Pós-Graduação em Ciência da Computação

Felipe Ebert

Understanding Confusion in Code Reviews

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2019

Felipe Ebert

Understanding Confusion in Code Reviews

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Engenharia de Software
Orientador: Fernando José Castor de Lima Filho
Coorientador: Alexander Serebrenik

Recife
2019

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

E16u Ebert, Felipe
Understanding confusion in code reviews / Felipe Ebert. – 2019.
127 f.: il., fig., tab.

Orientador: Fernando José Castor de Lima Filho.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2019.
Inclui referências e apêndice.

1. Engenharia de software. 2. Revisão de código. I. Lima Filho, Fernando José Castor de (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2019-026

Tese de Doutorado apresentada por **Felipe Ebert** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**Understanding Confusion in Code Reviews**" **Orientador: Fernando José Castor de Lima Filho** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Paulo Henrique Monteiro Borba
Centro de Informática / UFPE

Prof. Leopoldo Motta Teixeira
Centro de Informática/ UFPE

Prof. Flávia de Almeida Barros
Centro de Informática/ UFPE

Prof. João Arthur Brunet Monteiro
Dpto de Sistemas e Computação / UFCG

Prof. Uirá Kulesza
Dpto. de Informática e Matemática Aplicada / UFRN

Prof. **Orientador**: Fernando José Castor de Lima Filho
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 15 de Fevereiro de 2019.

Prof. Ricardo Bastos Cavalcante Prudêncio
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Eu dedico esta tese à minha esposa, à minha mãe e ao meu pai.

ACKNOWLEDGEMENTS

I would first like to thank my wife Camila. She has been the most important person in this process. She has always been at my side, giving me all kinds of support, being patient and caring. She has always been the first audience of all the presentations I have given. For this and for everything else, I am forever very grateful to her, I love you!

Eu queria agradecer primeiramente à minha esposa, Camila. Você foi a pessoa mais importante nesse processo. Você sempre esteve ao meu lado, me dando todo tipo de apoio, sendo paciente e carinhosa. Além disso, você sempre foi a primeira plateia de todas as apresentações que eu fiz. Por isso e muito mais, sou muito grato a você, te amo!

To my family, especially to my parents, I thank you from the bottom of my heart for my upbringing, for all the love you have provided, for always believing in me. I don't know what I would be without you, I love you!

À minha família, especialmente aos meus pais. Agradeço do fundo do meu coração por toda a criação que vocês me deram, por todo o amor fornecido, e por acreditarem sempre em mim. Eu não seria o que sou sem vocês, amo vocês!

To my advisor Fernando Castor. For your dedication and patience, for your brilliant ideas, and above all else, for your ever-captivating positivity. For your calm and collectiveness in every stressful moment of madness that I have faced. And not only for your guidance in the doctorate, but also for you friendship throughout all these years, thank you very much!

Ao meu orientador, Fernando Castor. Pela sua dedicação e paciência, pelas suas ideias sempre brilhantes, e acima de tudo, pelo seu positivismo sempre empolgante. Pela sua calma e moderação em todo momento de stress e loucura que enfrentei. Não somente pela sua orientação no doutorado, mas também pela sua amizade durante todos esses anos, muito obrigado!

To my "co"-advisor, Alexander Serebrenik. There are no words to thank you for all your support during my doctorate. If it were not for you, I would not have had the opportunity not only to live in The Netherlands, but to travel the world. I am quite sure that you are an incredible person as well as an extremely competent professional. I feel fortunate and honored to have been able to work with you, спасибо вам большое!

Ao meu "coorientador", Alexander Serebrenik. Não existem palavras para agradecer todo o seu suporte durante o meu doutorado. Se não fosse por você, eu não teria tido, não somente a oportunidade de morar na Holanda, mas de viajar o mundo. Tenho plena certeza que você é uma pessoa incrível, assim

como um profissional de extrema competência. Me sinto afortunado e honrado em poder trabalhar com você, muito obrigado!

I cannot fail to thank Mark van den Brand and his entire team at Eindhoven University of Technology (TU/e). I will always be grateful for the opportunity you provided me with to do part of my doctorate in The Netherlands and for all the support during that time, *dank u wel*!

Não poderia deixar de agradecer a Mark van den Brand e a toda sua equipe da Universidade de Tecnologia de Eindhoven (TU/e). Serei sempre grato pela oportunidade de fazer o doutorado sanduíche na Holanda e por todo o suporte durante esse período, muito obrigado!

To Professor Nicole Novielli. Your guidance and help have been of paramount importance to my thesis. I have learned a lot from you during this period, *grazie mille*!

À professora Nicole Novielli. Sua orientação e ajuda foi de extrema importância para a minha tese. Aprendi muito durante todo esse período em parceria com você, muito obrigado!

To all those who have helped me on my thesis with the card sorting: Weslley Torres, Wellington Oliveira, Tianyu Liu, Fernando, Alexander, and Nicole. I am very grateful to all of you for your effort and patience in classifying the huge number of comments. In particular, I thank Wesley for always being able to count on as a friend during the period in The Netherlands, as well as his help in reviewing my thesis.

A todos que ajudaram na minha tese com o card sorting: Weslley Torres, Wellington Oliveira, Tianyu Liu, Fernando, Alexander, e Nicole. Sou muito grato a todo o seu esforço e paciência para classificarmos aquele enorme número de comentários. Em especial, agradeço a Weslley, por poder sempre contar com um amigo durante o período na Holanda, assim como a sua ajuda na revisão da minha tese.

To my Brazilian friends, who have always been around. Our encounters are always joyful and fun.

Aos meus amigos brasileiros, que sempre estiveram por perto. Todos os nossos encontros sempre regados a descontração e alegria.

To my friends in The Netherlands. It is impossible to name all the names, but I am grateful for all your help and friendship. I am very happy that you are all still my friends.

Aos meus amigos da Holanda. É impossível citar todos os nomes, mas sou muito grato por toda ajuda e parceria. Fico muito feliz em poder ainda contar ainda com estas amizades.

To my friend David Hunt, for his excellent work in reviewing my thesis.

Ao meu amigo David Hunt, pelo seu excelente trabalho na revisão da minha tese.

Finally, I thank CIn, UFPE, FACEPE, CAPES, and TU/e for funding this research.

Finalmente, agradeço ao CIn, a UFPE, a FACEPE, a CAPES e a TU/e por financiarem esta pesquisa.

"É preciso força pra sonhar e perceber
Que a estrada vai além do que se vê"
— (CAMELO, 2003)

ABSTRACT

Code review is a technique of systematic examination of a code change. It is an important practice for software quality assurance. The benefits of code reviews are well-known, such as decreasing the number of defects, improving software quality, and knowledge transfer. Nevertheless, they can also incur costs on software development projects as they can delay the merge of a code change and, consequently, slow down the overall development process. Furthermore, performing a code review might not be such an easy task, it will probably require developers' knowledge about the code change and the context of the system. Hence, the merge of a code change can be further delayed if reviewers experience difficulties in understanding the change. In fact, understanding the code change and its context is one of the main issues reviewers face during a code review. In this thesis, we tackle two important problems related to *confusion* in code reviews: the *lack of knowledge* in the research community about confusion in code reviews; and the *lack of tools* for confusion identification in code review comments. In the first study, we address the first problem: we create an understanding of *what* constitutes confusion by building a definition of confusion, and a confusion coding scheme. Then, we manually annotate several code review comments and build an automated approach for detecting confusion to address the second problem. Our classifiers present a considerable performance on the classification of *confusion*. Moreover, to improve the current understanding on confusion in code reviews, we conduct a second study aiming at identifying the reasons for confusion, its impacts, and how developers cope with confusion. As such, we re-annotate the aforementioned code review comments and conduct a survey of developers. Based on our findings, we provide a model of *confusion in context* with 30 reasons for confusion, 14 impacts, and 13 coping strategies. The most frequent reasons for confusion are: *missing rationale*, and *discussion of non-functional* requirements of the solution. The most popular impacts of confusion are: the *delay* on the merge decision, and the *decrease on the review quality*. The most common strategies developers adopt to cope with confusion are: *requesting information*, and *improving familiarity with existing code*. During the former studies, we observe that identification of confusion in questions is a challenging task and that *communicative intentions* are one of the reasons for confusion. Hence, we decided to conduct an in-depth analysis of the communicative intention of developers' questions in code reviews in the third study. We categorise 499 questions into 12 different categories of intentions. Even though the majority of questions actually serve *information seeking* goals, they still represent fewer than half of the annotated sample. These results suggest that questions are actually used by developers in code review to serve a wider variety of communicative purposes, including *suggestions*, *requests for action*, and *criticism*.

Keywords: Code Reviews. Confusion. Mining Software Repositories.

RESUMO

A revisão de código é uma técnica de verificação sistemática realizada em uma alteração de código. Esta é uma importante prática para a garantia de qualidade de software. Os benefícios das revisões de código são bem conhecidos, podemos citar, a redução no número de defeitos, a melhora na qualidade do software e a transferência de conhecimento. Entretanto, as revisões de código também podem gerar custos em projetos de desenvolvimento de software, pois, elas podem atrasar a integração da alteração e, consequentemente, retardar o processo de desenvolvimento. Além disso, revisar código pode não ser uma tarefa tão fácil, pois, provavelmente será exigido dos desenvolvedores o conhecimento sobre a alteração e o contexto do sistema. Assim, a integração desta alteração pode ser ainda mais retardada caso os revisores enfrentem dificuldades em compreender a alteração. De fato, entender uma alteração e o seu contexto é um dos principais problemas que os revisores enfrentam durante uma revisão de código. Nesta tese, abordamos dois importantes problemas relacionados à *confusão* em revisões de código: a *falta de conhecimento* na comunidade científica sobre confusão em revisões de código; e a *falta de ferramentas* para a identificação de confusão em comentários de revisão de código. No primeiro estudo, abordandamos o primeiro problema: criamos um entendimento sobre *o que* constitui confusão através da construção de uma definição de confusão e de um esquema de codificação de confusão. Em seguida, classificamos manualmente diversos comentários de revisões de código e criamos uma abordagem automática para detectar confusão, para abordar o segundo problema. Nossos classificadores apresentam performance considerável para a classificação de *confusão*. Complementarmente, para melhorar o entendimento atual sobre confusão em revisões de código, conduzimos um segundo estudo com o objetivo de identificar as razões da confusão, seus impactos e como os desenvolvedores lidam ela. Para isto, classificamos novamente os comentários de revisão de código e realizamos um questionário com desenvolvedores. Com base em nossos resultados, desenvolvemos um modelo de *confusão em contexto* com 30 razões para confusão, 14 impactos e 13 estratégias. As principais razões de confusão são: *motivação ausente* da alteração e *discussão dos requisitos não funcionais* da solução. Os principais impactos da confusão são: o *atraso* na integração da alteração e a *diminuição na qualidade da revisão*. As estratégias mais comuns para lidar com a confusão são: *solicitar informação* e *melhorar a familiaridade com o código existente*. Durante estes estudos, observamos que a identificação de confusão em perguntas é uma tarefa desafiadora e que as *intenções comunicativas* são uma das razões de confusão. Assim, decidimos realizar uma análise aprofundada da intenção comunicativa das perguntas dos desenvolvedores em revisões de código em nosso terceiro estudo. Nós categorizamos 499 perguntas em 12 diferentes categorias de intenções. Apesar da maioria das perguntas atender aos objetivos de *solicitação de informação*, estas ainda representam menos da metade das perguntas categorizadas. Estes resultados sugerem que as perguntas

são realmente usadas pelos desenvolvedores em revisões de código para atender a uma variedade mais ampla de propósitos comunicativos, incluindo *sugestões*, *solicitação de ações* e *criticismo*.

Palavras-chaves: Revisão de Código. Confusão. Mineração de Repositórios de Software.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Code review is a technique of systematic examination of code change, which can be conducted before or after the change is integrated into the main code repository (BACCHELLI; BIRD, 2013). Code changes submitted by a developer are reviewed by one or more of their peers. This is why code reviews are also known as *peer review* or *peer code review*. For the sake of simplicity, we use the term code review in this thesis.

Code review is an important practice for software quality assurance (TAO; KIM, 2015; BAVOTA; RUSSO, 2015; BOEHM; BASILI, 2001; MÄNTYLÄ; LASSENIUS, 2009; BARNETT et al., 2015). Several open source projects, *e.g.*, ANDROID[1], QT[2], and ECLIPSE[3], and companies, *e.g.*, MICROSOFT[4], ORACLE[5], and SAMSUNG[6], have already adopted code review as part of their development process. Likewise, several studies have also shown that code review can provide multiple benefits in the development process (BACCHELLI; BIRD, 2013; PANGSAKULYANONT et al., 2014; MORALES; MCINTOSH; KHOMH, 2015; COHEN; TELEKI; BROWN, 2006; MCINTOSH et al., 2016).

The main goals of code reviews are to find bugs in the code change, and verify whether the project guidelines and coding style are being respected (FAGAN, 1976; WIEGERS, 2002; WANG et al., 2015; BACCHELLI; BIRD, 2013; BOSU et al., 2017). Furthermore, code reviews help to improve the quality of the code on production, find better ways to implement the change, spread the knowledge about the source code, and create awareness of the changes in the project (BACCHELLI; BIRD, 2013; PANGSAKULYANONT et al., 2014; MORALES; MCINTOSH; KHOMH, 2015; COHEN; TELEKI; BROWN, 2006; MCINTOSH et al., 2016).

Despite the benefits code reviews bring, they can incur costs on software development projects, as they can delay the merge of a code change in the repository and, consequently, slowdown the overall development process (PASCARELLA et al., 2018; GREILER, 2016). The time invested by a developer in reviewing code is non-negligible (TAO; KIM, 2015) and may take up to 10%–15% of the overall time invested in software development activities (BOSU et al., 2017; COHEN; TELEKI; BROWN, 2006). Furthermore, performing a code review might not be such an easy task. In fact, understanding the code change and its context is one of the major issues reviewers face during code reviews (BACCHELLI; BIRD, 2013; COHEN; TELEKI; BROWN, 2006; TAO et al., 2012; SUTHERLAND; VENOLIA, 2009; LATOZA; VENOLIA; DELINE, 2006). The merge of a code change in the repository can be further delayed when reviewers experience difficulties in understanding the change, *i.e.*, when they are not certain of its correctness, run-time behaviour and impact on the system (COHEN; TELEKI;

[1] https://android-review.googlesource.com
[2] https://codereview.qt-project.org
[3] https://git.eclipse.org/r
[4] https://queue.acm.org/detail.cfm?id=3292420
[5] https://smartbear.com/product/collaborator/overview
[6] https://www.perforce.com/case-studies/vcs/samsung

BROWN, 2006; BACCHELLI; BIRD, 2013; TAO et al., 2012; SUTHERLAND; VENOLIA, 2009; LATOZA; VENOLIA; DELINE, 2006).

We believe that *confusion*, *i.e.*, a reviewer not being able to understand something during the code review, can affect the artifacts that developers produce and the way they work, and hence, negatively impact the development process (COHEN; TELEKI; BROWN, 2006; BACCHELLI; BIRD, 2013; TAO et al., 2012; SUTHERLAND; VENOLIA, 2009; LATOZA; VENOLIA; DELINE, 2006). For instance, the code review might take longer than it should, the quality of the review might decrease, more discussions might take place, or even the code change might be blindly accepted or summarily rejected. As such, we believe that a proper understanding of the phenomenon of *confusion* in code reviews is a necessary starting point towards reducing the cost of code reviews and enhancing the effectiveness of this practice, thereby improving the overall development process.

1.1 THE GOAL

In this thesis, we tackle two important problems related to confusion in code reviews. The first one is the *lack of knowledge* as to how confusion influences and affects the code review process. Currently, the phenomenon of confusion in code reviews is not well-understood by the community: what confusion is, what the reasons and impacts of confusion are, and how developers cope with it. Understanding confusion is crucial for researchers studying the impact of affective aspects of software development on the development process and on software itself, as well as for development teams aiming to reduce the negative consequences of code reviews, such as the delay of the development process.

The second problem, the *lack of tools* for confusion identification in code reviews, stems from the lack of knowledge about confusion. Such tools are important for identifying developers experiencing confusion, and in designing interventions to support them. The goal of this thesis is to mitigate these two problems: the lack of knowledge about confusion in code reviews, and the lack of tools for confusion identification.

Firstly, we need to understand *what confusion is*. Thus, in our first study, we propose a definition of *confusion* and create an automated approach for confusion identification, comprising three classifiers that can automatically identify confusion in code review comments. Subsequently, in the second study we focus on investigating the reasons for confusion, its impacts, and the way developers cope with it. We collect and analyse the data from surveys of developers and from code review comments. Lastly, in the third study, we deepen our focus further to investigate the communicative intention of developers' questions, *i.e.*, the talkative goal of the questions, as they are one of the causes of confusion in code reviews.

The three studies conducted in this thesis are represented according to the *degree of deepening* in Figure 1. We start broadly with the confusion identification study, then we deepen our study to understand the reasons, impacts, and coping strategies related

to confusion. Finally, we focus even in more depth on understanding the communicative intentions of the questions in code reviews.

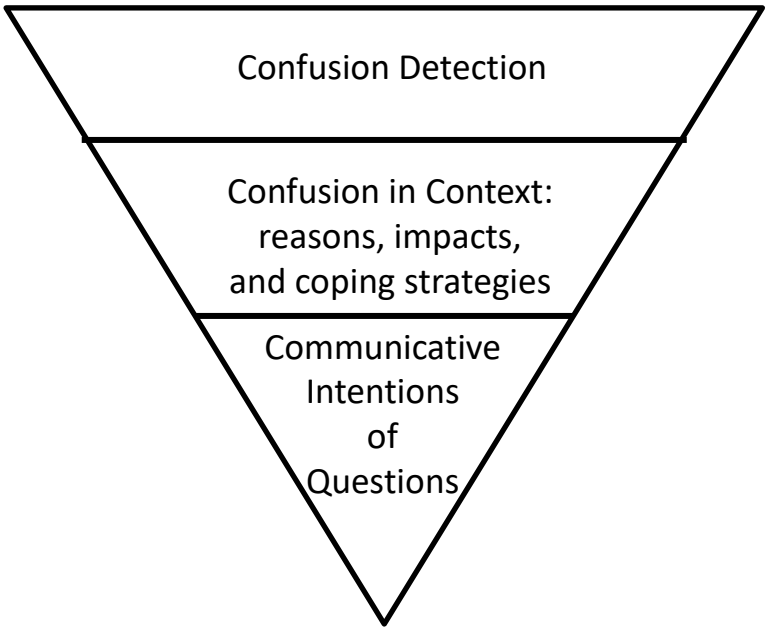

Figure 1 – Diagram of the studies of this thesis.

1.2 THE CONTRIBUTIONS

This thesis offers several contributions. Most of them provide new results and insights related to confusion in code reviews.

- **A confusion coding scheme, together with an automated approach for detecting confusion in code review comments.** To the best of our knowledge, this is the first study focused on confusion identification in code reviews. We provide a scheme for coding confusion and an approach capable of identifying confusion in developers' comments in code reviews. We also provide a gold standard set with code review comments from ANDROID, labelled as *confusion* and *no confusion*, comprising of 1,542 general and 1,190 inline comments.
- **A model of *confusion in context*, with the reasons and impacts of confusion in code reviews, as well as the strategies that developers adopt to cope with confusion.** To the best of our knowledge, this is the first study of confusion in context. Following a concurrent triangulation strategy, we build a comprehensive confusion model with 30 reasons, 14 impacts, and 13 coping strategies. We also provide a gold standard set collected from code review comments, and a series of surveys with the reasons, impacts, and coping strategies related to confusion in code reviews.

- **A series of practical and actionable suggestions to improve code review tools.** Based on the results of our second study, we provide a series of implications for tool builders to be able to improve code review tools.

- **A classification of the communicative intentions expressed in the developers' questions in code reviews.** To the best of our knowledge, this is the first study focusing on the intention of code review questions. We conduct an exploratory qualitative case study of questions in code reviews. We categorise 499 questions into 12 different categories of communicative intentions, and propose three hypotheses on the intentions of code review questions that should be confirmed or refuted by follow-up studies. The gold standard set of questions with corresponding communicative intention labels is also provided.

All the gold standard sets and classification models are publicly available[7] for research purposes (EBERT, 2019).

1.3 ORGANISATION

The remainder of this work is organised as follows.

- Chapter 2 presents the background on code reviews. We also discuss general work related to code reviews. Specific work related to each study is discussed within each chapter.

- Chapter 3 introduces the confusion definition, the confusion coding scheme, and the automated approach for detecting confusion in code review comments. It also explains the data collection and gold standard sets building process, which served all three studies conducted in this thesis.

- Chapter 4 presents a model of *confusion in context*, with the reasons for confusion, its impacts, and how developers deal with it in code reviews.

- Chapter 5 shows the study on the communicative intentions of developers' questions in code reviews.

