolvability, functional, documentation, visual representation, and structure), we show developers an overview of the information collected by the designed approach, together with one real example of each type of change extracted from their project (see Figure 19). Below the latest commits on the GERRIT platform, we show the general overview of the results. The overview shows the results in three forms. One chart (Figure 19, A) represents the total number of reviews that happened over the last year. This data is intended to give an idea of the

<sup>3</sup> Java implementation of GERRIT API: <https://github.com/uwolfer/gerrit-rest-java-client>

amount of merged code reviews in the project. The second chart shows the total types of changes, computed over the last year, in a pie-chart (Figure 19, B). Finally, the histograms (Figure 19, C) represent the types of review changes that happened over the latest twelve months. Given the differences between the number of functional and evolvability changes, the two classes are shown in the same chart but with two different scales. Although this factor should be evident from the data shown on the single histogram of each change, we also inform the developers about the two different scales in the description of the figure and with a label below the chart. Following, we show to the participants both charts in detail (Figure 19 B and C) and ask developers to rate the information gained for the overview and for each detailed chart. More in detail, we ask if the figures are clear and if the information is useful for project managers and for developers. Furthermore, participants can also indicate how useful the entire report is, whether they perceive they learned something new from it and if this information might have a positive influence on their code review process.

At the end, participants can fill out background information to complete the collection of their data. We give them the possibility of contact us in case they were interested in knowing more details about our tool and the results obtained for their project. The answers to all the questions, excluding the ones requiring an extensive answer, are expressed using a Likert scale [54] ranging from 1 (Strongly disagree) to 5 (Strongly agree). All reports are available in our replication package [35].

## 6.2 Analysis of the Results

Out of the 20 reports sent, 18 developers from 10 different projects reached back to us with feedback. 17 of them replied to our questionnaire: we received 15 complete answers and 2 partial ones. Moreover, five developers additionally contacted us directly to give more extensive feedback. All respondents except one are core developers, integrators (*i.e.*, developers having committing/merging rights), or product owners. For this reason, we consider them a valuable source of feedback as they are fully aware of what happens in the code review process. The last participant declared not to be involved in the development of the project, so we excluded their answer from our analysis.

The majority of the respondents acknowledged the novelty of the information provided by the designed approach assigning, on average, a score of 4.2 (std.  $\approx 0.5$ ) to this question on a Likert scale. One developer further commented on the value of this kind of data remarking that: “*this provides a concrete way to measure one of the effects of code review*”. To further confirm this aspect, some developers stated that they learned something new about their projects. For instance, one participant reported that: “*The distribution of changes per group [is] different than expected*”, while another was not aware of the distribution of the changes over time.

Moreover, we confirmed our initial hypothesis about the potential users of review changes data. Our participants agreed that the information provided by the overview (avg. 4.2, std. 0.5), graph B (avg. 3.9, std. 0.4), and graph C (avg. 4.3, std. 0.4) might be particularly valuable for a project manager. One of the developers reported that a closer look to the review process might help management to better understand the development process. However, they also mentioned the possibility of misuse of these data by the management. Such a possibility was also considered by some of the participants in our concept testing.

In contrast, and as found in **RQ<sub>2.1</sub>**, we obtained mild reactions when asking if the report could have been useful for developers (*e.g.*, avg. 3.5, std. 0.62 for the overview). A developer further commented on this matter: “*these kind of statistics are nice for a manager, but no clue how this would influence daily practice of an engineer?*”.

Overall, developers positively assessed the value of the information shown in the reports. This gave an initial indication of the potential usefulness of information on review changes for practitioners. However, further studies need to be conducted to fully evaluate the usefulness of this information: *e.g.*, integrating our classification approach into the code review pipeline of existing projects.

***Finding 5.*** *Participants positively assessed the novelty of the information on review changes. Furthermore, they confirmed our findings from RQ<sub>2.1</sub>: the main target of these data are project managers, while their importance for the developers varies.*

As a further step of the investigation, we evaluated potential actions on the review process that could be triggered by the information of our report. We asked participants to report how the data shown could influence code review policies, use of support tools, and reviewers assignment. However, we could not reach any strong conclusion on this matter since we got neutral responses (*e.g.*, avg. 3.0, std. 1.03 for the tools use). We argue that a potential cause of these results might be the lack of further documentation and interaction with our visualization. For instance, one of our participants stated that they would need more details about each category to fully grasp the potential of the information. Similarly, another developer said that they might want to have more background on the specific changes that were classified in each category. In other words, developers recognized the value of the reported information but, at the same time, require additional understanding and possibly other data to fully benefit from it.