- Chapter 6 presents the final considerations and the future work of this thesis.

The main chapters of this thesis have been published at premier Software Engineering conferences. Chapter 3 is an ICSME 2017 paper (EBERT et al., 2017) with an extended version under preparation for a premier Software Engineering journal. Chapter 4 is the distinguished award SANER 2019 paper (EBERT et al., 2019). Chapter 5 is an ICSME 2018 paper (EBERT et al., 2018). These chapters have been extended and revised while writing

[7] https://github.com/felipeebert/confusion-in-code-reviews

this thesis. Besides those papers directly related to this thesis, several papers have been published at premier Software Engineering conferences and journals (EBERT; CASTOR; SEREBRENIK, 2015; MOURA et al., 2015; REBOUÇAS et al., 2016).

2 BACKGROUND

Code review is a widely employed practice in software quality assurance: developers inspect the code changes, before or after the changes are integrated into the main repository (BACCHELLI; BIRD, 2013; BOSU et al., 2017). In this chapter, we start by presenting a short history of code review in Section 2.1. The code review process is discussed in Section 2.2. The general work related to code reviews is presented in Section 2.3. Finally, we discuss confusion, uncertainty and lack of knowledge in Section 2.4.

2.1 A BRIEF HISTORY OF CODE REVIEWS

Formal code review was first defined by Fagan in 1976 as *software inspections* (FAGAN, 1976). Software inspection, the most formal type of code review (RIGBY; BIRD, 2013), is a structured process for reviewing source code that relies on rigid roles and steps, with the single goal of finding defects (FAGAN, 1976). Firstly, prior to inspection, the code change should meet the predefined criteria. The inspection process follows these steps: planning, overview, preparation, inspection meeting, reworking, and follow-up (FAGAN, 1976).

In the planning step, one of the experts acting as a moderator assigns other experts to the roles of designer (responsible for producing the program design), coder/implementor (responsible for translating the design into source code), and tester (responsible for writing and testing the code change). Subsequently, the designer creates an inspection package which determines what is going to be inspected, and meetings are scheduled. In the inspection meeting, the code change is inspected for defects line-by-line. A written report is created with all issues raised during the inspection, and it is addressed during the reworking step. Finally, the moderator verifies whether all issues have been fixed in the follow-up.

Notwithstanding the initial success of Fagan's inspections with both the industry and research, its formality brings several drawbacks. Indeed, the inspections are very time consuming because the meetings need to be organised and the participants need to do some preparation. Another disadvantage is the chance of turning the inspection meeting into a political or social disaster (WIEGERS, 2002). Moreover, the formality of the inspection does not fit well with agile development methods (MARTIN, 2003). As a result, a more *lightweight* code review process with a better fit for test-driven and iterative development processes started to become more popular (BAKER JR., 1997; BERNHART; MAUCZKA; GRECHENIG, 2010).

Lightweight code review processes impose fewer formal requirements on the review process and on its participants, are asynchronous, usually supported by tools, and less time consuming. Such code reviews suit Open Source Software (OSS) development processes

better, because of their open collaboration format (GUZZI et al., 2013; RIGBY et al., 2015; RIGBY; BIRD, 2013). Indeed, OSS projects have been employing these lightweight code reviews for a couple of decades. In particular, this practice became very popular with the LINUX[1] operating system kernel, as documented by Raymond (RAYMOND, 1999). By now, the majority of the largest and most successful OSS projects consider code review to be the most important quality assurance practice (ASUNDI; JAYANT, 2007; NUROLAHZADE et al., 2009; RIGBY et al., 2014).

Formalising this practice, Bacchelli and Bird (BACCHELLI; BIRD, 2013) defined the lightweight code review process as a "modern code review", which is a review that is informal (as opposed to Fagan's inspections), supported by code review tools, and occurs regularly in practice. The focus of our study is on modern code reviews. We also use the term *code reviews* as a synonym for modern code reviews and discuss code reviews in further detail in Section 2.2.

2.2 THE CODE REVIEW PROCESS AND APPROACHES

Code review is an iterative process and can be instantiated in different ways. As input, a code review receives the original code change and the outcome is the reviewed change, which might be either accepted or rejected. The developer who wrote the code change is the author, and might also be responsible for submitting the change for review. The reviewer is responsible for assuring that the code change is functionally correct, meets the performance requirements, and follows the quality standards of the project.

In general, there are two types of workflow for code reviews, depending on when the review is conducted in the development process:

- **Review-then-commit (pre-commit):** the code is reviewed before it is integrated into the main repository of the Version Control System (VCS) (TICHY, 1985);
- **Commit-then-review (post-commit):** the code is reviewed after it is integrated into the main repository of the VCS (TICHY, 1985);

Since the most common type of code review is *review-then-commit* (RIGBY, 2011), it will be the focus of this thesis. We present an example of the code review process within this approach in Figure 2.

It starts with the author submitting the code change (1). The reviewers are notified and start reviewing the code change (2). They should check and verify it based on several quality criteria, such as correctness, adherence to the project guidelines, and conventions. If the reviewers believe that the code change does not fulfil those requirements, they ask the author to fix it, or to submit a new one (3). Thus, the author needs to work on the code change and submit it again (1) for review (2). When the reviewers are satisfied

[1] www.linux.org

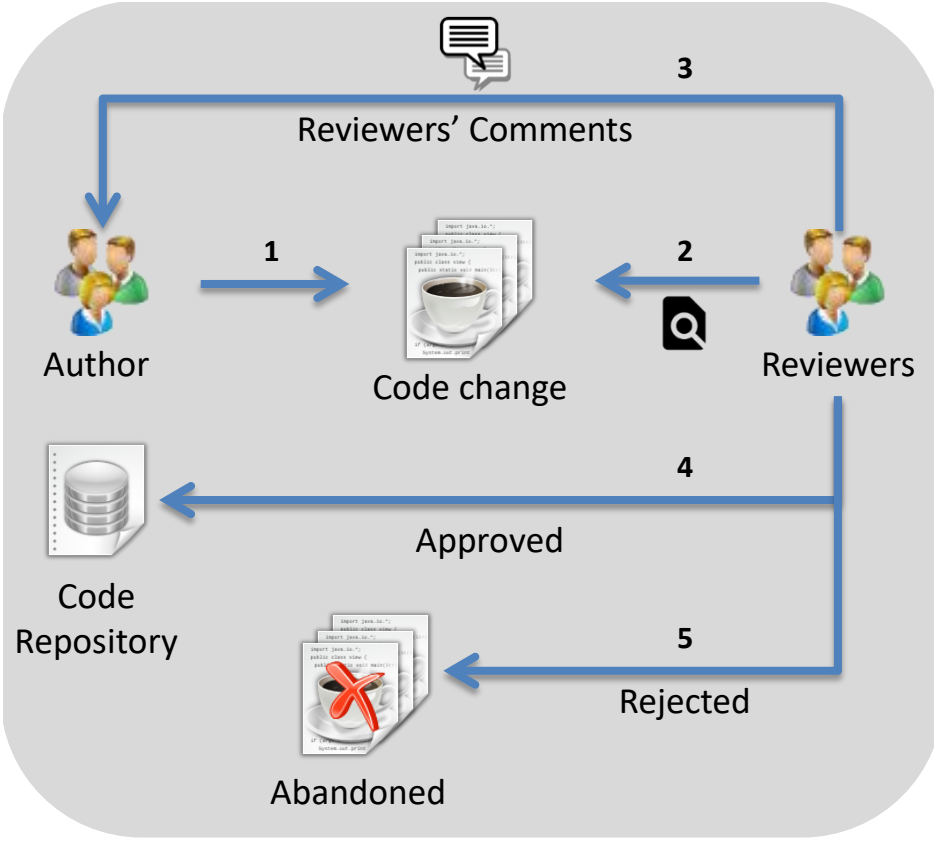

Figure 2 – Code review process.

that the code change is suitable, it is integrated into the code repository (4). However, if reviewers' quality criteria are not achieved by the code change, it is rejected, and the code review is abandoned (5). There might be several iterations before the reviewers decide to end the process (1 to 3), where the code change might be accepted (*i.e.*, it is merged into the main repository), or rejected (*i.e.*, it is discarded).

The code review process is, most usually, implemented by means of one of the following four approaches: email pass-around, over-the-shoulder, pair programming, and tool-assisted (COHEN; TELEKI; BROWN, 2006; SVOBODA, 2014; RIGBY et al., 2015). We discuss each one of these approaches in more detail in the reminder of this section.

2.2.1 Email Pass-Around

This is a traditional code review approach used in large OSS projects (RIGBY; BIRD, 2013; WANG et al., 2015; RIGBY et al., 2015). In this approach, when the author is finished working on the task, they are responsible for sending the code change to the appropriate colleagues via email. Next, each reviewer can reply to that email with comments, suggestions or questions, and, in the end, this email thread is the review, and the author needs to be up-to-date with all the emails. Email pass-around is much simpler than Fagan's inspections (FAGAN, 1976) for example, as reviewers no longer need to be physically present in the same space at the same time (SVOBODA, 2014). This facilitates involving

additional reviewers, *e.g.*, having specific expertise. Drawbacks of email pass-around are: i) the author can become overwhelmed by the email thread and easily get lost in the emails, ii) the project manager might not be certain which, and/or if, the code changes are being reviewed, and iii) the author might have problems in finding the correct files to be reviewed within the email thread (SVOBODA, 2014; WANG et al., 2015). Hence, due to such scalability difficulties in mailing lists (WANG et al., 2015), the code review process evolved and the community has started using other approaches such as bug tracking systems and specific code review tools to review code changes (RIGBY; BIRD, 2013; WANG et al., 2015; RIGBY et al., 2015).

2.2.2 Over-the-Shoulder

Similar to Fagan's inspections and, as opposed email pass-around, the over-the-shoulder code review approach assumes that the reviewer and the author are physically present in the same space at the same time. However, as opposed to Fagan's inspections, the only predefined roles are author and reviewer. When the author finishes working alone, they merely need to find a qualified colleague to review the code. The author can then explain the rationale behind their decisions. The code review can be performed at the author's or reviewer's workstation. The main advantage of this approach is the fast feedback cycle, due to face-to-face communication between the author and the reviewer (WIEGERS, 2002; SVOBODA, 2014). Its mains disadvantages are the need to be in the same place, and the extra effort required to keep the code review documentation up-to-date (WIEGERS, 2002; SVOBODA, 2014).

2.2.3 Pair Programming

Extreme Programming (XP)[2] introduced pair programming as an agile software development practice whereby two developers work on the same workstation together. As such, each developer can verify each other's work. The roles in pair programming are: the driver (responsible for implementing the code change), and the navigator (responsible for reviewing the code from the driver) (RIGBY, 2011). The developers can also change roles. As is the case with over-the-shoulder, in pair programming the developer and the reviewer are required to be in the same place. In contrast to the over-the-shoulder approach, the implementation phase and the review phase are merged. During pair programming, the code change is written and reviewed at the same time by both developers, while during over-the-shoulder, only the author implements the change, and the reviewer can see the code as a whole. This approach is particularly useful for experienced developers to mentor less experienced colleagues, and also for knowledge sharing (RIGBY, 2011; SVOBODA, 2014). The main drawback is the time consumed by two developers working on the same

[2] www.extremeprogramming.org

code change, and only the parts highlighted by the author are reviewed (RIGBY, 2011; SVOBODA, 2014). There is also a further generalisation of pair programming called *mob programming* (BUCHAN; PEARL, 2018; WILSON, 2015). In this approach, the whole team works on the same code change at the same time, space, and computer.

2.2.4 Tool-Assisted

The tool-assisted approach makes use of distributed code review tools. As detailed below, features of code review tools have been designed to address the shortcomings of the earlier code review approaches. This makes the tool-assisted approach the simplest, most efficient, and currently the most popular code review approach (SVOBODA, 2014; COHEN; TELEKI; BROWN, 2006).

- **Automated File Gathering.** This feature addresses the drawback of the email pass-around approach by including all the correct files to be reviewed in the same place, *i.e.*, in the version control system (VCS).
- **Combined Display - Differences, Comments, Defects.** This feature addresses all the above-mentioned approaches. The most time-consuming tasks during code reviews are: i) finding the differences in the source code, ii) associating the comments with a particular file or line number, and iii) understanding them. Hence, code review tools show the difference between the files, *i.e.*, before and after the change, in a way the code review comments can be threaded (these are called **general comments**) and they can also be linked to specific parts of the source code (these are called **inline comments**). Thus, the developers do not need to spend time trying to cross-reference comments, defects, and source code.
- **Automated Metrics Collection.** This feature also addresses the problem of all other approaches: to produce accurate metrics so project managers can understand and measure the review process. It is probably certain that no developer likes to have a stopwatch and line-counting tools while code reviewing. Consequently, the code review tool can collect all code review metrics without bothering the developers.
- **Review Enforcement.** This feature addresses the drawback of almost all previous approaches: the manager not knowing whether the reviewer has already reviewed the code change. Code review tools can enforce this workflow.
- **Clients and Integrations.** This feature addresses a drawback of all other approaches: the possibility for the developer to use different ways of reviewing. Code review tools can support reviews by command-line tools, integrations with Integrated Development Environments (IDE), and version control GUI clients. By providing all these kinds of support, they can satisfy the different needs of different

developers. Furthermore, the tools can integrate with continuous integration systems which can build and run tests automatically.

- **Asynchronously and non-locally.** This feature addresses the drawback of over-the-shoulder and pair programming as it does not require the developers to be in the same place. Furthermore, the communication between the author and reviewers is through asynchronous messages.

- **Automatically reviewers' selection.** This feature addresses the problem of finding the best developer to review the code change. Code review tools can indicate the appropriate reviewer based on the expertise with the modified component.

Figure 2 is an example of the tool-assisted approach. The code review tool is responsible for keeping all the information together (the source code, the commit message, the code review comments, etc.). Code review tools can also notify both authors and reviewers when there is a new event in the code review, *e.g.*, when the author submits a new code change or when there is a new message.

There are several software-based code review tools nowadays, some of them are browser-based, *e.g.*, Gerrit Code Review[3], Crucible[4], and Differential[5], and some of them integrate within standard IDE, *e.g.*, Gerrit and Review Board[6] can be integrated with Eclipse[7] and NetBeans[8], respectively. These tools can also be integrated with VCS (tichy, 1985), *e.g.*, Gerrit can be integrated with Git[9] and Crucible can integrate with, among others, Mercurial[10] and Subversion[11].

Due to the boosting of the code review tools, the OSS projects, and companies using them (rigby et al., 2012; bacchelli; bird, 2013; rigby; bird, 2013; bosu et al., 2017), we focus our study on the tool-assisted code review approach.

2.3 MODERN CODE REVIEW

Code review has been the focus of a plethora of studies (cohen; teleki; brown, 2006; bavota; russo, 2015; bacchelli; bird, 2013; tao et al., 2012; kononenko et al., 2015; hentschel; hähnle; bubel, 2016; mukadam; bird; rigby, 2013; hamasaki et al., 2013; thongtanunam et al., 2014; yang et al., 2016; wesel et al., 2017). In this section, we discuss general work related to code reviews. The following Chapters 3, 4, and 5 provide a related work discussion specific to each chapter.

[3] www.gerritcodereview.com
[4] www.atlassian.com/software/crucible
[5] www.phacility.com/phabricator/differential
[6] www.reviewboard.org
[7] www.eclipse.org
[8] https://netbeans.org
[9] https://git-scm.com
[10] www.mercurial-scm.org
[11] https://subversion.apache.org

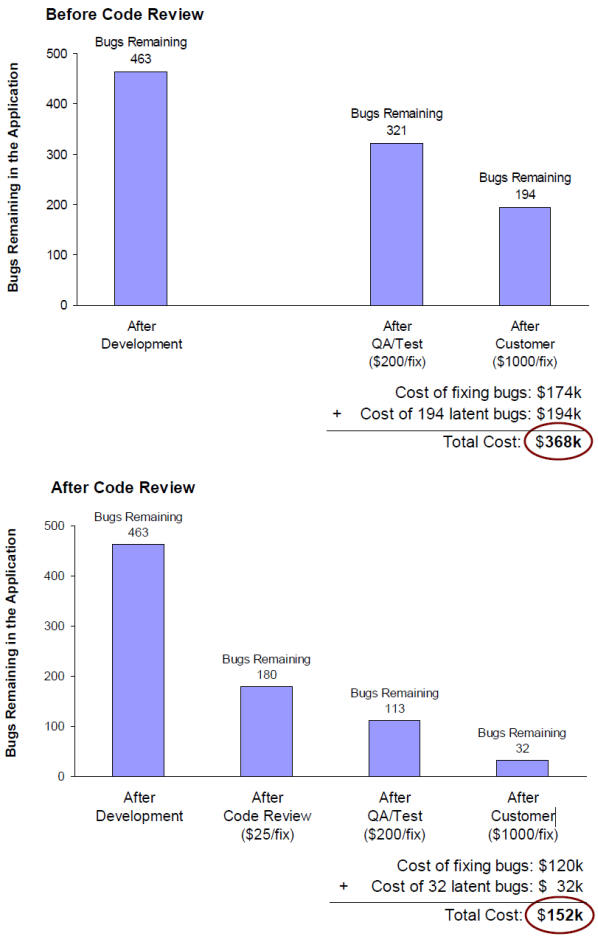

Figure 3 – A real-world study comparing the cost of development with and without code review (COHEN; TELEKI; BROWN, 2006).

Code review is a well-known practice for improving software quality. Finding and fixing bugs during code reviews before they go on production can save time and money. A real-world case study showed that the use of code review could have saved more than U$ 200.000,00 in a project (COHEN; TELEKI; BROWN, 2006). Figure 3 shows a real-world study on the difference between development with and without code review. We can observe the amount that would have been saved from the costs of fixing bugs on production. Furthermore, a code review would have found 162 additional bugs.

Bacchelli and Bird (BACCHELLI; BIRD, 2013) introduce the term *modern code review* which is supported by tools, is informal, and which happens frequently. They explore the motivations, challenges, and outcomes of code reviews by observing, interviewing, and surveying software developers. Their study shows that finding defects is not the only benefit of code reviews, knowledge transfer and team awareness are also advantages coming from reviews. They also show that the main challenge of code review is understanding the code change and its context.

Tao *et al.* (TAO; HAN; KIM, 2014) analyse the reasons why code changes are rejected at ECLIPSE and MOZILLA. They manually analyse a total of 300 rejected code changes,

conduct survey of developers, and perform a literature survey. As a result, they derive a comprehensive list of code change rejection reasons and also present guidelines for developers writing acceptable code changes. The list contains 30 reasons which are divided into five main categories: problematic implementation or solution, difficult to read or maintain, deviating from the project focus or scope, affecting the development schedule, and lack of communication or trust.

Rigby *et al.* (RIGBY et al., 2015) study several systems using a mixed-method approach to understand the code review process. They also examine the reasons why pull requests on GitHub are rejected and discover that there is no one clear outstanding reason for rejecting code changes in code reviews. The main reason for rejecting (18% of the pull requests) is superseded, *i.e.*, when another pull request has already solved the problem with a better implementation. The second most common reason, and still only 13% of pull requests, is due to technical issues.

Bavota and Russo (BAVOTA; RUSSO, 2015) investigate how code reviews influence the chance of inducing bug fixes, and the quality measured by code coupling, complexity, and readability of the code changes. They show that commits not reviewed are twice as likely to introduce defects than reviewed commits. Furthermore, the reviewed code changes have a substantially higher readability as compared to unreviewed code changes.

Kononenko *et al.* (KONONENKO et al., 2015) investigate the quality of code reviews in an OSS project by exploring the factors that might affect the reviews. They use the SZZ algorithm to find code changes that introduce defects and then relate them to the code review information. They show that 54% of the code changes that went through the review process introduced defects into the system. Furthermore, personal metrics (reviewer experience and workload) and participation metrics (number of reviewers) are associated with the quality of the code review process. Another interesting result is that the technical properties of the code change (the size, number of files changed, etc.) have a significant impact on the chance of inducing defects in the system.

Barnett *et al.* (BARNETT et al., 2015) introduce a static analysis tool for decomposing code changes to facilitate code reviews at MICROSOFT. The rationale for their technique is that the composite code change can be partitioned using *def-use* (the definition of any entity and its uses are related) and *use-use* (the uses of the same entity are related) relationships. As such, the set of diff files that are syntactically and semantically related can be grouped together. It is important to mention that their *def-use* and *use-use* relationships did not present any false positives. After the tool evaluation, they received positive feedback from developers and most of them consider the idea helpful for reviewing the code changes.

There are also several studies providing datasets of code reviews. Each one of them has a different characteristic. Mukadam *et al.* (MUKADAM; BIRD; RIGBY, 2013) provide a dataset of ANDROID containing information about code changes, reviewers, review dura-

tion, and what kinds of discussions and feedback developers usually give in code reviews. This dataset does not provide the inline comments. Hamasaki *et al.* (HAMASAKI et al., 2013) provide dataset from four OSS projects, including ANDROID. The dataset contains information related to the code review, code change, and reviewers. They do not include general and inline comments. Instead, they show only the number of each type of comment. They also provided extraction tools, so the community can access and use them to mine other datasets.

Thongtanunam *et al.* (THONGTANUNAM et al., 2014) also provide a code review dataset ANDROID and additionally, a web visualisation tool for analysing it. The tool presents the code review information from three perspectives: i) the review perspective presents the number of code reviews, reviewers, code changes, general comments, and modified files on a weekly basis; ii) the process perspective shows the number of activities (number of code reviews created, number of code changes submitted, number of reviews merged and abandoned, etc.) performed on a daily basis; and iii) the human perspective presents the activities of each developer. It also does not provide the inline comments from the reviews.

Yang *et al.* (YANG et al., 2016) present a data set from five OSS projects, including ANDROID. They present the dataset from three different perspectives: i) the people-related perspective shows personal information about reviewers; ii) the process-related perspective shows data about the review, such as review status, approval and period; and iii) the product-related perspective shows information related to the code change itself.

To the best of our knowledge, there is no data set related to confusion in code reviews. Furthermore, no other study provided inline comments in their data sets. Our study provides three different data sets of code reviews, described in the following chapters.

2.4 CONFUSION, UNCERTAINTY, AND LACK OF KNOWLEDGE

There are several studies which tried to model the *affective disequilibrium* related to confusion, uncertainty, and lack of knowledge, specially from the Psychology field. In this section, we discuss the most relevant studies on those topics.

Merriam-Webster dictionary[12] provides the following definitions of the word **confusion**:

> "*a situation in which people are uncertain about what to do or are unable to understand something clearly*"
>
> "*the feeling that you have when you do not understand what is happening, what is expected, etc.*"

[12] www.merriam-webster.com/dictionary/confusion

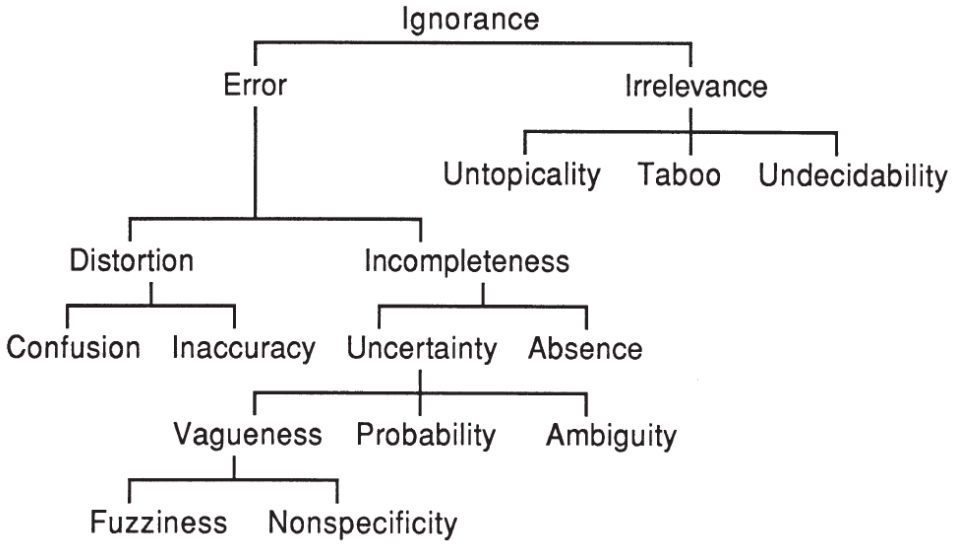

Figure 4 – The taxonomy of unknowns (SMITHSON, 1989).

Smithson (SMITHSON, 1989) defined the *taxonomy of unknowns*, which is presented in Figure 4. We will discuss only the most relevant topics from the taxonomy for this thesis. Smithson uses *ignorance* as a starting point. Next, he distinguishes passive (*error*) and active (*irrelevance*) ignorance. From his definition, *confusion*, which is a type of *distortion*, means a wrongful substitution, *i.e.*, a mistake of one thing for another. As for *uncertainty*, it is referred to partial information, *i.e.*, when knowledge is incomplete to a certain degree.

Armour (ARMOUR, 2000) suggested categorising ignorance into layers based on what we know and what we do not know. He defined the *Five Orders of Ignorance*:

- **0th Order Ignorance - Lack of Ignorance:** when we know something, *i.e.*, it is knowledge;
- **1st Order Ignorance - Lack of Knowledge:** when we do not know something, but we can easily identify that fact;
- **2nd Order Ignorance - Lack of Awareness:** when we do not know that we do not know something, *i.e.*, when we are unaware of that fact;
- **3rd Order Ignorance - Lack of Process:** when we do not know a suitably efficient way to find out we do not know that we do not know something;
- **4th Order Ignorance - Meta Ignorance:** when we do not know about the Five Orders of Ignorance.

D'Mello and Graesser (D'MELLO; GRAESSER, 2014) focused on *confusion* and how it impacts learning and problem solving. They consider confusion to be an affective state rather than an emotion. According to D'Mello and Graesser, confusion happens when an individual detects new or discrepant information, *e.g.*, there is a conflict with prior

knowledge. Thus, confusion happens when such new or discrepant information triggers an impasse on the individual, which blocks the goal and results in the person being uncertain about how to proceed next. After confusion is present, the person must deal with problem solving activities to be able to successfully restore equilibrium. To solve confusion, an individual needs to "*stop, think, effortfully deliberate, problem solve, and revise their existing mental models*" (D'MELLO; GRAESSER, 2014). In contrast, if confusion is not resolved, it can spawn negative affective states in the individual, such as frustration and anger.

Jordan *et al.* (JORDAN et al., 2012) investigated the frequency of *uncertainty expressions* in discussions of students using a computer-mediated environment. They created their own definition of *uncertainty* and provided a coding scheme to describe and model it. Despite acknowledging that defining uncertainty was not simple, their definition is:

> "*situations when individuals have a sense of wondering, doubt, or unease about how the future will unfold, what the present means, or how to interpret the past.*"

The uncertainty coding scheme defined by Jordan *et al.* (JORDAN et al., 2012) is presented in Figure 5. Uncertainty is divided into four main groups: i) indirect expression of uncertainty, ii) request for a solution, iii) direct expression of uncertainty, and iv) discussing uncertainty. The first group contains three categories of uncertainty: expressions related to the degree of truth (*hedges*), expressions related to the likelihood of occurrence of an event (*probables*), and expressions related to hypothetical scenarios (*hypotheticals*). The second group contains one category with direct and indirect *questions*. The third group contains two categories, one related to direct expression of uncertainty in the first person (*I statements*), and another related to paralinguistic indicators (*nonverbals*). The last group contains one category (*meta*) with expressions related to uncertainty in the past, or from another person.

We believe all the aforementioned definitions about *confusion*, *uncertainty*, and *lack of knowledge* are somehow connected. Lack of knowledge and confusion, which can also encompass doubt and uncertainty, are strictly linked (*e.g.*, confusion could be determined as lack of knowledge) and are both actionable (D'MELLO; GRAESSER, 2014). In the following Chapter 3, we provide our own definition of *confusion* based on this literature.

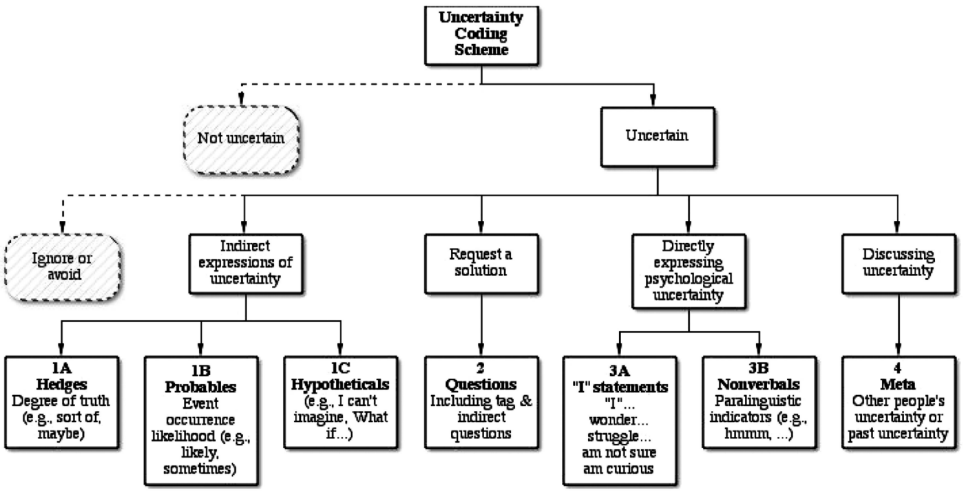

Figure 5 – The uncertainty coding scheme (JORDAN et al., 2012).

3 CONFUSION DETECTION

In this chapter, we present our first study aiming to solve the problems of the *lack of knowledge* about confusion in code reviews and the *lack of tools* for confusion identification. This study was conducted in two steps. Firstly, we performed a preliminary analysis of the feasibility of the identification of confusion in code reviews. Taking as a starting point the positive results of the preliminary study, we conducted a more comprehensive study on this topic. Hence, this chapter presents our *confusion* definition, the confusion coding scheme we built, and the automated approach we developed for detecting confusion in developers' comments in code reviews.

In Section 3.1, we discuss the overview of these studies. Next, we present our definition of confusion, and the confusion coding scheme in Section 3.2. The annotation process for building the gold standard sets is described in Section 3.3. The preliminary study is presented in Section 3.4 and the more comprehensive study in Section 3.5. Section 3.6 lists the threats to validity we identified of each study. Related work to both studies is discussed in Section 3.7. Finally, we summarise our work in Section 3.8.

3.1 OVERVIEW

Several studies point out that one of the main challenges developers face during code reviews is understanding the code change and its context (BACCHELLI; BIRD, 2013; COHEN; TELEKI; BROWN, 2006; TAO et al., 2012; SUTHERLAND; VENOLIA, 2009; LATOZA; VENOLIA; DELINE, 2006). We believe that *confusion* can negatively impact the process. Understanding confusion faced by developers in code reviews and being able to automatically identify it is an important step in improving the code review process. For example, the presence of confusion in code reviews can make it take longer than it should (*i.e.*, delay the review process), decrease the quality of the review, increase the number of discussions, or even cause the code change be blindly accepted.

In this scenario, code review tools should be able to check for confusion in developers' comments and, when confusion is present, they should trigger specific actions, such as providing documentation or requesting the intervention of the code change author. Additionally, a precise approach for identifying confusion in code reviews can help researchers build a large-scale database of comments that can be employed to identify causes, sentiments, and contexts associated with confusion. The topic of confusion in code reviews and how to identify it is still unexplored. To the best of our knowledge, this is the first study aiming to: i) understand confusion in code reviews, and ii) provide an automatic approach for confusion detection.

Our preliminary study takes the first step towards confusion detection. We used

396 general and 396 inline comments labelled as exhibiting and not exhibiting confusion to train and test several classifiers. We observed that *confusion* can be reasonably well-identified by humans. The agreement, measured with Fleiss' kappa (FLEISS, 1971), achieved a moderate value among the raters: 0.59 for the general comments, and 0.49 for the inline ones. Subsequently, we built a series of classifiers for each kind of comment, *i.e.*, general and inline. The best precision for *confusion* identification reached 0.87 for the general, and 0.61 for the inline comments. The highest values for recall reached 0.94 for the general, and 0.98 for the inline comments. Considering both precision and recall, general comments presented 0.69 and 0.54, and the inline ones 0.43 and 0.58, respectively.

These results motivated us to further research about confusion identification and its impact on the code review process. Hence, we decided to conduct a more in-depth study about confusion detection by extending the preliminary one. The dataset used to train the classifiers in the in-depth study contains three times more comments, both general and inline. Furthermore, we have separate sets for training and testing, and we have also evaluated the impact of data balancing on the training sets. Finally, as opposed to our preliminary study, we considered two important techniques from machine learning: feature selection and automated parameter tuning. The former is important because it extracts only the most relevant features for the prediction model, and the latter, because manual tuning, *i.e.*, the act of choosing the best parameters for the classifier, is not feasible (BERGSTRA; BENGIO, 2012).

3.2 CONFUSION DEFINITION AND CODING SCHEME

We use the studies discussed in Section 2.4 as inspiration to define *confusion* and to build a *confusion coding scheme*. We define *confusion* broadly as *a situation where a person is uncertain about or unable to understand something.*

To be able to model confusion, we use the uncertainty coding scheme proposed by Jordan *et al.* (JORDAN et al., 2012). Even though their scheme is intended to model uncertainty, we believe that their definition of uncertainty is broad enough to be similar to our confusion definition. Our coding scheme contains the same seven categories of the scheme by Jordan *et al.* (JORDAN et al., 2012). Each category contains keywords and expressions, *i.e.*, textual elements, related to expression of confusion:

- ***Hedges*:** expressions related to the degree of truth, *e.g.*, "maybe";
- ***Probables*.** expressions related to the likelihood of occurrence of an event, *e.g.*, "likely";
- ***Hypotheticals*:** expressions related to hypothetical scenarios, *e.g.*, "what if";
- ***Questions*:** expressions related to direct and yes/no questions, *e.g.*, "why is this here?";

- ***I Statements*:** direct expressions of confusion, *e.g.*, "I'm not sure";
- ***Nonverbals*:** expressions related to paralinguistic indicators, *e.g.*, "hmm";
- ***Meta*:** expressions related to confusion in the past or from another person, *e.g.*, "I didn't understand".

The coding scheme of Jordan *et al.* (JORDAN et al., 2012) has been designed for describing the occurrence of uncertainty expressions in discussions of graduate students using a computer-mediated system, while we aim at identifying confusion in comments of software developers using code review tools. Thus, we needed to extend the scheme of Jordan *et al.* with expressions related to our definition of confusion. We found several studies providing examples of uncertainty and confusion expressions (JORDAN et al., 2012; HOLMES, 1982; JORDAN; JR., 2014; JORDAN et al., 2014; LAKOFF, 1975; VARTTALA, 2001). Hence, we extended our coding scheme with several expressions of confusion. For instance, most of the *hedges* expressions are found in the article of Lakoff (LAKOFF, 1975). Varttala (VARTTALA, 2001) provides a comprehensive list of *probables* and *hypothetical* expressions. Holmes (HOLMES, 1982) shows some examples of *probables*. The studies of Jordan *et al.* (JORDAN et al., 2012; JORDAN; JR., 2014; JORDAN et al., 2014) provide a few examples of all categories.

However, even after adding all such expressions, the framework still missed expressions of confusion such as "*I'm not sure*" and "*I don't understand*". Therefore, we included in our confusion coding scheme some expressions of confusion based on common sense knowledge. This process was performed by four researchers on online meetings, where they discussed and agreed on confusion expressions. Furthermore, we augmented the list of expressions by adding all verb tenses for each verb, *e.g.*, "believe", "believed", and "believes", and for the adjectives and nouns, we added both the singular and plural forms, *e.g.*, "assumption", and "assumptions". The complete list of expressions of each category of our *confusion coding scheme* is presented in the Appendix A, and publicy available online (EBERT, 2019).

3.3 GOLD STANDARDS FOR CONFUSION IN CODE REVIEWS

In this section, we explain the data collection process used to build the datasets of code review comments, which were available for all studies in this thesis. These datasets were used in this study to create the gold standards for confusion in code reviews; in the study of Chapter 4 for the labelling of reasons for confusion, its impacts, and the strategies developers adopt to deal with it; and in the study of Chapter 5 for the classification of the communicative intentions of questions.

We selected ANDROID as the case study subject. We decided to focus only in one project and conduct an in-depth analysis on it, the analysis of other systems will be

part of the future work. Besides being a large and well-known OSS project, it also has a rigorous code review process and a large number of publicly available code reviews. We consider Android as a "typical" or "paradigmatic" case (FLYVBJERG, 2007) of a large open source project. ANDROID uses GERRIT[1] as its code review system. Developers submit their changes into GERRIT and invite others to review the changes. The change is merged only after having been verified and approved by a senior developer. The web interface provided by GERRIT allows reviewers to create general comments on the code review page, and inline comments in the source code file, referencing a word, a line or a group of lines.

We downloaded all code reviews from the ANDROID project using GERRIT API until November 25th, 2016, comprising a total of 140,006 code reviews, including 28,091 with inline comments. The GERRIT API provides all information about the code reviews as *.json* files. The information supplied by GERRIT includes, among others, the code review ID, name and email address of the author and reviewers, the code review status, the creation date, the number of insertions and deletions of the code change, the general and inline comments, etc.

Our original dataset contains 899,105 general comments and 232,471 inline comments. While analysing it, we identified several bots acting in the general comments, such as TREEHUGGER ROBOT, DECKARD AUTOVERIFIER, and ANDROID MERGER. Thus, we decided to exclude the 238,260 bot comments from our dataset, leaving 660,845 general comments. There are no bot messages in the inline comments. These results are presented in the first step of Figure 6.

In the second step, we used the confusion coding scheme to filter out comments that do not contain expressions related to confusion. This way, we optimised our approach to identify confusion in order to improve recall. To implement this step, we used APACHE SOLR[2] to tokenise the comments and to store all code review comments from our dataset. We then ran queries on it for each feature of our scheme. As a result of this filtering process, we kept 91,658 general and 116,292 inline comments, shown in the second step of Figure 6.

We started the annotation process with a random sample of 25 general comments of the *hedges* category for training purposes. The first author classified them as *confusion* or *no confusion*, and then the other authors agreed on all labels of the sample. These comments were used as a guideline for the rest of the annotation process.

During the implementation of our confusion coding scheme, we split the seven categories into three groups. The rationale behind this decision is simple: all the scheme categories, except for *questions*, were implemented similarly with regular expression matching. Additionally, since **hedges** is the largest group, containing more than 97% of the general and 87% of the inline comments, we decided to separate it from the other categories.

[1] http://www.gerritcodereview.com
[2] http://lucene.apache.org/solr

Figure 6 – Gold standard sets build process: **GC**—*general comments*, **IC**—*inline comments*.

Hence, the three groups we created are: **hedges**, **questions**, and **other**. The **other** group comprises the rest of the categories of the scheme: *probables*, *hypotheticals*, *I statements*, *nonverbals*, and *meta*. Figure 6 shows this arrangement in the second step.

The **questions** group was implemented differently because regular expression matching alone would not be satisfactory. For example, just looking for question marks to identify questions would return several erroneous cases, such as URLs and code snippets identified as questions. Hence, we decided to adopt a more appropriate approach: we used the StanfordNLP API (Manning et al., 2014) to identify questions in developers' comments. Using this API we can identify the grammar structure of each sentence from the code review comment and then check if it is an affirmative sentence or a question.

To achieve a confidence level of 95% and a confidence interval fewer than 5% (Metcalfe, 2001), we sampled 400 comments of each of the groups **hedges**, **other**, and **questions**, for both general and inline comments, *i.e.*, 2,400 comments overall. During the annotation process, four raters having at least a master's degree in Computer Science manually and individually classified the comments as expressing *confusion* or *no confusion*. As a guideline, the raters were instructed not to consider as *confusion* comments that employ uncertainty in expressing politeness, *e.g.*, "Could anyone submit this?" and "Maybe add checker tests to make sure you cover the cases you intended?". Indeed, the former is a polite request and the latter makes a polite suggestion. Additionally, they were instructed to consider the comments where manifestations of *confusion* appear within *quoted* text as *no confusion*, in order to avoid duplication. The quoted texts in Gerrit are pieces of older comments to which the current comment makes a reply, and they start with the character '>'. For instance, the comment presented below from Android[3] is a case of the feature from our scheme being found in the quoted text: this is an example of *no confusion* comment.

> *> Good catch. Might make sense to enforce setting this variable for*
> *> anything that builds a shared library.*
>
> *and static libraries too :-)*

To measure the agreement in the annotation process for general and inline comments, we used Fleiss' kappa (Landis; Koch, 1977). The disagreement was resolved in two steps, first we used majority vote. Then in cases where the number of raters favouring confusion and no confusion were equal, the raters had an online meeting. There were some cases where the agreement among the raters was not possible, *i.e.*, eight comments from **hedges**, five from **other**, and two from **questions**. Hence, we decided to discard such comments. The results of the agreement and the annotated comments are presented in the second step of Figure 6. At this stage, we had a dataset (comprising the three groups **hedges**,

[3] https://android-review.googlesource.com/c/166000

other, and **questions**) of 1,136 code reviews with 1,189 general and 1,190 inline comments labelled as *confusion* and *no confusion*.

Since we started with a random sample of each group, our dataset still had code review comments that had not yet been verified, *i.e.*, there were 14,996 **remaining** unlabelled general comments from the 1,136 code reviews. We did not analyse the remaining inline comments from our dataset. This is our plan for future work. As another round of manual annotation would be infeasible on this enormous amount of comments, we decided to use a different approach, which led us to the third step (cf. Figure 6).