***Finding 6.*** *To determine the potential of review changes data, developers need more information on the metrics and the possibility to interact with the displayed data.*



## 7 Discussion and Implications

Our results highlighted aspects to be further discussed.

**Detecting functional defects in code review.** As noticed in the context of **RQ<sub>1</sub>**, most of the changes applied in code review refer to evolvability modifications rather than functional ones. This finding confirms the results reported by Beller et al. [17] and further supports the research aimed at improving static analysis tools to pinpoint potential issues in source code [28, 41, 92, 93]. Moreover, this result suggests that developers might benefit from the outcome of dynamic analysis methods during code review: for instance, providing developers with the output of test cases may help them in finding defects in source code.

**On the value of classifying review changes.** First and foremost, the results achieved from both **RQ<sub>2.1</sub>** and **RQ<sub>2.2</sub>** suggest that the information coming from the analysis of the review changes is perceived as useful for project managers, and possibly software developers, to better understand the types of modifications done and achieve better insights on the code review process of the project. As such, despite our automatic approach has still some limitations when classifying functional changes, it represents the first step to put this kind of knowledge at the service of developers. On the one hand, our work has implications for practitioners: They can adopt our automated approach to improve their code review process rather than relying on time-consuming manual classification of code changes. On the other hand, our work contributes to the state of the art in code review research and opens new directions for researchers, who are called to further investigate code review practices and how to provide meaningful information to practitioners.

**Integrate automatic review changes classification into code review.** To support the retrospective analysis of the code review process, we envision our approach to be integrated in a tool linked to the Gerrit repository of the project, which analyses it at constant intervals. The tool will display this information to developers and project managers. The results of **RQ<sub>2.1</sub>** and **RQ<sub>2.2</sub>** allowed us to collect some preliminary requirements that such a tool would need to fulfill. In particular, developers reported that (1) the tool should display the evolution of this information over time (i.e., number of changes per type in a specific time-frame), (2) the information displayed needs to be interactive, and (3) the tool needs to provide background information on each type of change.

Our machine-learning approach could also be employed at review-time, possibly integrated as a Gerrit plug-in, to display information on review changes directly in the Gerrit UI. We envision to extract the changes in the code to be reviewed, classify them, and visualize this information as warnings to the user. Each warning will be associated to each modification (or code chunk) to inform the user beforehand about the type of changes they will review. For

both scenarios, it is fundamental that the devised tools are well-integrated into developers' existing work-flow, as reported by previous findings [78, 79].

**Further research on review changes is needed.** One of the key results of our study relates to the use of information on review changes as an instrument to support review practices: as stated by developers involved in **RQ<sub>2.1</sub>** and **RQ<sub>2.2</sub>**, the provided information represents a novelty and gives a new perspective on the internal mechanisms of the code review process. Moreover, some developers commented that these data were interesting and gave them a better idea on the types of changes on which they focus in their reviews. However, given the limitations of a static report, it was harder for developers to evaluate the data shown and transform them into actionable improvements of their code quality. Given the positive results, further work can be devoted to creating a tool that interactively displays review changes data, and deploys it with software projects in a longitudinal study, in which developers have the chance to test and learn how to use such a tool and get insights.

**Investigate the impact of review comments.** Only approximately 33% of the review changes in our dataset were caused by a reviewer's comment. Moreover, the same comment might have triggered multiple changes, but using the Gerrit API we could only link a comment to the change next to which it was placed during the review. This further reduced the number of modifications in our dataset linked to a comment. Overall, this did not allow us to employ more complex techniques to analyse the comments. Nonetheless, code review comments might constitute a promising source of information to improve the performance of our classification approach. For this reason, we believe future work should focus on the analysis of comments characteristics, possibly combining them with source code characteristics using code vectorizer techniques (*e.g.*, code2vec). To apply this technique, a future study should address the current two limitations of our approach: (1) devise an approach to link a comment to all review changes that it caused and (2) increase the amount of *triggered by review changes* in the dataset.

## 8 Threats to validity

We discuss the threats to the validity of our study and the strategies we put in place to mitigate them.