We consider a code review as presenting *confusion* if it contains at least one confusion comment, either general or inline. Hence, in order to be able to determine whether code reviews contain confusion, we needed to verify all the other comments from our dataset. Since our dataset contains code reviews with (at least) one *no confusion* comment, we could not argue about confusion on those code reviews before verifying their remaining comments. This is the reason why we followed a different process to build another group of code review comments using the **remaining** comments.

We started the third step by applying our confusion coding scheme to the set of 14,995 remaining comments to filter out those not containing indication of confusion, which resulted in 3,018 comments. Once again we optimised the confusion detection for recall. Next, we removed 396 comments whose features were within quoted text, avoiding repetition. This still left us with 2,622 comments to be labelled: labelling 2,622 comments manually is unfortunately not feasible due to time restrictions. At this stage of the process of building our datasets, we had already completed the preliminary study, which is discussed in Section 3.4. Hence, we decided to take advantage of the classifiers already built by such study. We used the classifier trained over the groups **hedges**, **other**, and **questions** of the general comments.

Instead of tuning the classifier with the best performance on the *confusion* class, we decided to do the opposite. We used the classifier with the best precision on the *no confusion* class to exclude as many as possible of the comments without *confusion*. Thus, we would not need to manually analyse those comments classified as *no confusion* by the classifier because it has a high precision. Hence, we could focus on the remaining comments, *i.e.*, classified as *confusion* by the classifier. The classifier used in this step was the Multinomial Naive Bayes with precision of 0.94 and recall of 0.44 for *no confusion* class. The results are presented in the third step of Figure 6, the classifier indicated 356 **remaining** comments as *confusion*, which were manually annotated by the four raters. These labelled comments were integrated into our dataset of general comments.

Finally, we built two gold standard sets: one for general comments with 350 (23%) *confusion* and 1,192 (77%) *no confusion* comments, and the other for inline comments with 270 (23%) *confusion* and 920 (77%) *no confusion* comments. Both gold standard sets are publicly available (EBERT, 2019).

3.4 CONFUSION CLASSIFIER: A PRELIMINARY STUDY

In this section, we explain our preliminary study. It is our first step towards the identification of *confusion* in developers' comments in code reviews. The main vehicle of this study is an exploratory case study (RUNESON; HÖST, 2009) on confusion in code reviews. We *aim* at understanding how developers express confusion during code reviews. As a preliminary step towards building a classifier for confusion, we assess whether confusion can be identified by humans. Hence, our *research questions* are:

- ***RQ1.*** *Can human raters agree on the presence of confusion in code review comments?*
- ***RQ2.*** *Can a tool perform similarly to humans when classifying confusion in code review comments?*

Since this is an initial study, we start considering only the **hedges** group of our dataset to answer those **RQs**. This decision is based on the fact that **hedges** is the largest group. The annotation process is described in the first and second steps of Figure 6 of Section 3.3. The **hedges** dataset contains 396 general and 396 inline comments, as four general and four inline comments have been discarded, this is the gold standard set for this study.

3.4.1 Methodology

Before training the classifiers, we first remove the line breaks and replace URLs, numbers, commits' ID and user names with meta tokens (*e.g.*, @URL, USERNAME, COMMIT and NUMBER). To identify the user names we leveraged the name list collection by Vasilescu *et al.* (VASILESCU; CAPILUPPI; SEREBRENIK, 2014). We exploit machine learning techniques using our gold standard for training and validation. We experiment with several state-of-the-art classifiers using WEKA (FRANK; HALL; WITTEN, 2016). To understand the impact of the feature choice on the classifier performance, we run the classifiers over three different models.

In the first model, *i.e.*, the *baseline*, features are uni- and bi-grams extracted by the unsupervised STRINGTOWORDVECTOR filter from WEKA, and selected based on the Term Frequency Inverse Document Frequency (TF-IDF) (RAJARAMAN; ULLMAN, 2011). The second model, *baseline + 3*, extends the *baseline* by including i) the modal verbs count, ii) the count of the hedges from our confusion coding scheme, and iii) the presence of question marks. In this model, we do not distinguish between different hedges or modal verbs. The last model, *baseline + hedges*, also adds to the *baseline* hedges from our confusion coding scheme, but as opposed to *baseline + 3*, it considers different hedges as different features.

To assess the performance of different classifiers, we compare them against the ZeroR classifier, which always predicts the majority class, and Random Guessing. We run our

experiments in a 10-fold cross-validation setting, using stratified sampling as implemented by WEKA. The same sets are used for training and testing, *i.e.*, we do not split the gold standard set into train and test sets in this study.

3.4.2 Results

The agreement between the four raters of the manual labelling of the group **hedges**, measured with Fleiss' kappa, is 0.59 for general and 0.49 for inline comments. Hence, regarding our **RQ1**, we believe those results reveal that humans can, indeed, reasonably identify confusion in code review comments (kappa between 0.41 and 0.60 is considered to be moderate (LANDIS; KOCH, 1977)). After solving the disagreement, we observe that confusion is expressed both in the general (18%) and in the inline (21%) comments, and while there are slightly more confusion comments among the inline ones, the association between the kind of comments and presence of confusion is not statistically significant ($p \simeq 0.33$ for Fisher's exact test.)

As for our **RQ2**, the classifiers' results are shown in Table 1. The table shows the performance of each classifier on each class: *confusion* comments and *no confusion* comments. The results show that tools can perform better than humans in the task of identifying confusion comments. The best precision on the *confusion* class is obtained by OneR for both general and inline comments albeit with different feature settings: for general comments, the best model is *baseline + 3* (0.875), and for inline ones—*baseline + hedges* (0.615). For recall, the best classifier is Multinomial Naive Bayes, for both general (0.944) and inline comments (0.998). It achieves the same performance for all models, both for general and inline comments. Regarding the balance between precision and recall, the best classifier is JRip for general (0.609) and Logistic for inline comments (0.497). In both cases, the model with best performance is *baseline + 3*.

3.4.3 Discussions

The interrater agreement suggests that identification of confusion can be reliably performed by human raters. This confirms the reliability of our annotation schema and the resulting golden sets. However, dealing with small, highly unbalanced dataset is something undesirable when training a classifier in a supervised machine learning setting (HE; GARCIA, 2009). Still, our preliminary results confirm that automatic detection of confusion in both general and inline comments is feasible.

More specifically, we observe that the more advanced models *baseline + 3* and *baseline + hedges* tend to outperform the baseline model, and none of the best classifiers reviewed above makes use of the baseline model. This means that addition of more specific features geared towards detection of confusion is indeed beneficial.

Highest recall is observed for both general and inline comments when using Multinomial Naive Bayes, regardless of the feature setting. This is consistent with previous evi-

Classifier	Class	General comments									Inline comments								
		Baseline			Baseline + 3			Baseline + *H*			Baseline			Baseline + 3			Baseline + *H*		
		P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F
Naive Bayes	C	.39	.47	.43	.44	.50	.47	.40	.50	.44	.38	.53	.44	.39	.54	.45	.38	.54	.45
	NC	.87	.84	.85	.88	.86	.87	.88	.83	.85	.86	.76	.81	.86	.77	.81	.86	.76	.81
Multinomial Naive Bayes	C	.20	.94	.34	.20	**.94**	.34	.21	.94	.34	.23	.98	.37	.23	**.98**	.37	.23	.98	.37
	NC	.94	.20	.33	.94	.20	.33	.94	.21	.35	.97	.12	.22	.97	.12	.22	.97	.13	.23
Logistic	C	.34	.61	.44	.36	.63	.46	.35	.59	.44	.43	.56	.48	**.43**	**.58**	.49	.43	.56	.48
	NC	.89	.74	.81	.90	.75	.82	.89	.75	.81	.87	.80	.83	.87	.79	.83	.87	.80	.83
Simple Logistic	C	.47	.12	.19	.68	.23	.35	.50	.12	.20	.45	.17	.25	.50	.10	.17	.51	.20	.29
	NC	.83	.96	.89	.85	.97	.90	.83	.97	.89	.81	.94	.87	.80	.97	.87	.81	.94	.87
SMO	C	.41	.09	.15	.50	.13	.21	.38	.09	.15	.51	.20	.29	.52	.22	.31	.50	.20	.28
	NC	.82	.96	.89	.83	.96	.89	.82	.96	.89	.81	.94	.87	.81	.94	.87	.81	.94	.87
IBk	C	.30	.09	.14	.26	.09	.14	.29	.13	.18	.33	.01	.02	.25	.01	.02	.10	.01	.02
	NC	.82	.95	.88	.82	.94	.87	.82	.92	.87	.78	.99	.87	.78	.99	.87	.78	.97	.86
KStar	C	.18	1	.30	.18	1	.30	.18	1	.30	.21	1	.35	.21	1	.35	.21	1	.35
	NC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
JRip	C	.56	.12	.20	**.69**	**.54**	.60	.64	.12	.20	.57	.14	.22	.43	.15	.22	.60	.16	.26
	NC	.83	.97	.90	.90	.94	.92	.83	.98	.90	.80	.97	.88	.80	.94	.87	.81	.97	.88
OneR	C	.57	.05	.10	**.87**	.19	.31	.57	.05	.10	.23	.98	.37	.46	.07	.12	**.61**	.09	.16
	NC	.82	.99	.90	.84	.99	.91	.82	.99	.90	.97	.12	.22	.79	.97	.87	.80	.98	.88
J48	C	.36	.16	.22	.59	.34	.43	.31	.16	.21	.30	.11	.17	.42	.25	.31	.31	.13	.18
	NC	.83	.93	.88	.86	.94	.90	.83	.92	.87	.79	.92	.85	.81	.91	.86	.79	.92	.85
Random Forest	C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	NC	.81	.99	.89	.81	.99	.89	.81	.99	.89	.78	1	.88	.78	1	.88	.78	1	.88
REPTree	C	.36	.05	.09	.63	.48	.55	.38	.06	.11	.42	.16	.23	.50	.21	.30	.39	.15	.22
	NC	.82	.97	.89	.89	.93	.91	.82	.97	.89	.80	.93	.86	.81	.94	.87	.80	.93	.86
Random Guessing	C	.18	.50	.26	.18	.50	.26	.18	.50	.26	.21	.50	.29	.21	.50	.29	.21	.50	.29
	NC	.81	.50	.62	.81	.50	.62	.81	.50	.62	.77	.50	.61	.78	.50	.61	.78	.50	.61
ZeroR (Majority)	C	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	NC	.81	1	.90	.81	1	.90	.81	1	.90	.78	1	.88	.78	1	.88	.78	1	.88

Table 1 – Classifiers' results: C—confusion class, NC—no confusion class; P–precision, R—recall, F—the F-measure, H—hedges.

dence in literature showing how Multinomial Naive Bayes outperforms other approaches when dealing with a small, unbalanced training set with few positive examples (FORMAN; COHEN, 2004), as in our case.

While differences between general and inline comments affect the precision and recall figures, the same classifiers seem to perform best. This is, however, not the case for the F-measure. TODO: Comment here!!! Overall, reasonably high precision and recall have been obtained, enabling future statistical studies of causes and effects of confusion in code reviews.

The best precision for the general comments is higher than the one for the inline comments, suggesting that identification of confusion in inline comments might be a more challenging task. This observation concurs with the previous observation that identification of confusion in inline comments turned out to be more difficult to the human raters as well. Alternatively, one might need different predictors, *e.g.*, the context of the code change, to detect confusion in inline comments. However, OneR produces a simplistic model, a set of rules operating on a single predictor, which may be poorly generalizable on new unseen data.

These results motived us to continue studying confusion detection and to work on a more in-depth study, described in the next Section 3.5.

3.4.4 Implications

As direct implications, the identification of confusion comments presented by our classifiers can enable statistical studies of causes and effects of confusion in code reviews. Moreover, researchers now have evidence that inline comments are prevalent and relevant for future code review studies: the number of confusion comments is similar for general and inline comments. Previous work analysing code review comments has focused predominantly on general comments (AHMED et al., 2017; MUNAIAH et al., 2017; PANGSAKULYANONT et al., 2014; MUKADAM; BIRD; RIGBY, 2013; HAMASAKI et al., 2013; THONGTANUNAM et al., 2014; YANG et al., 2016).

Furthermore, *tool builders* can benefit from our classifiers by expanding code review tools so as to be able to support confusion faced by developers and expressed by *hedges* expressions in the code review comments. *Researchers* can use our classifiers to study confusion expressed by *hedges* expressions in other systems, and hence, propose solutions to overcome the problems that confusion in code reviews bring forth, *e.g.*, by including more context information in the code reviews.

3.5 CONFUSION CLASSIFIER: A COMPREHENSIVE STUDY

In this section, we present the extension of our preliminary study. The goal of our preliminary study was to assess whether *confusion* could be manually and automatically identified in code reviews. Since the results provided positive evidence, we decided to conduct an in-depth analysis on confusion identification.

In this study, we consider the final gold standard set of the annotation process described in Figure 6 of Section 3.3 to build the classification models. The gold standard set contains 1,542 general comments, 350 labelled as *confusion* and 1,192 as *no confusion*, and 1,190 inline comments, 270 labelled as *confusion* and 920 as *no confusion*.

3.5.1 Methodology

The methodology used in this study to build our classification models is presented in Figure 7. Each step will be described in the following sections.

3.5.1.1 Pre-Process

Firstly, we pre-processed the comments in the dataset so as to make them more amenable to analyses (*pre-process* step). The pre-processing is performed by a Java script that we wrote. Several removal and replacements with meta-tokens (CALEFATO et al., 2018) were performed:

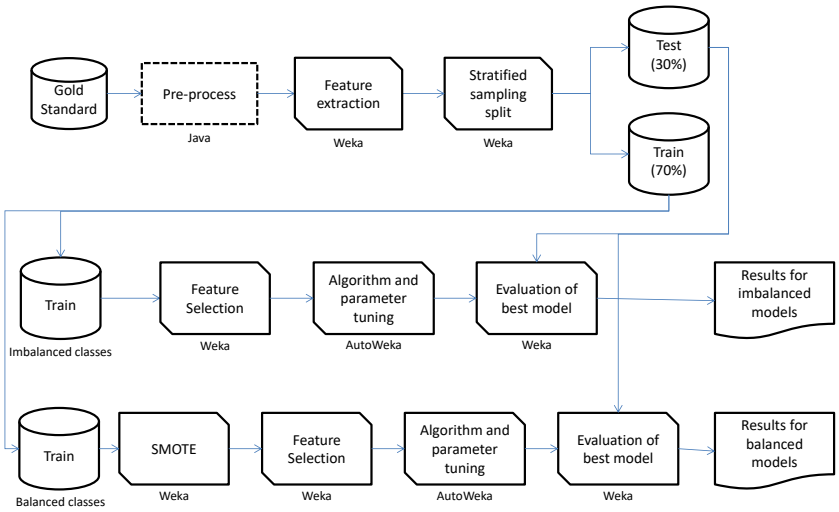

Figure 7 – Comprehensive study methodology.

1. Replacement of source code snippets, log, and stack trace messages by @CODE and @STACKTRACE meta-tokens. This step was done manually;
2. Removal of quoted texts. Quotes from other comments are discarded to avoid duplication;
3. Removal of line breaks;
4. Removal of corrupted characters;
5. Removal of XML meta-tokens;
6. Replacement of the Gerrit "Patch Set" pattern by @PATCHSET. Gerrit automatically adds the expression "Pact Set" in the general comments;
7. Replacement of URLs by the @URL meta-token;
8. Replacement of user names by the @USERNAME meta-token. We used the dataset provided by (vasilescu; capiluppi; serebrenik, 2014) to identify user names;
9. Replacement of commits IDs by the COMMIT meta-token;
10. Replacement of numbers by the NUMBER meta-token;
11. Expansion of the contractions, *e.g.*, "I'll" is expanded into "I will";
12. Escaping of single and double quotes. It was necessary in order to create the input file to Weka;
13. Removal of special chars, such as =, -, &, #, ∗, $, +, [, and];

14. Handling of the negation: we appended the "_NEG" suffix to every word appearing between a negation word and a clause-level punctuation mark (PANG; LEE; VAITHYANATHAN, 2002; DAS; CHEN, 2001);

15. Removal of the stop words (*e.g.*, "and", "or", "an", and "one"). We used the Rainbow[4] list of stop words;

The goal of expanding contractions is to have only one feature representing the same expression, and in the end, improve the classifiers' performance. For instance, "didn't" and "did not" are expressions with the same meaning, but the model would have considered one feature for each expression. We found several tools online providing such functionality, however, none of them was capable of handling more elaborate cases, such as the negative questions. The concern with negative questions and contractions is the need to change the position of the subject of the sentence. For example, the correct expansion of the question "Didn't you see the error?" is "Did you not see the error?". We did not find any tool able to handle such cases, mostly because it requires the construction of the syntactic structure of the question. Hence, we decided to develop our own tool using the StanfordNLP API. The replication package with the pre-processing steps, and the classification models are publicy available (EBERT, 2019).

The reason for handling negation is that we believe that features of negative sentences have a different effect in the classification (PANG; LEE; VAITHYANATHAN, 2002; DAS; CHEN, 2001). For instance, the features *really* and *good* from the examples "The code change is really good." and "The code change is not really_NEG good_NEG." will not have the same effect in the classification model, as the latter *really* and *good* have a negative connotation.

3.5.1.2 Feature Extraction

In the second step, we applied *feature extraction* to our dataset. We consider keyword based features in our study by counting n-grams appearing in the code review comment. Each n-gram corresponds to a feature with the number of occurrences as its value in our study. This is similar to traditional approaches used in text classification (JOACHIMS, 1998). We consider uni- and bi-grams. The StringToWordVector[5] filter from Weka was used to: i) extract the uni- and bi-grams, ii) apply stemming, and iii) employ the TF-IDF (RAJARAMAN; ULLMAN, 2011) transformation.

All expressions of our confusion coding scheme, listed in Appendix A, are considered features in our model. The categories of our scheme are also considered as individual features, *i.e.*, the sum of the occurrence of all expressions of each category.

[4] <http://www.cs.cmu.edu/~mccallum/bow/rainbow>
[5] http://weka.sourceforge.net/doc.dev/weka/filters/unsupervised/attribute/StringToWordVector.html

Feature	Description
Uni-grams	Total occurrences of uni-grams.
Bi-grams	Total occurrences of bi-grams.
Hedges	Total occurrences of hedges.
Probables	Total occurrences of probables.
Hypotheticals	Total occurrences of hypotheticals.
Questions	Total occurrences of questions.
I Statements	Total occurrences of I Statements.
Nonverbals	Total occurrences of nonverbals.
Meta	Total occurrences of meta.
SBARQ	Total occurrences of direct questions introduced by a wh-word or a wh-phrase.
SQ	Total occurrences of yes/no questions.
Questions	The sum of SBARQ and SQ.
Question marks	Total occurrences of question marks.

Table 2 – The list of features used in our models.

Additionally, we included four more features related to *questions*, in order to assess their influence in confusion detection. The first one is the presence of the question mark. The other are related to the kind of question. Since we were using the StanfordNLP API (MANNING et al., 2014) to build the category *questions* of our scheme, we decided to use it to identify and count the kind of question in the comments. The StanfordNLP API can identify two types of questions:

- **SBARQ:** a direct question introduced by a wh-word or a wh-phrase, *e.g.*, "what is that?";
- **SQ:** an inverted yes/no question, *e.g.*, "is it correct?".

Hence, we considered the total number of SBARQ, SQ, and their sum as features related to questions. In the end, we explored 13 different kinds of features, which are presented in Table 2. The feature *questions* represents the sum of the SBARQ and SQ.

3.5.1.3 Creation of Training and Testing Sets

Based on our gold standard sets, we created three models to be tested and evaluated in the step *stratified sampling split* of Figure 7. The model **GC** contains *general comments*, **IC** contains *inline comments*, and **All** contains the combination of both *general and inline comments*. We split all models (**GC**, **IC**, and **All**) into training (70%) and testing (30%) sets. We used Weka (FRANK; HALL; WITTEN, 2016) to split them using stratified

sampling, *i.e.*, keeping the proportion of the *confusion* and *no confusion* comments the same in the training and the testing sets.

3.5.1.4 Resampling

A dataset is *imbalanced* when the classified categories are not approximately equally represented. Oftentimes in real world datasets, this is the representation: the class of interest, in our case *confusion*, represents only a small percentage (CHAWLA et al., 2002) of all the cases. This is the case of our three models, the number of *confusion* comments is smaller than the number of *no confusion* comments. The problem with imbalanced classes within a dataset is degradation of the classifier's performance: the result might be biased towards the majority class, which means the misclassification of the minority class.