**Construct Validity.** In **RQ<sub>1</sub>**, the validity of our analyses might have been threatened by the correctness of the changes dataset that we created manually. The classification of changes could have been negatively impacted by the type of experience and subjectivity of the rater. To assess the extent of this threat, we computed the joint inter-rater agreement achieving a result of 90% for the categories and 75% for the types (as explained in Section 4.1). To further corroborate these results, we computed Krippendorff's alpha coef-

ficient obtaining a value of 0.447 for the categories and 0.673 for the types. As reported in Section 4.1, the alpha coefficient for the categories reveals only a moderate agreement between the two raters.

Krippendorff’s alpha takes into account the possibility that an agreement is reached just by randomly assigning a label. Therefore, dataset with significant unbalances between the number of elements per label are likely to achieve a low alpha coefficient. Unfortunately, this is an intrinsic property of the phenomenon under analysis: Previous research reported that the number of evolvability changes during code review is significantly higher than the number of functional changes [17, 58]. However, the moderate agreement reached computing Krippendorff’s alpha might still constitute a threat to the goodness of our dataset.

Another potential threat is related to the selection of the independent variables used to build our automated approach. We exploited a set of well-know features presented in previous work and covering different aspects of source code quality, understandability, and textual coherence [23, 59, 66]. We cannot rule out that other metrics, not considered in the study, could provide additional contributions to the performance of the machine learning model.

In this study, we devised our own linking approach to connect related code chunks. However, other approaches (*e.g.*, the one offered by *java-diff-utils*<sup>4</sup>) might have been valid alternatives. The use of a different approach might modify the performance of our classification approach. Future work can investigate this further.

To conduct semi-structured interviews in the context of **RQ**<sub>2.1</sub>, we first produced a field guide following well-established guidelines [73] to maximize the rigor of the study. Nevertheless, during their execution, interviewees may provide insights in different manners, *e.g.*, by answering certain predefined questions while discussing others. For this reason, we adapted the interview schema based on the discussions we had with developers; at the same time, we made sure that all the predefined questions were addressed and asked further opinions in case these were not clear enough.

**Conclusion Validity.** In **RQ**<sub>1</sub>, a first threat is related to the interpretation of the performance achieved by the approach. We mitigated this problem by considering more than one evaluation metric (*e.g.*, F-measure, AUC-ROC, and MCC). Also concerning **RQ**<sub>1</sub>, previous work has shown the importance of considering data pre-processing actions to properly set machine learning models [40, 69, 86, 88]. For this reason, in our research we considered the application of techniques to deal with data normalization, feature selection, data balancing, and hyper-parameters configuration. In addition, the validation strategy may be object of discussion: according to a recent paper [87], the 10-fold cross-validation may bias the interpretation of the results, as it relies on random splitting of the data used to train and test the model. To mitigate the effect of randomness, we repeated the validation ten times and reported the mean performance coming from running the model multiple times.

---

<sup>4</sup> <https://github.com/java-diff-utils/java-diff-utils>

Our dataset of review changes presents a significant unbalance between the number of evolvability and functional changes. This reflects the ratio of these two categories of changes in real world code reviews [17, 58]. Nonetheless, this unbalance in our data increases the risk for our model to suffer from overfitting. To mitigate this bias, we applied the following techniques: (1) We performed features selection, evaluating the contribution of each feature both in terms of gain ratio and Pearson’s correlation with the class to be predicted; (2) We oversampled the minority class using SMOTE, applying it only on the training set and never on the test set (following the best practices reported by Santos et al. [80]); (3) We evaluated the performance of our model using 10-times 10-fold cross validation; (4) We tested our model using random forest, whose *bagging* feature reduces the chance of overfitting.

Code review is a process conducted in chronological order. The chronological nature of code review might have an effect on the performance of our classifier. We believe this is not a concern in our investigation since we do not rely on metrics that might be impacted by the chronological nature of code review and we treat each review modification independently. Nonetheless, we test the possible effect of the chronological order by comparing the performance achieved by our approach when evaluated using leave-one-out and using “*chronological-ordered leave-one-out*”. In the first approach, we iteratively use each modification in our dataset as a test set and train the model using all other modifications. We randomize the modifications in our dataset to control for any chronological effect on the performance. In the “*chronological-ordered leave-one-out*” approach, we ordered each modification using the timestamp associated to the revision from which it was extracted. We begin our evaluation by using half of the ordered modifications as training set and the next ordered modification as test set. Then, we add this modification to the training set and we select the next one as test set. We proceed iteratively until no modifications are left. Comparing the performance of the two approaches, we do not notice any significant difference. This result seems to confirm that the performance of our classifier is not influenced by the temporal nature of code review.

In **RQ<sub>2.1</sub>** to reduce the subjectivity of the evaluation of the interview data, first one of the authors did the analysis, then a second double-checked the results. In **RQ<sub>2.2</sub>**, we assessed the developers’ opinions on the generated report by also relying on open questions. To reduce subjectivity, two authors conducted the analysis of the answers.

**External Validity.** Concerning the generalizability of the results, our study considers three open-source projects. Although we considered projects that (1) come from different domains, (2) have different size as well as number of contributors, and (3) have a different organization structure, we cannot confirm that the results we achieved with our automated approach will hold in other systems, for example using other programming languages and in a closed-source setting. In the context of **RQ<sub>2.1</sub>**, only one female developer participated in the semi-structured interviews. Previous works found that gender might

influence how people approach the solution of problems [16, 22]. Therefore, we can not completely exclude that this factor might have influenced the collected answers.

Participants' answers in **RQ<sub>2.1</sub>** might have been influenced by moderator acceptance bias. To mitigate this issue, we emphasized to participants the preliminary nature of our investigation: Our aim was not to collect information on a tool we created but simply to understand the potential use and importance of information on review changes. Participants were not made aware of the approach developed in the context of our **RQ<sub>1</sub>**. Moreover, we clarified to the interviewees how the tool UI views shown in the interview slides were only meant as examples of applications of this kind of information in practice and did not represent the UI of an existing tool.

We received 17 answers when inquiring developers in **RQ<sub>2.2</sub>**. In this respect, we targeted original developers with the aim of gathering insights from people who are expert in the code review processes analyzed. As such, the audience of our study was limited by nature.

## 9 Conclusion

We presented a study that evaluates the classification of review changes as a mean to support developers. The study first addresses the scalability issues of previously proposed approaches, caused by their manual nature, investigating and evaluating a machine learning-based technique capable of automatically classifying review changes. To achieve this goal, the designed approach works using changes types and categories, based on the taxonomy proposed by previous studies [17, 44, 58]. Our best configuration achieved an AUC-ROC of 0.91 in classifying the changes categories (evolvability and functional) and an AUC-ROC between 0.91 and 0.95 for the detection of the four changes groups, obtained further refining the two main categories.

Then, we explored the use of information on review changes with twelve developers using semi-structured interviews. Participants provided positive feedback and ideas for refinement, confirming the goodness of our investigation and opening interesting directions for future work.

Finally, the relevance of review changes data for practitioners was evaluated by generating reports for 20 different open source projects, which were assessed by 17 core developers. The answers to our report confirmed the novelty of the displayed information on review changes: the respondents declared of not being aware of any tool providing the same information. Furthermore, they confirmed managers as the main potential target.

Overall, the results of our investigation give clear indication that using information on review changes to assess and improve current review practices is feasible and well perceived by developers.

## Acknowledgment

The authors would like to thank the anonymous reviewers for their thoughtful and important comments and gratefully acknowledge the support of the Swiss National Science Foundation through the SNF Projects No. PP00P2\_170529.

## References

1. Crucible official website. <https://www.atlassian.com/software/crucible>, 2019.
2. Gerrit Code Review. <https://www.gerritcodereview.com>, 2019.
3. GitHub official website. <https://github.com>, 2019.
4. About qt. [https://wiki.qt.io/About\\_Qt](https://wiki.qt.io/About_Qt), 2019.
5. U. Abelein and B. Paech. Understanding the influence of user participation and involvement on system success—a systematic mapping study. *Empirical Software Engineering*, 20(1):28–81, 2015.
6. Android. Android gerrit online repository. <https://git.eclipse.org/r/q/status:open+-is:wip>, August 2020.
7. S. Arlot, A. Celisse, et al. A survey of cross-validation procedures for model selection. *Statistics surveys*, 4:40–79, 2010.
8. A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, pages 712–721, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL <https://doi.org/10.1109/ICSE.2013.6606617>.
9. R. Baeza-Yates, B. Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.
10. T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy. If your version control system could talk. In *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, volume 11, 1997.
11. T. Baum, O. Liskin, K. Niklas, and K. Schneider. A faceted classification scheme for change-based industrial code review processes. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 74–85, Aug 2016. doi: 10.1109/QRS.2016.19.
12. T. Baum, H. Leßmann, and K. Schneider. The choice of code review process: A survey on the state of the practice. In *Product-Focused Software Process Improvement*, pages 111–127, Cham, 2017. Springer International Publishing. ISBN 978-3-319-69926-4.
13. T. Baum, K. Schneider, and A. Bacchelli. Associating working memory capacity and code change ordering with code review performance. *Empirical Software Engineering*, pages 1–37, 2019. URL <https://doi.org/10.1007/s10664-018-9676-8>.
14. G. Bavota and B. Russo. Four eyes are better than two: On the impact of code reviews on software quality. In *2015 IEEE International Conference*