Hence, within each model, we experimented datasets with balanced classes in the *SMOTE* step of Figure 7. The resampling was only applied in the training set of each model (PARMANTO; MUNRO; DOYLE, 1996; ESTABROOKS; JO; JAPKOWICZ, 2004). We used Synthetic Minority Oversampling TEchnique (SMOTE) (CHAWLA et al., 2002) to over-sample the minority class, *i.e.*, *confusion*. This is a technique for automatically creating synthetic comments. The number of confusion comments from the **IC** and **GC** groups was over-sampled with 250%, and the group **All** with 200%. We defined those percentages aiming to reach a similar number of comments of each class. Figure 8 shows number of confusion and no confusion comments from each dataset within each model. Each of the three initial models **GC**, **IC**, and **All** derived one additional model with balanced classes. Finally, we have a total of six models used for training:

- **GC-I**: model of general comments with *imbalanced* classes.
- **GC-B**: model of general comments with *balanced* classes.
- **IC-I**: model of inline comments with *imbalanced* classes.
- **IC-B**: model of inline comments with *balanced* classes.
- **All-I**: model with the combination of **GC** and **IC** with *imbalanced* classes.
- **All-B**: model with the combination of **GC** and **IC** with *balanced* classes.

3.5.1.5 Feature Selection

Feature selection (cf. Figure 7) is an important step for text classification in machine learning because it reduces the training time, simplifies the model for users, and increases the generalisability of the model by reducing over-fitting (GUYON; ELISSEEFF, 2003; FORMAN, 2003). It is the process to automatically select a subset of the most relevant features to

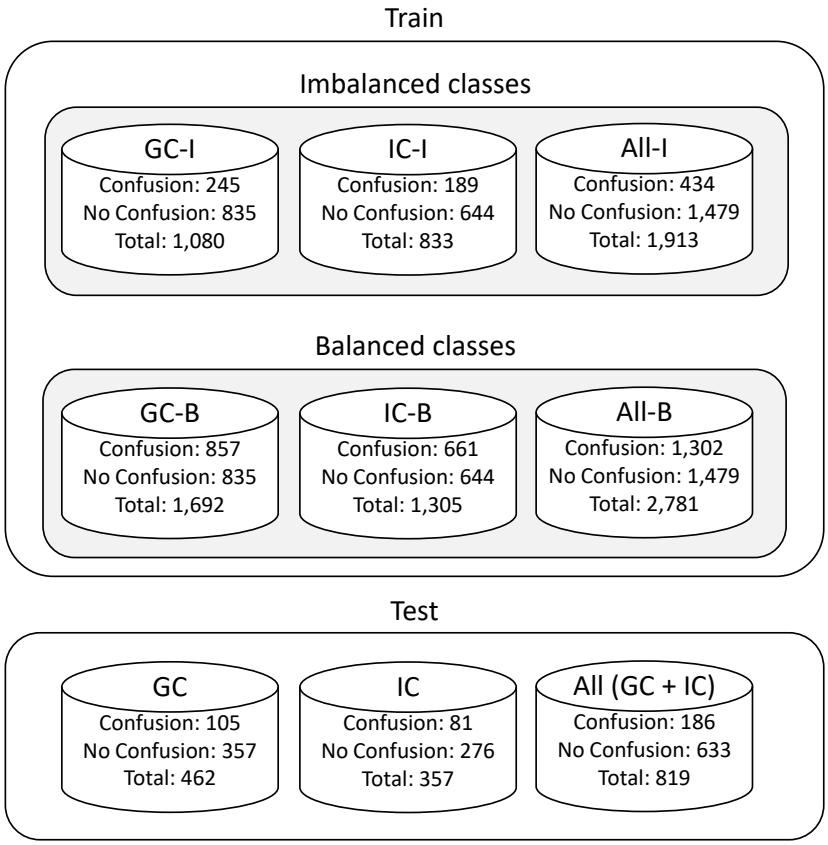

Figure 8 – The six models of our experiment.

predict the concern we are interested in, *i.e.*, it identifies and removes as much irrelevant and redundant features as possible.

Weka provides several techniques for feature selection. We followed the best recommendations (karegowda; m.a.jayaram; manjunath, 2010; menzies, 2016) to use wrapper-based methods (kohavi; john, 1997). Wrapper methods treat feature selection as a search problem by evaluating several models using procedures that add and/or remove predictors to find the optimal combination that maximizes the performance of the model. The wrapper-based methods use search algorithms that consider the predictors as inputs, and the performance of the model as the output to be optimised (karegowda; m.a.jayaram; manjunath, 2010). In general, these methods present a good performance. However, they can be expensive in terms of computational complexity and time, since each feature subset combination should be evaluated with the algorithm. We used the WrapperSubsetEval[6] method provided by Weka to run feature selection in all our six models.

[6] http://weka.sourceforge.net/doc.dev/weka/attributeSelection/WrapperSubsetEval.html

3.5.1.6 Algorithm and Parameter Tuning

In machine learning, the algorithms have parameters which are learned by analysing the data, and others which are not learned, *i.e.*, they need to be supplied by the user. Parameter tuning is the process of finding the best parameters for the specific machine learning algorithm. It is an important technique because it changes the heuristics on how the algorithms learn, and subsequently, the performance of the algorithms. For instance, it decides the criteria to split a node in a decision tree and the number of trees in a Random Forest. Several studies on defect prediction show that performance degradation can happen when a model is trained with suboptimal parameters because they are dependent on the dataset used (HALL et al., 2012; JIANG; CUKIC; MENZIES, 2008; MENDE; KOSCHKE, 2009). Nonetheless, models built without parameter tuning may have statistically indistinguishable performances (GHOTRA; MCINTOSH; HASSAN, 2015).

Hence, we used AUTO-WEKA (THORNTON et al., 2013) to perform automatic *algorithm and parameter tuning* (cf. Figure 7) in our models to optimise their performance when building prediction models. AUTO-WEKA executes algorithm selection and parameter optimisation over classification and regression algorithms on WEKA. It performs a statistically rigorous evaluation internally with 10-fold cross-validation, and provides the best classifier with the best parameter configuration as result.

3.5.1.7 Evaluation Criteria

The evaluation of classifiers' performance learned from imbalanced datasets should be conducted using specific metrics to take into account the class distribution and to correctly assess the effectiveness of learning algorithms (CALEFATO; LANUBILE; NOVIELLI, 2018a). Traditional scalar metrics such as *accuracy* (the proportion of correctly classified instances), and *error rate* (the proportion of incorrectly classified instances) cannot properly provide information about the performance of a classifier with imbalanced classes (PROVOST; FAWCETT; KOHAVI, 1998; RINGROSE; HAND, 1997). *Precision* is also sensitive with imbalanced datasets as it cannot identify how many positive examples are incorrectly classified, *i.e.*, the number of false negatives (HE; GARCIA, 2009). Moreover, Menzies *et al.* (MENZIES; GREENWALD; FRANK, 2007; MENZIES et al., 2007) showed that precision has instability problems, *i.e.*, it can have large standard deviations when dealing with skewed class distributions, which makes it hard to compare classifier performance. Even though *recall* is not sensitive to imbalanced classes, any assessment based only on it is inadequate because it does not provide any information about how many examples are incorrectly classified as positives. The harmonic mean of precision and recall, *i.e.*, *F-measure*, has been also shown to be problematic with imbalanced datasets (RAHMAN; POSNETT; DEVANBU, 2012).

Differently from the scalar metrics, which impose a one-dimensional ordering, two-

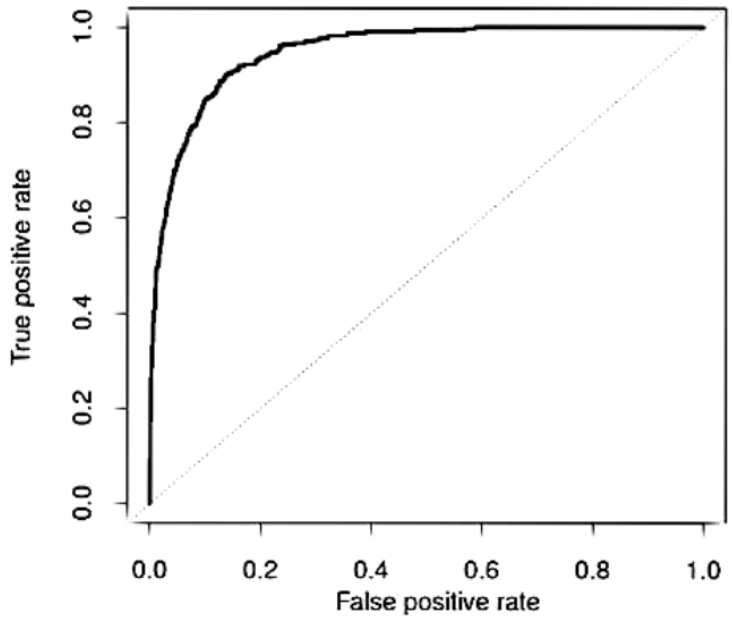

Figure 9 – The ROC plot for graphical assessment of performance.

dimensional plots conserve all the information related to the performance of a classifier, and hence, support a visual analysis and comparison of the classification results (DRUMMOND; HOLTE, 2006). The *Receiver Operating Characteristic* (ROC) plot (PROVOST; FAWCETT, 1997) is a two-dimensional graphic with the false positive rate as the x-axis, and the true positive rate as the y-axis. ROC is not sensitive to class distribution because it is only based upon *true positive rate* (*i.e.*, true positive divided by the total positive instances) and *false positive rate* (*i.e.*, falso negative divided by the total negative instances), which is a constant ratio for both balanced and imbalanced datasets. Thus, ROC plots are appropriate for comparison of classifiers over imbalanced datasets (LESSMANN et al., 2008). Figure 9 shows an example of a ROC plot. The ROC plot provides the visualisation of the performance of the classifier as the trade-off between the accurate classification of the positive instances and the misclassification of the negative instances. The diagonal line connecting the points (0, 0) and (1, 1) represents the performance of a random classifier. Hence, the closer the curve is to the point (0, 1), *i.e.*, the upper-left-hand corner, the better the classifier performance is.

The scalar metric *Area Under the ROC Curve* (AUC) represents the ROC performance of the a classifier. The larger the AUC value, the higher the classification potential of a classifier. The AUC value ranges from 0 and 1. The ROC plot also shows the random guessing as the diagonal line between the points (0, 0) and (1, 1), which has the AUC of 0.5. An AUC value of 0.5 means no discrimination of the classifier, *i.e.*, it acts as a coin flip. According to Hosmer and Lemeshow (HOSMER; LEMESHOW, 2000), the relevance of the AUC value can be classified as follows:

- **0.7 $\leq$ AUC $<$ 0.8:** means a considerable performance;
- **0.8 $\leq$ AUC $<$ 0.9:** means an excellent performance;
- **AUC $\geq$ 0.9:** means a outstanding performance.

Thus, we use the AUC as scalar metric to represent the ROC performance of confusion prediction. The ROC plots are drawn to perform a graphical comparison of the performance of the classifiers.

3.5.2 Results

The manual annotation of the groups **other** and **questions** resulted in agreement values fewer than the group **hedges** (cf. Figure 6). We present again the results of the agreement, measured with Fleiss' kappa, among the four raters for all three groups in Table 3. From all groups, **hedges** presented the best agreement (0.59 for general and 0.49 for inline comments) and **questions** had the lowest rate (0.32 for the general and 0.41 for the inline comments). The group **questions** of general comments is the only with a *fair* agreement (between 0.21 and 0.40), while all the other groups present a *moderate* agreement (between 0.41 and 0.60) (LANDIS; KOCH, 1977).

	hedges	**other**	**questions**
GC	0.59	0.48	0.32
IC	0.49	0.43	0.41

Table 3 – Fleiss' kappa values of the manual annotation process.

The results of the feature selection of our models are presented in Tables 4, 5, and 6. The features of our confusion coding scheme are specified with the respective category name in the tables. Furthermore, the features representing each category are specified with the word *category*.

We can observe that features related to *questions* are good predictors of confusion, being present in all models, *e.g.*, question marks, SQARQ, SQ, and *questions*. The categories *hedges*, *probables*, and *I statements* are also relevant predictors of confusion. The features from the *nonverbals*, *hypotheticals*, and *meta* categories were not selected as good predictors of confusion in any of our models.

Table 7 presents the results of AUTO-WEKA: the best classifier and its performance (the AUC value) on the training set of each model. Each model was trained and evaluated with its respective classifier identified by AUTO-WEKA. All six of our trained models are publicy available (EBERT, 2019). Table 7 also shows the performance of each classifier on the test set, and their ROC plots are shown in Figure 10. To assess the performance of our models, we compare them against the ZeroR classifier, which always predicts the majority class. The performance of the ZeroR classifier is similar to random guessing: it is

GC-Imbalanced	**GC-Balanced**
seems (hedges)	might (hedges)
question mark (category)	pretty (hedges)
cl	typically (hedges)
number @code	I am not sure (I statements)
	hedges (category)
	nonverbals (category)
	question marks (category)
	@code
	@stacktrace
	@username
	boot
	build
	end
	find_neg
	number number
	number verifiednumber
	review_neg
	space
	test_neg
	verifiednumber

Table 4 – The list of features selected of the models **GC-I** and **GC-B**.

represented with the diagonal line between the points (0, 0) and (1, 1) in the ROC plot. The best AUC values are highlighted in bold.

The best performances on the training set are achieved by the balanced models, with **IC-B** presenting the highest AUC value of 0.957. The evaluation on the testing set showed the best performance for the models related to *inline comments* **IC-I** and **IC-B**, with AUC of 0.706 and 0.760, respectively. The imbalanced model related to *general comments* **GC-I** performed worst both on training and testing sets, with AUC values of 0.597 and 0.599, respectively.

Previous work has argued against the use of resampling in datasets (TURHAN, 2012; LÓPEZ et al., 2013). Thus, we decided to experiment our imbalanced models with other scenarios as a way to seek better performance. Despite the removal of stop words being popular in machine learning, we followed previous research (SAIF et al., 2014; CALEFATO et al., 2018) by trying our imbalanced models without removing those words. The results are presented in Table 8 and the ROC plots in Figure 11. We can observe an improvement of 25% on the model **GC-I**, and a degradation of 11% on the **IC-I** model. The model **All-I** with stop words included showed a minor reduction of 1.4% in the performance.

IC-Imbalanced	**IC-Balanced**
really (hedges)	rather (hedges)
about (hedges)	about (hedges)
probably (probables)	I am not sure (I statements)
questions (category)	I do not understand (I statements)
SBARQ (category)	probably (probables)
question mark (category)	possible (probables)
@code_neg	hedges (category)
case	SBARQ (category)
check	question mark (category)
commit	@code
default_neg	call
	SQ (category)
	time

Table 5 – The list of features selected of the models **IC-I** and **IC-B**.

All-Imbalanced	**All-Balanced**
could (hedges)	might (hedges)
may (hedges)	seems (hedges)
about (hedges)	seems like (hedges)
opinion (hedges)	I am not sure (I statements)
would (hedges)	I think (I statements)
I am not sure (I statements)	I do not know (I statements)
I think (I statements)	meta_weird
I do not know (I statements)	hedges (category)
probables (category)	SBARQ (category)
questions (category)	@username
SQ (category)	adding
android	build_neg
worth_neg	fine
	format_neg
	function
	issue
	understand_neg
	wrong

Table 6 – The list of features selected of the models **All-I** and **All-B**.

3.5.2.1 Error Analysis

We decided to manually examine the comments misclassified by our models to get a deeper understanding of the reason for the classification error. The goal of this analysis was to identify the reasons why the classifiers produced a wrong prediction, and hence, we aimed at improving our models based on these results. As an initial step, we verified the misclassification of the imbalanced models (**GC-I**, **IC-I**, and **All-I**). The error

	Classifier	**AUC** (train set)	**AUC** (test set)
Baseline	ZeroR	0.500	0.500
GC-I	J48	0.597	0.599
GC-B	Vote	**0.916**	0.666
IC-I	PART	0.733	**0.706**
IC-B	RandomCommittee	**0.957**	**0.760**
All-I	J48	0.691	0.676
All-B	LWL	0.896	0.682

Table 7 – The results of the training and evaluation of the classifiers on each model.

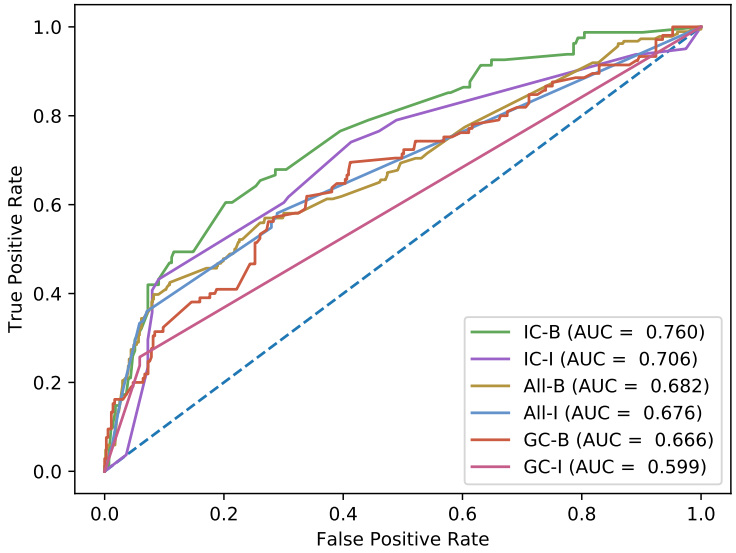

Figure 10 – The ROC plots of the performance of the six models.

	AUC (without stop words)	**AUC** (with stop words)	*Improvement*
GC-I	0.599	**0.748**	*+25%*
IC-I	**0.706**	0.626	*-11%*
All-I	0.676	0.666	*-1.4%*

Table 8 – The results of the improvement of the imbalanced models without the removal of stop words.

analysis of the balanced models is left for future work. The total number of misclassified comments by model is presented in Table 9. Next, we discuss notable error classes we derived from a manual analysis of misclassified comments. The total number of misclassified comments was split among three researchers, who manually annotated the comments with hypothesized causes of errors.

Questions not grammatically correct. The question posed by the developer does not have the correct grammar structure of a question, and hence, the model cannot take

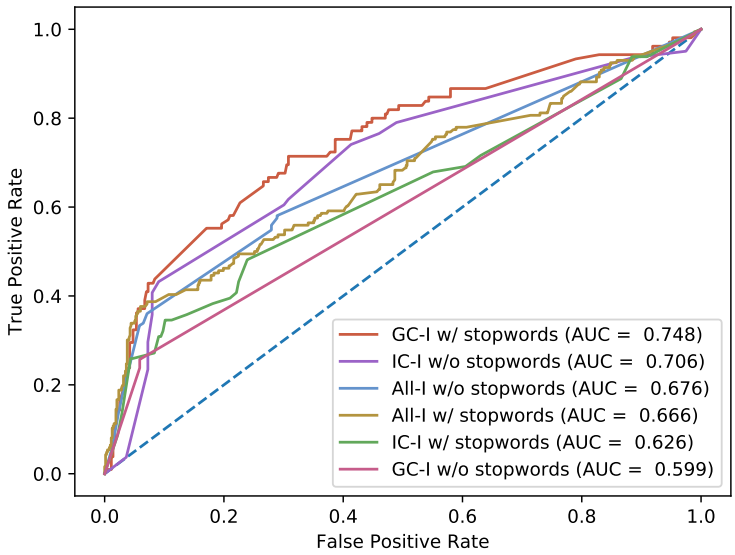

Figure 11 – The ROC plots of the performance of imbalanced models with and without removing stop words.

	Misclassified (% of the test set)	**Test set size**
GC-I	101 (22%)	462
GC-B	97 (20%)	462
IC-I	74 (20%)	357
IC-B	70 (20%)	357
All-I	161 (20%)	819
All-B	162 (20%)	819
Total	665 (20%)	3,276

Table 9 – Distribution of texts misclassified by our models.

the question into account for the classification. For example, "*'which' what?*" was not identified as a question. We believe the reason is intrinsically related to the *informality* used by developers when writing comments in code review tools (ZHOU; ZHANG, 2005).

Questions not detected by StanfordNLP API. A second problem related to questions is the StanfordNLP API not detecting them in the comment, *e.g.*, "*So you want to capture *verbose* logging, but it has to be in a *user/release* build?*" was not identified as a question. Thus, such comment in our model was not indicated as containing question, which in turn could have influenced the confusion classification. As future work, we consider experimenting different Natural Language Processing (NLP) libraries (OMRAN; TREUDE, 2017) in order to identify questions.

Communicative intention of questions. Another problem still related to questions is their communicative intention, *i.e.*, they have different meanings other than request for

information, and consequently, they do not express confusion. For example, in the question "*How about keeping the comment 'once enqueued, the pending next is always non-null'?*", the developer uses a question to make a kind *suggestion*. The use of *politeness* is also another reason for misclassification of questions.