- on *Software Maintenance and Evolution (ICSME)*, pages 81–90. IEEE, 2015.
15. O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering*, 21(3):932–959, 2016.
  16. L. Beckwith, C. Kissinger, M. Burnett, S. Wiedenbeck, J. Lawrance, A. Blackwell, and C. Cook. Tinkering and gender in end-user programmers’ debugging. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 231–240, 2006.
  17. M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 202–211, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2863-0. doi: 10.1145/2597073.2597082. URL <http://doi.acm.org/10.1145/2597073.2597082>.
  18. N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 308–318, 2008.
  19. C. Bird, T. Carnahan, and M. Greiler. Lessons learned from building and deploying a code review analytics platform. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 191–201. IEEE, 2015.
  20. C. M. Bishop. *Pattern recognition and machine learning*. springer, 2006.
  21. A. Bosu and J. C. Carver. Impact of developer reputation on code review outcomes in oss projects: An empirical investigation. In *Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement*, page 33. ACM, 2014.
  22. M. M. Burnett, L. Beckwith, S. Wiedenbeck, S. D. Fleming, J. Cao, T. H. Park, V. Grigoreanu, and K. Rector. Gender pluralism in problem-solving software. *Interacting with computers*, 23(5):450–460, 2011.
  23. R. P. L. Buse and W. R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, July 2010. ISSN 0098-5589. doi: 10.1109/TSE.2009.70. URL <http://dx.doi.org/10.1109/TSE.2009.70>.
  24. G. Chandrashekar and F. Sahin. A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1):16–28, 2014.
  25. N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.
  26. Couchbase. Couchbase gerrit online repository. <http://review.couchbase.org/q/status:open>, August 2020.
  27. J. Czerwonka, M. Greiler, and J. Tilford. Code reviews do not find bugs: how the current code review best practice slows us down. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 27–28. IEEE Press, 2015.

28. M. Di Penta, L. Cerulo, and L. Aversano. The life and death of statically detected vulnerabilities: An empirical study. *Information and Software Technology*, 51(10):1469–1484, 2009.
29. P. M. Domingos. A few useful things to know about machine learning. *Commun. acm*, 55(10):78–87, 2012.
30. Eclipse. Eclipse gerrit online repository. <https://git.eclipse.org/r/q/status:open+-is:wip>, August 2020.
31. C. Elkan. The foundations of cost-sensitive learning. In *International joint conference on artificial intelligence*, volume 17, pages 973–978. Lawrence Erlbaum Associates Ltd, 2001.
32. M. Fagan. Design and code inspections to reduce errors in program development. In *Software pioneers*, pages 575–607. Springer, 2002.
33. A. Fink. *How to design survey studies*. Sage, 2003.
34. B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering*, 33(11):725–743, 2007.
35. E. Fregnan, F. Petruccio, L. Di Geronimo, and A. Bacchelli. What happens in my code reviews? replication package. <https://doi.org/10.5281/zenodo.5592254>, 2020.
36. W. Gata, G. Grand, R. Fatmasari, B. Baharuddin, Y. E. Patras, R. Hidayat, S. Tohari, and N. K. Wardhani. Prediction of teachers’ lateness factors coming to school using c4. 5, random tree, random forest algorithm. In *2nd International Conference on Research of Educational Administration and Management (ICREAM 2018)*, pages 161–166. Atlantis Press, 2019.
37. E. Giger, M. D’Ambrosio, M. Pinzger, and H. C. Gall. Method-level bug prediction. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 171–180. IEEE, 2012.
38. M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.
39. M. A. Hall. Correlation-based feature selection for machine learning. 1999.
40. J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan. The impact of correlated metrics on the interpretation of defect models. *IEEE Transactions on Software Engineering*, 2019.
41. B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don’t software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681. IEEE Press, 2013.
42. R. B. Johnson and A. J. Onwuegbuzie. Mixed methods research: A research paradigm whose time has come. *Educational researcher*, 33(7):14–26, 2004.
43. Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *2010 IEEE international conference on software main-*



- tenance, pages 1–10. IEEE, 2010.
44. C. Kaner, J. Falk, and H. Q. Nguyen. *Testing Computer Software Second Edition*. Dreamtech Press, 2000.
  45. A. G. Karegowda, A. Manjunath, and M. Jayaram. Comparative study of attribute selection using gain ratio and correlation based feature selection. *International Journal of Information Technology and Knowledge Management*, 2(2):271–277, 2010.
  46. C. F. Kemerer and M. C. Paulk. The impact of design and code reviews on software quality: An empirical study based on psp data. *IEEE transactions on software engineering*, 35(4):534–550, 2009.
  47. O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey. Investigating code review quality: Do people and participation matter? In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 111–120, Sep. 2015. doi: 10.1109/ICSM.2015.7332457.
  48. S. Kotsiantis, D. Kanellopoulos, and P. Pintelas. Data preprocessing for supervised learning. *International Journal of Computer Science*, 1(2):111–117, 2006.
  49. V. Kovalenko, N. Tintarev, E. Pasyukov, C. Bird, and A. Bacchelli. Does reviewer recommendation help developers? *IEEE Transactions on Software Engineering*, 46(7):710–731, 2020. URL <https://doi.org/10.1109/TSE.2018.2868367>.
  50. B. Krawczyk. Learning from imbalanced data: open challenges and future directions. *Progress in Artificial Intelligence*, 5(4):221–232, 2016.
  51. K. Krippendorff. Computing krippendorff’s alpha-reliability. 2011.
  52. L. Kumar, S. M. Satapathy, and L. B. Murthy. Method level refactoring prediction on five open source java projects using machine learning techniques. In *Proceedings of the 12th Innovations on Software Engineering Conference (formerly known as India Software Engineering Conference)*, pages 1–10, 2019.
  53. S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.
  54. R. Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.
  55. R. Longhurst. Semi-structured interviews and focus groups. *Key methods in geography*, 3:143–156, 2003.
  56. E. E. Lyons and A. E. Coyle. *Analysing qualitative data in psychology*. Sage Publications Ltd, 2007.
  57. T. Mahboob, S. Irfan, and A. Karamat. A machine learning approach for student assessment in e-learning using quinlan’s c4. 5, naive bayes and random forest algorithms. In *2016 19th International Multi-Topic Conference (INMIC)*, pages 1–8. IEEE, 2016.
  58. M. V. Mäntylä and C. Lassenius. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, 35(3):430–448, May 2009. ISSN 0098-5589. doi: 10.1109/TSE.2008.71.

59. T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.
60. S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 192–201. ACM, 2014.
61. S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, 2016.
62. R. Morales, S. McIntosh, and F. Khomh. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 171–180. IEEE, 2015.
63. K. E. Newcomer, H. P. Hatry, and J. S. Wholey. Conducting semi-structured interviews. *Handbook of practical program evaluation*, 492, 2015.
64. A. Ouni, R. G. Kula, and K. Inoue. Search-based peer reviewers recommendation in modern code review. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 367–377. IEEE, 2016.
65. M. Paixao, J. Krinke, D. Han, and M. Harman. Crop: Linking code reviews to source code changes. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 46–49, May 2018.
66. F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman. A textual-based technique for smell detection. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10, May 2016. doi: 10.1109/ICPC.2016.7503704.
67. J. Pantiuchina, G. Bavota, M. Tufano, and D. Poshyvanyk. Towards just-in-time refactoring recommenders. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 312–3123. IEEE, 2018.
68. L. Pascarella, D. Spadini, F. Palomba, M. Bruntink, and A. Bacchelli. Information needs in contemporary code review. *Proc. ACM Hum.-Comput. Interact.*, 2(CSCW):135:1–135:27, Nov. 2018. ISSN 2573-0142. doi: 10.1145/3274404. URL <http://doi.acm.org/10.1145/3274404>.
69. F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia. Comparing heuristic and machine learning approaches for metric-based code smell detection. In *Proceedings of the 27th International Conference on Program Comprehension*, pages 93–104. IEEE Press, 2019.
70. A. Porter, H. Siy, and L. Votta. A review of software inspections. volume 42 of *Advances in Computers*, pages 39 – 76. Elsevier, 1996. doi: [https://doi.org/10.1016/S0065-2458\(08\)60484-2](https://doi.org/10.1016/S0065-2458(08)60484-2). URL <http://www.sciencedirect.com/science/article/pii/S0065245808604842>.
71. A. Porter, H. Siy, A. Mockus, and L. Votta. Understanding the sources of variation in software inspections. *ACM Transactions on Software En-*