General errors. These are errors related to the pre-processing steps. Some comments have an extremely short structure after all the pre-processing. For instance, the comment "*Patch Set 1: OK, but why?*" was shortened into "*PATCHSET NUMBER,?*". We believe those cases provide too little context for the models to identify confusion. We also observed some words, such as "we" and "can", being considered as user names, and being replaced with the meta-token @USERNAME. We think this can lead the models to misjudge the comments as well. Another problem we identified is in our confusion coding scheme: the expression *not sure* is not included, and it has been oftentimes present in confusion comments misclassified by our models. This is a case we should have handled in the pre-preprocessing steps, as the expression *unsure* is in our scheme, and both have the same meaning.

3.5.3 Discussions

The main contributions of this study are the confusion coding scheme and the automatic approach for confusion identification implemented by our models. All our six models outperformed the baseline, with the models related to inline comments having a considerable performance (HOSMER; LEMESHOW, 2000): the balanced model **IC-B** had the AUC of 0.760, and the imbalanced model **IC-I**, 0.706. The model of the general comments presented the worst performance from our models when the stop words are excluded, however, when we experimented without removing them, the imbalanced model **GC-I** improved to a considerable performance of 0.748. These results differ from the ones provided by our preliminary study, where the model of the general comments performed better than the inline comments.[[[[**From Flavia: why? -Felipe**]]]]

Despite of the approach to balance datasets being very popular, it has also some drawbacks (NOVIELLI; GIRARDI; LANUBILE, 2018). For example, it can change the underlying statistical problem (TURHAN, 2012) and it might also not bring classification-performance gain (LÓPEZ et al., 2013). In particular, Turhan (TURHAN, 2012) advocates against resampling because it optimises the training but alters the actual statistical problem. We believe this is the reason of the drop in the performance we observed on balanced models. As an initial analysis, we also reported the results of the imbalanced models without the removal of the stop words. It showed an improvement of 25% for the general model, but a degradation of 11% for the inline model. The model with the combination of general and inline comments showed a minor degradation of 1.4% in the performance.[[[[**From Flavia: why? -Felipe**]]]]

The importance of questions for confusion identification is expressed by the fact that

all of our models consider the features related to questions (*i.e.*, *questions*, SBARQ, SQ, or question marks) as the selected features. However, the group **questions** presented the lowest agreement between the four raters during the manual annotation. Additionally, the error analysis of the misclassified comments showed that there are several cases where the misclassification is related to questions, *i.e.*, several *no confusion* comments containing questions are incorrectly classified as *confusion*. Thus, we believe those are important factors that might negatively influence the performance of our classification models.

3.5.4 Implications

As direct implications of this in-depth study, the presented models enable statistical studies of causes and effects of confusion in code reviews. Moreover, now there is more evidence that *inline comments* are prevalent and relevant for future code review studies. Future research should consider including *inline comments* in their code review studies.

We believe that **tool builders** can benefit by expanding code review tools in order to automatically detect and support confusion. Bots that can identify confusion in developers' comments could be integrated into code review tools to provide support for those developers. We envision code review tools able to, for example, provide the documentation of the source for the reviewers when the bot identifies confusion on reviewers' comment, or provide the code context for the reviewers when confusion is due to other parts of code not in the *diff* file.

Furthermore, **researchers** can use our replication package to check for confusion in other software systems. It can enable future research on the understanding and comparison of how confusion is expressed among different systems. They can also use our approach as a basis to identify confusion in a number of different contexts, with different implications, *e.g.*, bug reports and the associated discussions (commit messages, email discussions, and design and requirements documentation).

3.6 THREATS TO VALIDITY

In this section, we discuss the threats related to the preliminary and comprehensive studies following the guidelines of Runeson *et al.* (RUNESON et al., 2012). We identify threats to construct, internal, and external validity:

Construct validity. Threats to construct validity are related to how properly a measurement reflects the concept being studied. Identifying *confusion* is not an easy task, and it has been operationalized using keywords and expressions from a confusion coding scheme. We built our confusion code scheme base on an existing scheme for uncertainty (JORDAN et al., 2012). Additionally, we considered several features from different sources (JORDAN et al., 2012; HOLMES, 1982; JORDAN; JR., 2014; JORDAN et al., 2014; LAKOFF, 1975; VARTTALA, 2001).

Internal validity. Threats to internal validity pertain to inferring conclusions from our study. None of the raters has been involved in ANDROID development, so they might have misinterpreted certain comments as *confusion* or *no confusion*. However, all raters are computer scientists and two of the four have a substantial experience with labeling textual information. Furthermore, all disagreements were solved with a majority vote and online meeting with all raters when required.

External validity. Threats to external validity are related to the generalizability of the study results. Our study targeted only ANDROID. This means that other projects might have different results. Replications are needed with larger datasets, using machine learning techniques specifically designed to deal with skewness in class distribution, to further assess the generality of our models. Furthermore, in our preliminary study we did not apply resampling on the training to counteract majority class bias that inherently affects our data. Learning from imbalanced data poses new emerging challenges that need to be addressed to build robust models of knowledge from raw data (HE; GARCIA, 2009). However, this threat was addressed in our comprehensive study with the application of oversampling (CHAWLA et al., 2002) of the minority class, *i.e.*, *confusion*.

3.7 RELATED WORK

Although automatic confusion identification has been advocated by several studies (YANG et al., 2015; JEAN et al., 2016), to the best of our knowledge no other study focused on the context of code reviews.

D'Mello *et al.* (D'MELLO et al., 2008) used an intelligent tutoring system able to help students learning by holding a conversation in natural language to explore detection of the learner's affect state. Their results showed that dialogue features could significantly predict the affective states of boredom, confusion, flow, and frustration. The experiments they conducted showed that standard classifiers were moderately successful in discriminating the affective states.

Baker *et al.* (BAKER et al., 2012) presented models able to automatically identify affect from students using a widely used learning application. They used the log files to detect students facing concentration, confusion, frustration, and boredom. Their models presented a better performance than a random classifier on identifying students' affective states.

Yang *et al.* (YANG et al., 2015) used textual content of comments from forums of massive open online courses and its clickstream data to automatically identify posts that express confusion. Their model to identify confusion comprises questions, users' click patterns, and users' linguistic features based on LIWC[7] words. They tried to identify the reasons why users are confused by looking at the recent click behavior. They found that the more

[7] https://liwc.wpengine.com

confusion students express or are exposed to, the lower the probability of their retention in the course. However, as their model is not publicly available we could not compare it to ours.

Another model to automatically detect expressions of uncertainty is presented by Jean *et al.* (JEAN et al., 2016). Their model uses multiple lexical and syntactic features to determine sentences through vector-based representations. However, their definition of confusion is quite different from ours. For instance, sentences such as "Could anyone submit this?" and "I rebased could you review again please." are examples of comments that the authors considered as exhibiting *confusion*. In our model, they are examples of *no confusion* comments. We could test their model in our dataset, but as just explained, the results were totally different. Hence, we decided not to include them in our discussions.

In the context of code reviews, there are some classification studies related to sentiment analysis rather than confusion. Ahmed *et al.* (AHMED et al., 2017) built a dataset of 2,000 code review manually labelled comments. This dataset was used to build a training dataset and to evaluate seven popular sentiment analysis tools. The poor performance of such tools motivated them to create their own sentiment analysis tool: SentiCR, especially designed for code review comments. They found Gradient Boosting Tree algorithm had the best mean accuracy (83%), the highest mean precision (67.8%), and the highest mean recall (58.4%) in identifying negative review comments.

Novielli *et al.* (NOVIELLI; GIRARDI; LANUBILE, 2018) described a benchmark study to evaluate the performance of three sentiment analysis tools, including SentiCR (AHMED et al., 2017), particularly customised for Software Engineering. They used four different datasets with sentiment labels on the benchmark, including the dataset with code review comments provided by Ahmed *et al.* (AHMED et al., 2017). They show that customisations related specifically to Software Engineering can boost accuracy for off-the-shelf tools, *i.e.*, not designed for Software Engineering. Specifically, supervised approaches provided the best performance for the classifiers.

3.8 SUMMARY

In this chapter, we presented two studies conducted to tackle the problems of *lack of knowledge* about confusion in code reviews and *lack of tools* for confusion detection. We provided our own definition of confusion and a confusion coding scheme. We constructed two gold standard sets: one with 1,542 general comments and other with 1,190 inline comments. These sets are publicy available (EBERT, 2019), as well as the replication package and the classification models for future research.

Furthermore, we built several models for confusion identification in developers' comments in code reviews. Our results show that the three models, **IC-I**, **IC-B**, and **GC-I** (with stop words included), reached a considerable performance on the identification of

confusion. We complemented the assessment of performance with the results of a qualitative error analysis.

4 CONFUSION IN CONTEXT: REASONS, IMPACTS, AND COPING STRATEGIES

This chapter presents the study we conducted to identify the reasons for confusion, its impacts, and the strategies developers use to cope with confusion in code reviews. The model we built based on the reasons, impacts, and coping strategies related to confusion in code reviews is simply called *confusion in context*. We start by providing a brief overview of the problem (Section 4.1). The concurrent triangulation strategy methodology is presented in Section 4.2. We present the results in Section 4.3. Then we discuss the implications (Section 4.4), and the threats to validity (Section 4.5). The related work is presented in Section 4.6. Finally, we summarise this study in Section 4.7.

4.1 OVERVIEW

In order to be able to perform code reviews, developers should understand *what* they are reviewing, *i.e.*, the code change. However, several studies show that understanding the code change and its context during code reviews are the major challenges faced by reviewers (BACCHELLI; BIRD, 2013; COHEN; TELEKI; BROWN, 2006; TAO et al., 2012; SUTHERLAND; VENOLIA, 2009; LATOZA; VENOLIA; DELINE, 2006).[[[[**From Flavia: explain better what is this context? -Felipe**]]]] Such challenges can cause the code reviews to be delayed (COHEN; TELEKI; BROWN, 2006; BACCHELLI; BIRD, 2013; TAO et al., 2012; SUTHERLAND; VENOLIA, 2009; LATOZA; VENOLIA; DELINE, 2006). Thus, we believe that confusion in code reviews should be properly understood, including the reasons why it exists, its impacts, and how developers deal with it. The outcome of such understanding of confusion can help to reduce the costs of code review time, and to improve the code review and the overall development processes.

The goals of this study are threefold. Firstly, we aim to obtain empirically-driven actionable insights for both researchers and tool builders, on what the main causes of confusion in code reviews are. Thus, we formulate our first research question:

RQ1. *What are the reasons for confusion in code reviews?*

We have observed that the three most frequent reasons for confusion are *missing rationale*, *discussion of the solution: non-functional*, and *lack of familiarity with existing code*.

Secondly, while confusion can be expected to negatively affect code reviews, we would like to identify specific impacts of confusion. By monitoring these impacts, developers and managers can curb undesirable consequences. As such, we formulate our second research question:

RQ2. *What are the impacts of confusion in code reviews?*

Our results suggest that the merge decision is *delayed* when developers experience confusion, there is an increase in the number of messages exchanged during the discussion, and the *review quality decreases*. However, we also observed unexpected consequences of confusion, such as helping to find a *better solution*. This suggests that communicating uncertainty and doubts might be beneficial for collaborative code development, *i.e.*, by inducing critical reflection (EBERT et al., 2018) or triggering knowledge transfer (BACCHELLI; BIRD, 2013).

Finally, we believe that understanding the strategies adopted by the developers to deal with confusion can further inform the design of tools to support code reviewers in fulfilling their information needs associated to the experience of confusion. As such, we formulate our third research question:

RQ3. *How do developers cope with confusion during code reviews?*

The results suggest that developers try to deal with confusion by *requesting information*, *improving the familiarity with existing code*, and *discussing off-line* (outside the code review tool). We also found that confusion might simply induce developers to *blindly approve* the code change, regardless of its correctness.

The main contribution of this study is a **comprehensive model for *confusion in context* in code reviews** including reasons, impacts, and coping strategies. To address our research questions, we implemented a concurrent triangulation strategy (EASTERBROOK et al., 2008) by combining a survey ('what people think') with analysis of the code review comments ('what people do') from the dataset we built during the study presented in Chapter 3 (published at Ebert *et al.* (EBERT et al., 2017)). The data collected and manually annotated during the study are released to enable follow-up studies (EBERT, 2019). Based on the analysis of this model, we formulate a series of suggestions for tool builders and researchers.

The findings of our study complement recent research on comprehension in code reviews, *i.e.*, our study from Chapter 3 proposing a model to identify confusion in code reviews, and the one by Pascarella *et al.* (PASCARELLA et al., 2018), focusing on understanding the information needs of code reviewers.

4.2 METHODOLOGY

Next, we describe how we implemented the *concurrent triangulation strategy* (EASTERBROOK et al., 2008) used to address our research questions. Concurrent triangulation strategy is a mixed-methods approach which employs diverse methods concurrently with the goal to corroborate, confirm, or cross-validate findings, *i.e.*, to increase validity of the study. In particular, when it comes to human activities, Easterbrook *et al.* advocate triangulation since "often 'what people say' could be different from 'what people do' " (EASTERBROOK et al., 2008). Firstly, we conduct a survey to understand "what developers say" (Section 4.2.1). Then we analyse code review comments to understand "what

developers do" (Section 4.2.2). Finally, we compare and contrast the findings of the two analyses (Section 4.2.3).

4.2.1 Surveys

In literature, a theory is missing to describe what are the reasons for confusion in code reviews, the impact of confusion on the development process, and what coping strategies developers employ to deal with confusion. As such, to answer our **RQs** we opt for grounded theory building (GLASER; STRAUSS, 1967; STOL; RALPH; FITZGERALD, 2016). We implemented an iterative approach. During each iteration, we administer a survey with developers involved in code reviews. We ask developers that already answered the survey during one of the previous iterations to refrain from answering it again.

4.2.1.1 Survey design

The survey was designed according to the established best practices (GROVES et al., 2009; KITCHENHAM; PFLEEGER, 2008; SINGER; VINSON, 2002; STEELE; ARONSON, 1995): prior to asking questions, we explain the purpose of the survey and our research goals, disclose the sponsors of our research and ensure that the information provided will be treated in a confidential way. In addition, we inform the participants about the estimated time required to complete the survey, and obtain their informed consent. The invitation message includes a personalised salutation, a description of the criteria we used for participant selection, as well as the explanation that there would not be any follow up if the respondent do not reply. This last decision also implies that we did not send reminders.

The survey starts with the definition of confusion as provided in Section 3.2, followed by a question requiring the participants to confirm that they understood the definition. Next, we ask two series of questions: the questions were essentially the same, but were first asked from the perspective of the author of the code change, and then from the perspective of the reviewer of the change. The survey is presented in Table 10. Each series starts with the Likert-scale question about the frequency of experienced confusion: *never*, *rarely*, *sometimes*, *often*, and *always*. To ensure that the respondents interpret these terms consistently, we provide quantitative estimates: 0%, 25%, 50%, 75% and 100% of the time. For respondents who answered anything different from *never*, we pose four open-ended questions (to get as rich as possible data (FODDY, 1993)): i) what are the reasons for confusion, ii) whether they can provide an example of a practical situation where confusion occurred during a code review (**RQ1**), iii) what are the impacts of confusion (**RQ2**), and iv) how do they cope with confusion (**RQ3**). Finally, we ask the participants to provide information about their experience as developers and frequency of reviewing and authoring code changes. We ask these question at the end of the survey rather than at the beginning to reduce the *stereotype threat* (STEELE; ARONSON, 1995). Prior to deploying the survey, we discussed it with other software engineering researchers and clarified it when necessary.

Table 10 – Confusion in Code Reviews Survey. The questions marked "*" were only used in the first survey, "+" —only in the second and third surveys.

Electronic Consent
0. Please select your choice below. Selecting the "yes" option below indicates that: i) you have read and understood the above information, ii) you voluntarily agree to participate, and iii) you are at least 18 years old. If you do not wish to participate in the research study, please decline participation by selecting "No".
Definition of Confusion
The remainder of this survey is dedicated to "confusion". We do not make a distinction between lack of knowledge, confusion, or uncertainty. For simplicity reasons, we use the "confusion" to refer to all these terms.
1. By clicking "next" you declare that you understand the meaning of confusion on this survey.
Review-Then-Commit
2.+ Have you ever taken part in a "review-then-commit" type of code review (*i.e.*, the code is reviewed before it is integrated into the main repository), either in the role of author or reviewer?
When reviewing code changes
3. Developers might feel confused or think that they do not understand the code they review. How often did you feel this way when reviewing code changes?
4. What usually makes you confused when you are reviewing code changes? Please explain which factors led you to be confused.
5. Please describe a change you have been reviewing that has confused you.
6. How does the confusion you experience as a reviewer impact code review?
7. What do you usually do to overcome confusion in code reviews? Please explain the actions you take when you feel confused.
8.* When you do not understand a code change, do you usually express this in general comments or in inline comments? Please explain why in the "other" field.
When authoring code changes
9. Developers who authored code changes might feel confused or think that they do not understand something when their code is being reviewed. How often did you feel this way when your code has been reviewed?
10. What usually makes you confused during the code review when you are the author of the code changes? Please explain which factors led you to be confused.
11. Please describe a change you have been authoring that has confused you.
12. How does confusion you experience as the code change author impact the code review?
13. What do you usually do to overcome confusion in code reviews? Please explain the actions you take when you feel confused.
14.*When you do not understand a code change, do you usually express this in general comments or in inline comments? Please explain why in the "other" field.
Background
15. What is your experience as a developer?
16. What is your experience as a code reviewer?
17. How often do you submit code changes to be reviewed?
18. How often do you review code changes?
19.*Do you have the merge approval right (*i.e.*, the permission to give +2) in Gerrit at least for one software development project?
20.*Which option would best describe yourself? - I contribute to Android voluntarily. - I'm employed by a company other than Google and I contribute to Android as part of my job. - I'm employed by Google and I contribute to Android as part of my job. - Other.
Results
21. Would you like to be informed about the outcome of this study and potential publications? Please leave a contact email address.
22. Would you be willing to be interviewed afterwards?
23. Please add additional comments below.

4.2.1.2 Participants

The target population consists of developers who participated in code reviews either as a change author or as a reviewer. During the first iteration, we targeted Android developers who participated in code reviews on Gerrit. The dataset of code reviews developed during the study described in Chapter 3 provides 4,645 email addresses of Android

developers. We used this list to contact them by email and evaluate the response rate. In the subsequent iterations, the survey was announced on FACEBOOK and TWITTER. As the exact number of developers participating in code reviews reached by our posts on social media cannot be known, we do not report the response rate for the follow-up surveys.

4.2.1.3 Data analysis

To analyse the survey data, we use a card sorting approach (ZIMMERMANN, 2016). It is a manual technique to create mental models and derive taxonomies from text. We analyse the survey responses from the first iteration using *open card sorting* (ZIMMERMANN, 2016), *i.e.*, topics were not predefined, but emerged and evolved during the sorting process. After each subsequent survey iteration, we use the results of the previous iteration to perform *closed card sorting* (ZIMMERMANN, 2016), *i.e.*, we sort the answers of each survey iteration according to the topics emerging from the previous one. If the closed card sorting succeeds, this means that the saturation has been reached and sampling more data is not likely to lead to the emergence of new topics (FINFGELD-CONNETT, 2014; LENBERG et al., 2017). In such a case, the iterations stop. If, however, during the closed card sorting additional topics emerge, another iteration is required.

To facilitate analysis of the data, we use axial coding (KITCHENHAM; PFLEEGER, 2008) to find the connections among the topics and group them into dimensions. These dimensions emerge and evolve during the final phase of the sorting process, and they represent a higher level of abstraction of the topics.

As we have multiple iterations and multiple surveys answered by different groups of respondents, *a priori* it is not clear whether the respondents can be seen as representing the same population. Indeed, it could have been the case that, *e.g.*, respondents of the second survey happened to be less inclined to experience confusion than the respondents of the third survey and the reasons of their confusions are very different. This is why we first check similarity of the groups of respondents in terms of their experience as developers and code reviewers, frequency of submitting changes to be reviewed and reviewing changes, as well as frequency of experiencing confusion. If the groups of respondents are found to be similar, we can consider them as representing the same population and merge the responses. If the groups of respondents are found to be different, we treat the groups separately.

To perform the similarity check, we use two statistical methods: i) analysis of similarities (ANOSIM) (CLARKE, 1993), which provides a way to statistically test if there is a significant difference between two or more groups of sampling units, and ii) permutational multivariate analysis of variance using distance matrices (ADONIS) (ANDERSON, 2001; MCARDLE; ANDERSON, 2001).[1]

[1] Both ANOSIM and ADONIS are available as functions in the R package *vegan*.

4.2.2 Analysis of Code Review Comments

To triangulate the survey findings for the **RQs**, we perform an analysis of code review comments. Once again, we use the dataset of ANDROID code reviews from the study of Chapter 3. However, we only considered comments from the groups **hedges** and **other**. We decided not to include the comments from the **questions** group because they had the lowest agreement on confusion among the raters. Moreover, the intentions of questions have been shown to be one of the main reasons for misclassification of *no confusion* comments by our classification models (cf. Section 3.5.2.1). Those facts motivated us to conduct a specific study on that group (cf. Chapter 5). The code reviews of ANDROID are supported by GERRIT, which enables communication between developers during the process by using general and inline comments. The former are posted in the code review page itself, which presents the list of all general comments, while the inline comments are included directly in the source code file and usually reference a word, a line or a group of lines. The dataset comprises 307 code review comments manually labelled by the researchers as confusing: 156 are general and 151 are inline comments.