- gineering and Methodology (TOSEM)*, 7(1):41–79, 1998.
72. M. F. Porter. Readings in information retrieval. chapter An Algorithm for Suffix Stripping, pages 313–316. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 1-55860-454-5. URL <http://dl.acm.org/citation.cfm?id=275537.275705>.
  73. S. Portigal. *Interviewing users: how to uncover compelling insights*. Rosenfeld Media, 2013.
  74. A. Ram, A. A. Sawant, M. Castelluccio, and A. Bacchelli. What makes a code change easier to review: An empirical investigation on code change reviewability. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018*, pages 201–212, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5573-5. doi: 10.1145/3236024.3236080. URL <http://doi.acm.org/10.1145/3236024.3236080>.
  75. Y. Reich and S. Barai. Evaluating machine learning models for engineering problems. *Artificial Intelligence in Engineering*, 13(3):257–272, 1999.
  76. P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212. ACM, 2013.
  77. P. C. Rigby, D. M. German, L. Cowen, and M.-A. Storey. Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):35, 2014.
  78. C. Sadowski, J. Van Gogh, C. Jaspan, E. Soderberg, and C. Winter. Tri-corder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 598–608. IEEE, 2015.
  79. C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018.
  80. M. S. Santos, J. P. Soares, P. H. Abreu, H. Araujo, and J. Santos. Cross-validation for imbalanced datasets: Avoiding overoptimistic and overfitting approaches [research frontier]. *IEEE Computational intelligence magazine*, 13(4):59–76, 2018.
  81. C. Sauer, D. R. Jeffery, L. Land, and P. Yetton. The effectiveness of software development technical reviews: A behaviorally motivated program of research. *IEEE Transactions on Software Engineering*, 26(1):1–14, 2000.
  82. B. Shibuya and T. Tamai. Understanding the process of participating in open source communities. In *Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, pages 1–6. IEEE Computer Society, 2009.
  83. D. Spadini, M. Aniche, M.-A. Storey, M. Bruntink, and A. Bacchelli. When testing meets code review: Why and how developers review tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 677–687. IEEE, 2018. URL <https://doi.org/10.1145/3180155.3180192>.

84. D. Spadini, F. Palomba, T. Baum, S. Hanenberg, M. Bruntink, and A. Bacchelli. Test-driven code review: an empirical study. In *Proceedings of the 41st International Conference on Software Engineering*, pages 1061–1072. IEEE Press, 2019. URL <https://doi.org/10.1109/ICSE.2019.00110>.
85. S. Strüder, M. Mukelabai, D. Strüber, and T. Berger. Feature-oriented defect prediction. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A*, pages 1–12, 2020.
86. C. Tantithamthavorn and A. E. Hassan. An experience report on defect modelling in practice: Pitfalls and challenges. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 286–295. ACM, 2018.
87. C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18, 2016.
88. C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, 2018.
89. P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Investigating code review practices in defective files: An empirical study of the qt system. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 168–179. IEEE Press, 2015.
90. P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 141–150. IEEE, 2015.
91. P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th international conference on software engineering*, pages 1039–1050. ACM, 2016.
92. C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall. Context is king: The developer perspective on the usage of static analysis tools. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 38–49. IEEE, 2018.
93. C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 2019.
94. C. Vassallo, S. Proksch, H. C. Gall, and M. Di Penta. Automated reporting of anti-patterns and decay in continuous integration. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 105–115. IEEE, 2019.
95. K. Wiegers. *Peer Reviews in Software: A Practical Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0-201-73485-0.

- 
96. T. Wolf, A. Schroter, D. Damian, and T. Nguyen. Predicting build failures using social network analysis on developer communication. In *2009 IEEE 31st International Conference on Software Engineering*, pages 1–11. IEEE, 2009.
  97. L. Yujian and L. Bo. A normalized levenshtein distance metric. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1091–1095, June 2007. ISSN 0162-8828. doi: 10.1109/TPAMI.2007.1078.
  98. M. B. Zanjani, H. Kagdi, and C. Bird. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering*, 42(6):530–543, 2015.