Similarly to the analysis of the survey data, we use card sorting to extract topics from the code review comments. We conduct an open card sorting of the general comments to account for the possibility of divergent results, *i.e.*, we did not want to use the results from the surveys because what developers do might differ from what they think they do, and the emergent topics might *a priori* be different from those obtained when analysing the survey data. To confirm the topics emergent from the general comments, we then perform a closed card sorting on the inline comments.

4.2.3 Triangulating the findings

The goal of concurrent triangulation is to corroborate the findings of the study, increasing its validity. Thus, following Easterbrook *et al.* (EASTERBROOK et al., 2008) work, we expect to identify some differences between 'what people say' (survey) and 'what people do' (code review comments). Hence, if the topics extracted from the surveys and code review comments disagree, we conduct a new card sorting round only on the cards associated with topics discovered on the basis of the survey, but not on the basis of the code review comments, or vice versa. In order not to be influenced by the results of the previous card sorting, we perform open card sorting and exclude the researchers who participated in the previous card sorting rounds.

Finally, in order to finalise the comprehensive model of *confusion in context* in code reviews, we perform the consistency check within the topics and deduction of more generic topics, as recommended by Zimmermann (ZIMMERMANN, 2016), as well as a consistency check across **RQs** (*i.e.*, reasons, impacts, and coping strategies) and emergent dimensions.

4.3 RESULTS

We discuss the application of the research method in practice (Section 4.3.1), and analyse similarity between the responses received at each one of the survey iterations (Section 4.3.2). Then, we present the demographics results from the survey (Section 4.3.3), and discuss reasons for confusion (**RQ1**, Section 4.3.4), its impact (**RQ2**, Section 4.3.5), and the strategies employed to cope with it (**RQ3**, Section 4.3.6).

4.3.1 Implementation of Approach

The implementation of the approach designed in Section 4.2 is shown in Figure 12. Individuals involved in the card sorting are graduate students in computer science or researchers.

Firstly, following the iterative approach we have performed three iterations since saturation has been reached. Among the 4,645 emails sent during the first iteration, 880 emails have bounced; hence, 17 valid responses correspond to the response rate 0.45%. Such response rate was unexpected: indeed, the common response rates in Software Engineering range between 15% and 20% (PALOMBA et al., 2015; VASILESCU; FILKOV; SEREBRENIK, 2015; VASILESCU et al., 2015), and sometimes much higher response rates are reported (PALOMBA et al., 2018). We conjecture that the low response rate might have been caused by presence of inactive members or one-time-contributors (LEE; CARVER; BOSU, 2017). For the second and the third survey rounds, the number of responses are 24 and 13 respectively; the response rate could not be computed.

The open card sorting of the first survey resulted in 52 topics related to the reasons (25), impacts (14) and coping strategies for confusion (13). The closed card sorting of the second survey resulted in three additional topics: two for impacts and one for the coping strategies. Finally, the closed card sorting of the third survey resulted in no new topics. The open card sorting on the general comments resulted in 16 topics related only to the reasons for confusion, *i.e.*, no topics related to the impacts and coping strategies appeared. Then, the closed card sorting on the inline comments resulted in no new topics.

During the triangulation, we verified that what developers said about the reasons for confusion (survey) has little agreement with what developers said in the code review comments. Only six topics were found both among the survey answers and code review comments, 19 topics appeared only in the survey and 10 topics—in the code review comments. Thus, as explained in Section 4.2.3, we decided to conduct another card sorting on the divergent 29 topics. This time, since it was an open card sorting, from the cards belonging to divergent topics we identified 42 topics.

As the last step, we finalised the comprehensive model and obtained a total of 57 topics related to reasons (30), impacts (14), and coping strategies (13). After finalising the topics, we observe that 70% (21/30) of them have cards both from the surveys and from the review comments. Moreover, the shared topics cover the lion's share of the cards:

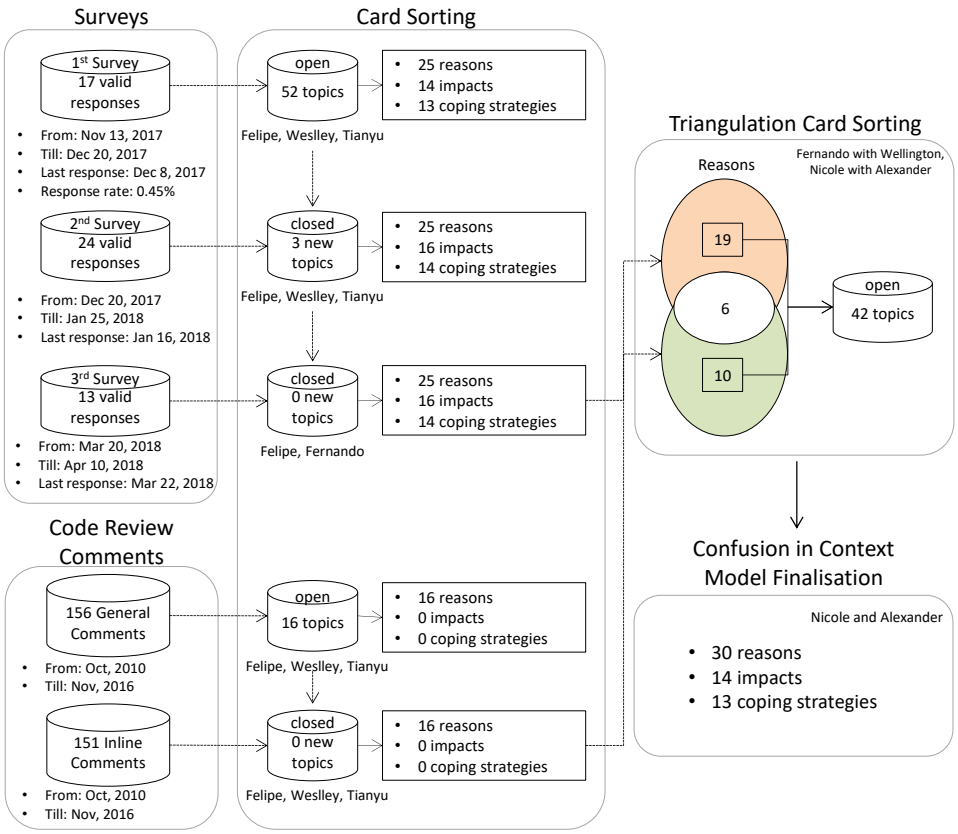

Figure 12 – Implementation of the approach: three survey rounds, general and inline comments, the triangulation, and finalisation rounds.

94.9% of the survey cards and 90.7% of the code review comments' cards.

As explained above, we identified the following dimensions using axial coding, common for the three **RQs**:

- **review process** (18 topics): the code review process, including issues that affect the review duration;
- **artifact** (15 topics): the system prior to change, code change itself and its documentation or the system after change;
- **developer** (15 topics): topics regarding the person implementing or reviewing the change;
- **link** (9 topics): the connection between developers and artifacts, *e.g.*, when a developer indicates that they do not understand the code.

Examples of topics of different dimensions can be found in Sections 4.3.4, 4.3.5 and 4.3.6.

Figure 13 – Experience of developers and reviewers.

4.3.2 Analysis of Similarity of the Surveys' Results

Firstly, we verified the similarity of the second and third surveys. Since both were published on FACEBOOK and TWITTER, we expect the values to be similar, *i.e.*, respondents to represent the same population. Using both ANOSIM ($R = -0.0171$ and p-value $= 0.542$) and ADONIS (p-value $= 0.975$), we could not observe statistically significant differences between the groups, *i.e.*, the answers can be grouped together.

Then, we checked the similarity between the answers to the first survey (ANDROID developers) and the answers to the second and the third surveys taken together. The results of the ANOSIM analysis, $R = 0.126$ and p-value $= 0.01$, showed that the difference between the groups is statistically significant. However, the low R means that the groups are not so different (values closer to 1 mean more of a difference between samples), *i.e.*, the overlap between the surveys is quite high. This observation is confirmed by the outcome of the ADONIS test: the p-value $= 0.191$ is above the commonly used threshold of statistical significance (0.05). Based on those results, we conclude that the respondents represent the same population of developers and report the results of all three surveys together.

4.3.3 Demographics of the survey respondents

The respondents are experienced *code reviewers*, 80% (38 of 47 respondents that answered questions about demographics) have more than two years of experience reviewing code changes. The experience of our population as *developers*, *i.e.*, authoring code changes, is even higher: 93% (44 respondents) have been developing for more than two years. The experience of code change authors and reviewers are presented in Figure 13. The number of years of experience as *developers* is higher than the number of years of experience as *reviewers*: this is expected because reviewing tasks are usually assigned only to more experienced individuals (WESEL et al., 2017).

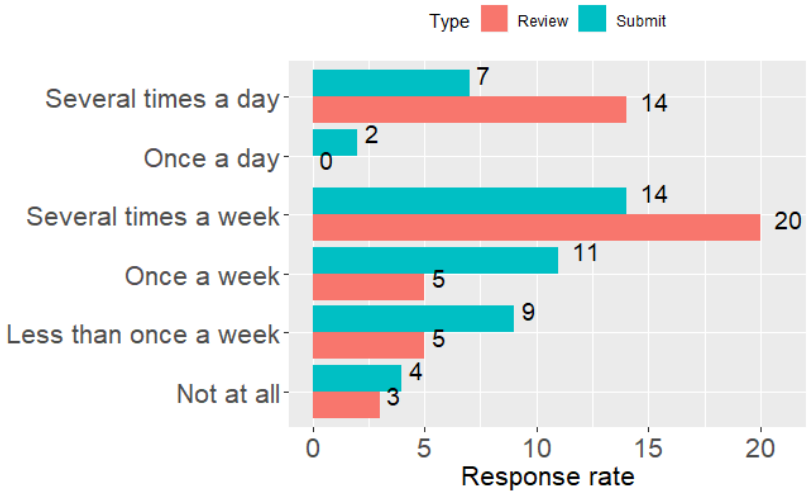

Figure 14 – Frequency of code review submissions and code reviews conducted.

Respondents are active in submitting changes for review, and even more active in reviewing changes: almost 49% (23 developers) submit code reviews several times a week, while for reviewing this percentages reaches 72% (34) (cf. Figure 14).

The frequencies with which code change authors and code reviewers experience confusion are summarised in Figure 15. On the one hand, when reviewing code changes, about 41% (20) of the respondents feel confusion at least half of the time, and only 10% (5) do not feel confusion. On the other hand, when authoring code changes, only 12% (6) of the respondents feel confusion at least half of the time, and 35% (17) of the respondents do not feel confusion. Comparing the figures, we conclude that confusion when reviewing is very common, and that developers are more often confused when reviewing changes submitted by others as opposed to when authoring the change themselves.

We also applied the χ^2 test to check whether experience influences frequency of confusion being experienced. The test was not able to detect differences between more and less experienced developers in terms of frequency of confusion being experienced as a developer, nor between more and less experienced reviewers in terms of frequency of confusion being experienced as a reviewer ($p \simeq 0.26$ and 0.09, respectively).

4.3.4 RQ1. What are the reasons for confusion in code reviews?

We found 30 reasons for confusion in code review (see Table 11). They are spread over all the dimensions, with the artifact and review process being the most prevalent.

There are seven reasons for confusion related to the code review process. The most common is *organisation of work*, which comprises reasons such as unclear commit message (*e.g.*, "*when the description of the pull request is not clear*" — R50), the status of the change (*e.g.*, "*I'm unsure about the status of your parallel move changes. Is this one ready to be reviewed? [...]*")[2], or the change addressing multiple issues (*e.g.*, "*change does more*

[2] https://android-review.googlesource.com/c/132581

Figure 15 – Frequency of confusion for developers and reviewers.

than one things" — R31). The second and third reasons most cited are, respectively, confusion about the tools, *e.g.*, "*I don't know why the rebases are causing new CLs*"[3], and the need of the code change, *e.g.*, "*If I understand correctly, this change might not be relevant any more*".[4]

The artifact dimension is the largest group with 11 topics related to the reasons for confusion. The most popular is the absence of the change rationale, *e.g.*, "*I do not fully understand why the code is being modified*" (R20). Discussion of the solution related to non-functional aspects of the artifact is the second largest topic and it comprises reasons such as poor code readability (*e.g.*, "*Poorly implemented code*" — R43), and performance (*e.g.*, "*is this true? i can't tell any difference in transfer speed with or without this patch. i still get roughly these numbers from 'adb sync' a -B build of bionic: [...]*").[5] The third most frequent reason indicates that developers experience confusion when they are *unsure about the system behavior*, *e.g.*, "*what is the difference between this path (false == unresolved) and the unresolved path below. [...]*".[6]

Six reasons for confusion are related to the developer dimension. *Disagreement* among the developers is the prevalent topic, *e.g.*, "*[...] If actual change has a big difference from my expectation, I am confused.*" (R11). The second most cited reason is the misunderstanding of the message's intention, *e.g.*, "*Sometimes I don't understand general meaning (need to read several times to understand what person means)*" (R13).

Six reasons are related to the link between the developer and the artifact. The most popular one is the *lack of familiarity with existing code*, *e.g.*, "*Lack of knowledge about the code that's being modified.*" (R37) followed by the *lack of programming skills*, *e.g.*, "*sometimes I'm confused because missing some programming*" (R13), and the *lack of un-*

[3] https://android-review.googlesource.com/c/71976
[4] https://android-review.googlesource.com/c/33140
[5] https://android-review.googlesource.com/c/91510
[6] https://android-review.googlesource.com/c/83350

Table 11 – The reasons, impacts and coping strategies developers use to deal with confusion; in the parenthesis are the numbers of cards.

	Reasons *30 topics (507)*	**Impacts** *14 topics (98)*	**Coping strategies** *13 topics (116)*
Review process *18 topics (120)*	Organisation of work (17) Issue tracker, version control (7) Unnecessary change (6) Not enough time (3) Dependency between changes (3) Code ownership (2) Community norms (2)	Delaying (31) Decreased review quality (11) Additional discussions (11) Blind approval (8) Review rejection (4) Increased development effort(4) Assignment to other reviewers (2)	Improved organisation of work (5) Delaying (2) Assignment to other reviewers (1) Blind approval (1)
Artifact *15 topics (300)*	Missing rationale (66) Discussion of the solution: non-func. (49) Unsure about system behavior (37) Lack of documentation (29) Discussion of the solution: strategy (29) Long, complex change (25) Lack of context (19) Discussion of the solution: correctness (14) Impact of change (11) Irreproducible bug (6) Lack of tests (5)	Better solution (1) Incorrect solution (1)	Small, clear changes (4) Improved documentation (4)
Developer *15 topics (124)*	Disagreement (18) Communicative intention (9) Language issues (3) Propagation of confusion (3) Fatigue (1) Noisy work environment (1)	Decreased confidence (10) Abandonment (6) Frustration (5) Propagation of confusion (2)	Information requests (36) Off-line discussions (12) Providing/accepting suggestions (10) Disagreement resolution (6)
Link *9 topics (177)*	Lack of familiarity with the existing code (47) Lack of programming skills (40) Lack of understanding of the problem (21) Lack of understanding of the change (17) Lack of familiarity with the technology (14) Lack of knowledge about the process (3)		Improved familiarity with the existing code (28) Testing the change (5) Improved familiarity with the technology (2)

derstanding of the problem, e.g., "*I'm embarrassed to admit it, but I still don't understand this bug.*".[7]

RQ1 Summary: We found a total of 30 reasons for confusion. The most prevalent are *missing rationale*, *discussion of the solution: non-functional*, and *lack of familiarity with existing code*. We observe that tools (code review, issue tracker, and version control) and communication issues, such as disagreement or ambiguity in communicative intentions, may also cause confusion during code reviews.

4.3.5 RQ2. What are the impacts of confusion in code reviews?

The total number of topics related to the impacts of confusion is 14 (see Table 11). They are related to the dimensions of the review process, artifact, and developer. There was no topic related to the link between the developer and the artifact.

[7] https://android-review.googlesource.com/c/170280

We identified seven impacts of confusion related to the code review process. *Delaying* the merge decision is the most popular impact, *e.g.*, "*The review takes longer than it should*" (R46). The second and third most cited impacts are that confusion makes the code review quality decrease, *e.g.*, "*Well I can't give a high quality code review if I don't understand what I am looking at*" (R5), and an increase in the number of messages exchanged during the discussion, *e.g.*, "*Code reviews take longer as there's additional back and forth*" (R1). One interesting impact of confusion is the blind approval of the code change by the developer, even without understanding it, *e.g.*, "*Blindly approve the change and hope your coworker knows what they're doing (it is clearly the worst; that's how serious bugs end up in production)*" (R16). Confusion may also lead developers to just reject a code change, *e.g.*, "*I'm definitely much more likely to reject a 'confusing' code review. Good code, in my experience, is usually not confusing*" (R36).

There are only two impacts of confusion related to the artifact itself. Firstly, the developer may find a better solution because of the confusion, *e.g.*, "*It has not only bad impact but also good impact. Sometimes I can encounter a better solution than my thought*" (R11). Secondly, the code change might be approved with bugs, as the reviewer is not able to review it properly due confusion, *e.g.*, "*Sometimes repeated code is committed or even a wrong functionality*" (R24). The *incorrect solution* impact is related to *decrease review quality*, however, the perspective is of the code change containing a bug in production rather than of the reviewing process.

Finally, there are four impacts of confusion related to the developer. The most quoted impact is the decrease of self confidence, either by the author, *e.g.*, "*I can't be confident my change is correct*" (R38), or by the reviewer, *e.g.*, "*I feel less confident about approving it*" (R48). Another impact is the developer giving up, abandoning a code change instead of accounting for the reviewer's comments, *e.g.*, "*other times I just give up*" (R14), or leave the project, *e.g.*, "*dissociated myself a little from the codebase internally*" (R14). We also found emotions being triggered by confusion, such as anger (*e.g.*, "*It pissed me off*" — R3) and frustration (*e.g.*, "*Cannot be an effective reviewer—can replace me with a lemur*" — R40). Finally, confusion can be contagious, *e.g.*, "*It often causes confusion spreading to other reviewers*" (R12).

RQ2 Summary: We identified 14 different impacts of confusion in code reviews. The most common are *delaying*, *decrease of review quality*, and *additional discussions*. Some developers blindly approve the code change, regardless the correctness of it; other impacts include *frustration*, *abandonment* and *decreased confidence*.

4.3.6 RQ3. How do developers cope with confusion?

We found 13 topics describing the strategies developers use to deal with confusion in code reviews. Four of them are related to the review process. The most common is to *improve*

the organisation of work, such as making clearer commit messages, *e.g.*, "*Leave comments on the files with the main changes*" (R50). It is followed by spending more time and *delaying* the code review, *e.g.*, "*I need to spend much more time*" (R13). Assigning other reviewers is also a strategy adopted by developer, *e.g.*, "*Sometimes I completely defer to other reviewers*" (R48). Interestingly, *blind approval* is also a strategy developers use to cope with confusion, *i.e.*, it is not just an impact, *e.g.*, "*assume the best, (of the change)*" (R34).

Two strategies are related to the artifact. Developers make the code change smaller, *e.g.*, "*Also I ask large changes to be broken into smaller*" (R31), and clearer, *e.g.*, "*Try to make the actual code change clear*" (R12). They also improve the documentation by adding code comments, *e.g.*, "*A good description in the commit message describing the bug and the method used to fix the bug is also helpful for reviewers*" (R5).

The dimension with the most quotes is related to the developers themselves. Requesting for information on the code review tool itself is the most quoted among developers, *e.g.*, "*Put comment and ask submitter to explain unclear points*" (R15). Developers also take the discussions off-line, *i.e.*, using other means to reach their peers, *e.g.*, "*schedule meetings*" (R50) or "*ask in person*" (R1). Providing and accepting suggestions is also mentioned as a good way to cope with confusion. It includes strategies such as being open minded to the comments of their peers, *e.g.*, "*Being open to critical review comments*" (R12), and providing polite criticism, *e.g.*, "*Trying to be 'a nice person'. Gently criticizing the code*" (R3). Disagreement resolution is also a good strategy to cope with confusion, *e.g.*, "*I try to explain the reasoning behind the decisions/assumptions I made*" (R31). The use of criticism by developers in code reviews was also found in our study from Chapter 5 (published as Ebert *et al.* (EBERT et al., 2018)), but that study focused on the intention of questions in code reviews.

Regarding the link between the developer and the artifact, there are three strategies developers use to cope with confusion. Firstly, to study the code or the documentation, *e.g.*, "*It forces me to dig deeper and learn more about the code module to make sure that my understanding is correct (or wrong)*" (R12), and "*Read requirements documentation*" (R24). Secondly, to test the code change, *e.g.*, "*play with the code*" (R9). Finally, developers also use external sources to improve their knowledge about the technology, *e.g.*, "*Sometimes further research on the web [...]*" (R25).

RQ3 Summary: We have identified 13 coping strategies. Common strategies include *information requests*, *improved familiarity with the existing code*, and *off-line discussions*.

4.4 DISCUSSIONS AND IMPLICATIONS

The main contribution of our study is the empirically-driven comprehensive model of *confusion in context*, with the reasons, impacts, and coping strategies for confusion. Section 4.4.1 provides general discussion of our results. Our study suggests practical, actionable implications for the tool builders (Section 4.4.2) as well as insights and suggestions for researchers (Section 4.4.3).

4.4.1 Discussions

Confusion is an inherent part of human problem-solving, which normally arises from information and goal-oriented assessment of situations (MANDLER, 1984; MANDLER, 1990). Evidence from Psychology research demonstrates that individuals engaged in a complex cognitive task continually assimilate new information into existing knowledge structures in order to achieve completion of their tasks (D'MELLO et al., 2014). Any possible mismatch between expectations and (lack of) previous knowledge might be responsible for triggering confusion. Indeed, 92 cards (over 290) for reasons for confusion are associated to 'discussion' about the *artifact*, *i.e.*, *discussion of the solution: non-functional, strategy*, and *correctness*. Symmetrically, common coping strategies involve *information requests* and *off-line discussions*.

Confusion might trigger in individuals the experience of negative emotions. Such negative emotions depend on the context, the amount of mismatch between expectation and reality, and the extent to which completion of tasks is threatened (STEIN; LEVINE, 1991). This is in line with the observation of *decreased confidence*, *abandonment*, and emotions such as *frustration* and *anger*, among the impacts of confusion. Early detection of confusion (EBERT et al., 2017) becomes crucial for preventing contributors' burnout and loss of productivity (MÄNTYLÄ et al., 2016). Such situations might even cause the abandonment of the project, which is undesirable as it eventually leads to undesired turnover; since developers focusing on documentation are more susceptible to turnover than those working on code (LIN; ROBLES; SEREBRENIK, 2017), project abandonment is likely to exacerbate *lack of documentation* further increasing confusion within the project.

A possible antidote to negative emotions, identified as a coping strategy by the survey respondents, is being open minded when *providing and accepting suggestions*. This attitude is often mandated by codes of conducts adopted by open source projects (TOURANI; ADAMS; SEREBRENIK, 2017), requiring, *e.g.*, to "gracefully accept constructive criticism" and "be respectful of differing viewpoints and experiences". One should be aware, however, that imposing such rules might lead to emotional labor (SEREBRENIK, 2017), *i.e.*, the process by which individuals are expected to handle their feelings in harmony with the rules and guidelines defined by the organization.

4.4.2 Implications for Tool Builders

Code reviews are supported by tools such as Gerrit. Currently, the only feature of Gerrit that we can relate to confusion reduction is flagging large code changes. Indeed, *large, complex changes* are among the most popular reasons for confusion in our model. As a strategy to deal with this issue, developers suggest to split big changes into smaller, clearer ones. This is consistent with the earlier findings (BARNETT et al., 2015; TAO; KIM, 2015; PASCARELLA et al., 2018), envisioning the emergence of tools enabling early detection of splittable changes, *i.e.*, before the pull request is submitted, in order to avoid both spending additional time in identifying such patches and asking the author re-work the change to reduce its complexity and likelihood of the discussion (THONGTANUNAM et al., 2017).

Based on our results, further tool improvements can be envisioned. We observe that the second most popular coping strategy is to *improve familiarity with existing code*. The burden of this task might be reduced if code review tools could provide the task context (LATOZA; VENOLIA; DELINE, 2006). Similarly, a summary of the change (PANICHELLA, 2018; RIGBY; STOREY, 2011) could be beneficial to overcome confusion due to *lack of understanding of the change*. *Organisation of work* can be improved by tools capable of automatically generating commit messages (LINARES-VÁSQUEZ et al., 2015; HUANG et al., 2017), disentangling commits performing multiple tasks (DIAS et al., 2015) and combining multiple commits performing one task (ARIMA; HIGO; KUSUMOTO, 2018). Furthermore, information retrieval tools able to understand written design discussions occurring in pull requests (VIVIANI et al., 2018a) could be integrated into code review tools to extract rationale, implicit knowledge, and other contextual information otherwise not available during the review process. Integrating such information into the documentation might prevent confusion due to a *missing rationale*, *lack of context*, or doubt about *necessity of a change*.

Another reason for confusion is the difficulty in assessing the *impact of the change*. Integration of change impact analysis tools might be thus beneficial. Similarly, integration of tools assessing the test coverage of a change might keep developers from committing changes with low test coverage, thus avoiding confusion due to *lack of tests*. Information about test coverage could be also integrated in the pull request, *i.e.*, by mean of badges as done by tools such as Coveralls.[8]

Developers also report *off-line discussions* as a strategy to quickly *resolve disagreement* as well as ambiguities in *communicative intentions*. This evidence is consistent with findings from previous research by Pascarella *et al.* (PASCARELLA et al., 2018), who also envision the integration of synchronous communication functions into code review tools to enable traceability of decisions and explanation provided, and, allow their integration into documentation for future reference.

[8] https://coveralls.io

Going beyond code review tools, developers experience confusion in using issue tracking or version control systems. Hence, these tools can be improved to facilitate their usage.

4.4.3 Implications for Researchers

In our model of *confusion in context*, we observed reasons for confusion are far more than coping strategies. Indeed, strategies are derived by the analysis of developers' self-report in the survey and represent what they already do to deal with confusion. Conversely, reasons for confusion are defined based on the analysis of both survey responses and developers' comments in code reviews. Of course, a symmetry is observed for those reason-strategy couples addressing the same cause of confusion, *e.g.*, *small, clear changes* are offered as a solution for confusion due to *long, complex changes*; *information requests* can address difficulties in understanding the others' *communicative intentions*; and *lack of familiarity with existing code* is addressed by studying the code and its documentation (*improved familiarity with existing code*). A significant amount of reasons, though, are not addressed. Addressing these topics with follow-up studies might be beneficial to identify what *could* be done and how, in order to improve code review, in addition to what is already done and reported by developers.

We observed that the assignment of other reviewers in the discussion is either an impact and a strategy developers adopt to deal with confusion. Thus, confusion can be beneficial as to contributing to decrease the number of bugs in the software as more reviewers tend avoid bugs (BAVOTA; RUSSO, 2015; MCINTOSH et al., 2014). Moreover, the inclusion of more people in the code review increases their awareness of the code change, *i.e.*, confusion resolution contributes to knowledge sharing. However, involving additional reviewers induces additional workload, while reviewers expertise might already be scarce. Indeed, Ruangwan *et al.* have observed that 16%–66% review requests have at least one invited reviewer who did not respond to the invitation (RUANGWAN et al., 2018). This is why research should consider both early detection of confusion alleviating the need of involving additional reviewers, and more precise recommendation of reviewers for a given code change (THONGTANUNAM et al., 2015).

Our results show that presence of confusion causes developers to move discussion off-line, *i.e.*, outside the code review tool. This finding is similar to earlier observations (ARANDA; VENOLIA, 2009), suggesting that despite the omnipresence of platforms such as Gerrit and GitHub fostering transparency of software development, cases experienced as confusing sometimes remain invisible. This finding might threaten validity of previously published studies of code reviews that have been solely based on mining digital trace data. More comprehensive research methods should therefore be sought.

The most popular coping strategy is *information requests*. So far, little research was conducted on information needs in code reviews (EBERT et al., 2018; PASCARELLA et al.,

2018). Our study from Chapter 5 (published as Ebert *et al.* (EBERT et al., 2018)) shows that almost half of the developers' questions have the intention to seek for some kind of information, such as confirmation, information, rationale, clarification, and opinion. The information needs expressed by developers during code reviews (PASCARELLA et al., 2018) are closely related to the reasons for confusion in Table 11. Indeed, *suitability of an alternative solution* (PASCARELLA et al., 2018) is related to the *discussion of the solution strategy*, *splittable* (PASCARELLA et al., 2018) to *long, complex change* and *specialised expertise* to *lack of familiarity with technology*. We believe, therefore, that confusion may trigger both of the requests for information (EBERT et al., 2018) and the information needs (PASCARELLA et al., 2018). Hence, we see a clear venue for researchers to understand better both the information needed during a code review and the ways developers aim at obtaining it. Using this understanding, one should try to automatise supplying the relevant information to resolve confusion.

4.5 THREATS TO VALIDITY

As any empirical study, our work is subject to several threats of validity (RUNESON et al., 2012). We identified three kinds of threats to its validity: construct, internal, and external, all of which are discussed below.

Construct validity. The threats to construct validity concern the relation between the concept being studied and its operationalisation. In particular, it is related to the risk of respondents misinterpreting the survey questions. To reduce this risk we included our own definition of confusion and requested the respondents to confirm that they understood it. For the same reason, we always anchored the frequency questions and adhered to well-known survey design recommendations (GROVES et al., 2009; KITCHENHAM; PFLEEGER, 2008; SINGER; VINSON, 2002; STEELE; ARONSON, 1995). We believe that this helps to diminish misunderstanding of this concept during the survey. Furthermore, despite following all best recommendations to build and deploy a survey, we used both closed and open-ended questions to get as much data as possible.

Internal validity. The threats to internal validity pertains to inferring conclusions from the data collected. The card sorting adopted in our work is inherently subjective because of the necessity to interpret text. To reduce subjectivity, every card sorting step has been carried out by several researchers. Moreover, to assure the completeness of the topics related to the reasons, impacts and confusion coping strategies, we conducted several survey iterations until the data saturation has been achieved, and augmented the insights from the surveys with those from the code review comments. As an additional step, we also double checked the reasons for confusion with an additional card sorting.

External validity. The threats to external validity are related to the generalisability of the conclusions beyond the specific context of the study. Our first survey targeted only a single project: ANDROID. However, the second and the third ones targeted a general soft-

ware developer population. Statistical analysis has not revealed any differences between the respondents of the different surveys suggesting that the answers obtained are likely to reflect opinions of the code review participants, in general. To complement the surveys, we consider 307 code review comments from GERRIT. While the functionality of GERRIT is typical for most modern code review tools, developers using more advanced code review tools do not necessarily experience confusion in the same way. For instance, COLLABORATOR[9] supports custom templates and checklists that, if properly configured, might require the change authors to indicate rationale of their change, reducing the importance of *missing rationale* from Table 11.

4.6 RELATED WORK

In this section, we discuss the work related to confusion in code reviews. We also discuss about studies related to what is needed to understand code changes.

Tao *et al.* (TAO et al., 2012) investigated how the understanding of code changes affects the development process. They conducted surveys and follow-up emails with software designers, testers, and software managers at MICROSOFT. They shown that *rationale* is the most important information for understanding a code change. However, respondents mentioned that code changes can be easily understood if a good description is provided. They discovered that reviewers could benefit more from the code-exploration features provided by common IDEs (*e.g.*, *call hierarchy* from Eclipse) when they are exploring the change context and estimating its risk.

Ram *et al.* (RAM et al., 2018) aimed to obtain an empirical understanding of what makes a code change easier to review. They empirically defined *reviewability* as how the code change is: i) explained (*e.g.*, in the change description), ii) properly sized and self-contained (*e.g.*, small changes), and iii) aligned with the coding style of the project. They researched academic literature papers, and also blogs and white papers, interviewed professional developers, and evaluated a tool to rate the reviewability of code changes. They found that reviewability is affected by several factors, such as the change description, size, and coherent commit history.

Uwano *et al.* (UWANO et al., 2006) proposed the use of eye tracking to characterise the performance of developers performing code reviews. They developed a system which captures the source code line number the reviewer's eye is looking at. It is also able to record the transition from a line to another when the reviewer's eyes move, as well as the time spent at each line. Their system was used to perform an experiment with five students reviewing code changes. As result, they identified a specific pattern in reviewer's eyes: "scan". This pattern is characterised by the reviewer's action of reading the entire

[9] https://smartbear.com/product/collaborator/overview

code before investigating in details each line. Furthermore, reviewers who did not spend sufficient time for the scan tend to take more time for finding defects.

Gopstein *et al.* (GOPSTEIN et al., 2017) introduced the term *atom of confusion* which is the smallest code pattern that can reliably cause confusion in a developer. Through a controlled experiment with developers, they studied the prevalence and significance of the atoms of confusion in real projects. They shown that the 15 known atoms of confusing occur millions of times in programs like the LINUX kernel and GCC, appearing on average once every 23 lines. They reported a strong correlation between these confusing patterns and bug-fix commits, as well as a tendency for confusing patterns to be eventually commented.

Pascarella *et al.* (PASCARELLA et al., 2018) investigated, by analysing code review comments, what information reviewers need to perform a proper code review. They analysed threads of comments which started from a reviewer's question from a total of 900 code reviews. Additionally, semi-structured interviews and one focus group with developers were conducted to understand the perceptions of the code review needs from developers. They found seven high-level information needs, such as the suitability of an alternative solution, the correct understanding of the code change, rationale, and the context of the code change.

The work presented in this study is complementary with respect to the ones discussed so far. To the best of our knowledge, this is the first study that aims at building a comprehensive model of what make developers confused during code reviews, their impacts and what strategies do developers implement to overcome confusion.

4.7 SUMMARY

In this chapter, we tackled the problem of *lack of knowledge* about confusion in code reviews by building a comprehensive model for *confusion in context*. This model includes the reasons and impacts of confusion, as well as the coping strategies adopted by developers. We used a concurrent triangulation strategy combining a series of developer's surveys and an analysis of code review comments.

We found 30 reasons for confusion, with the most common ones being *missing rationale*, *discussion of the solution: non-functional*, and *lack of familiarity with existing code*. Among the 14 impacts of confusion, the prevalent are *delaying*, *decrease of review quality*, and *additional discussions*. Finally, developers employ 13 strategies to cope with confusion, such as *information requests*, *improve the familiarity with the existing code*, and *off-line discussions*. Our study has several implications for both tool builders and researchers. Code review tools could be improved by integrating information that can reduce confusion, *e.g.*, related to rationale, test coverage or impact of the change. Researchers should investigate possible relations between confusion and information needs, as well as between confusion and migration of code review discussions to off-line channels.

5 COMMUNICATIVE INTENTIONS OF QUESTIONS

Communicative intentions are one of the reasons for confusion related to the developer dimension of code reviews. Furthermore, the error analysis of our classification models shows that intention of questions strongly influences the misclassification of *no confusion* comments. As such, in this chapter, we present an exploratory, in-depth study of communicative intention of developers' questions in code reviews.

A brief overview is presented in Section 5.1, and the background related to communicative intention is discussed in Section 5.2. The methodology used in this study is shown in Section 5.3. Section 5.4 presents the results. Then we discuss the results and their implications in Section 5.5. Section 5.6 discusses the threats to validity, and Section 5.7, the related work. Lastly, we summarise this study in Section 5.8.

5.1 OVERVIEW

As discussed in Chapter 2, during a code review developers might identify a defect, suggest a better solution, or ask about the rationale behind the implementation choices. When reviewing code changes, developers might provide either general or inline comments. General comments are posted in the code review page itself, which presents the list of all general comments as threads of messages. Inline comments are posted directly into the source code file and can reference a word, a line, or a group of lines. They are intended to be a dialogue between the reviewer and the code change author, as recognised by developers themselves, *e.g.*, Alan Fineberg, software engineer at Yelp: "*The reviewer then goes through the diff, adds inline comments on review board and sends them back. [...] The reviews are meant to be a dialogue, so typically comment threads result from the feedback*"[1]. As such, we focus on questions extracted from *inline comments*.

Code review discussions represent an invaluable source of information ready to be mined for i) extracting information about a software project and its evolution (VIVIANI et al., 2018a), and ii) understanding how developers conduct code review, *i.e.*, which understanding needs they try to fulfil (BACCHELLI; BIRD, 2013), and what comments they perceive as useful (BOSU; GREILER; BIRD, 2015; EFSTATHIOU; SPINELLIS, 2018). We believe that the identification of communicative intentions expressed by developers in their comments, such as making a direct suggestion, requesting a clarification, and expressing disagreement, is essential to this latter perspective. In line with this view, Viviani *et al.* (VIVIANI et al., 2018a) recently proposed to mine developers' discussions in pull requests to extract design information that explicitly documents design decisions (VIVIANI et al., 2018b), based on the analysis of the dialogue argumentative structure (VIVIANI et al.,

[1] Quoted by Marty Stepp in their slides <https://courses.cs.washington.edu/courses/cse403/13sp/lectures/10-codereviews.pdf>

2018a). Specifically, they argue in favour of future research aimed at developing approaches able to discover design elements in developers' discussions.

Furthermore, identification of the meaning of non-literal statements is also a well-known problem for individuals with high functioning autism (CHAHBOUN et al., 2016). One might therefore wonder whether challenges in identifying communicative intentions contribute to the lower self-assessment of neurodiverse software developers on requesting code reviews and reviewing other people's code (MORRIS; BEGEL; WIEDERMANN, 2015). Examples are those with autism spectrum disorder (ASD), attention deficit hyperactivity disorder (ADHD), and/or other learning disabilities, such as dyslexia (MORRIS; BEGEL; WIEDERMANN, 2015).

We envision the emergence of tools to support developers during code review based on the automatic analysis of the communicative goals conveyed by developers' comments. To this aim, we present the first study of the communicative intentions expressed in code review questions. We focus on *questions* as they have been recently described as triggers of useful conversation excerpts in code review, *i.e.*, in design discussions (VIVIANI et al., 2018a; VIVIANI et al., 2018b) or in knowledge-sharing (BACCHELLI; BIRD, 2013). Furthermore, our findings that questions presented the lowest agreement on confusion during the manual annotation among the raters (cf. Section 3.3), and that the intentions of questions are one of the main reasons for the classification models to predict wrongly *no confusion* comments (cf. Section 3.5.2.1) intrigued us to better understand developers' questions in code reviews.

We performed an exploratory study (RUNESON; HÖST, 2009) on the questions asked by developers in the inline comments of ANDROID code reviews. We manually classified 499 questions derived from 399 code reviews in GERRIT. Our findings suggest that questions in code review serve diverse communicative goals, *e.g.*, requesting clarifications, discussing hypothetical scenarios, and suggesting improvements. We observed that the majority of questions serve information seeking goals. Still, they represent less than half of the annotated sample, providing evidence that questions may convey a wider variety of developers' communicative intention. A large category is represented by the questions aimed at eliciting an action of the collaborators, *i.e.*, developers usually employ politeness when making suggestions by using questions instead of affirmative sentences.

Code change authors interviewed by Bosu *et al.* (BOSU; GREILER; BIRD, 2015) consider clarification questions as "not useful", as they do not immediately contribute to code improvement. However, such questions have been reported as useful to improve the reviewer's understanding of the change under review, which in turn can lead to improvement suggestions being formulated later on. Furthermore, as already observed by Bosu*et al.* (BOSU; GREILER; BIRD, 2015), those clarification questions can be useful to trigger knowledge transfer discussion between the contributors of a software project. This was mentioned as a reason for conducting code reviews, beyond finding defects, by all but one

of the interviewees in the study of Bacchelli and Bird (BACCHELLI; BIRD, 2013). They reported that developers interact during code reviews to fulfil their understanding needs, to speed-up and improve the review process. Our findings suggest that analysis of code review comments should be able to distinguish, at least, between requests and clarification questions.

Gachechiladze *et al.* argued that identification of anger can be used to design tools and interventions supporting developers (GACHECHILADZE et al., 2017). Complementarily, identification of communicative intention can provide a starting point in recommendations for good practices. If, for example, polite requests for action, rather than direct suggestions are more common among code reviews deemed to be useful (BOSU; GREILER; BIRD, 2015), then developers can be encouraged to embrace politeness (cf. discussion of how to ask for help on StackOverflow (CALEFATO; LANUBILE; NOVIELLI, 2018b) and to embrace envisioned guidelines on how to communicate efficiently in code reviews (EFSTATHIOU; SPINELLIS, 2018)). Identifying requests for clarification may also benefit in the development of a recommending system, identifying lack of reviewer expertise and triggering expert intervention (BOSU; GREILER; BIRD, 2015). Moreover, requests for rationale or suggestions of alternative solutions are steps towards identifying design discussion in code review (VIVIANI et al., 2018a).

Our findings, while preliminary, suggest research hypotheses worth investigating in future work. Following the guidelines of Runeson and Höst (RUNESON; HÖST, 2009), we formulate a set of hypotheses that should be confirmed or refuted by follow-up studies.

5.2 BACKGROUND ON COMMUNICATIVE INTENTION

People proceed in their conversations through a series of *speech acts* to yield a specific communicative intention: they ask for information, agree with their interlocutor, state facts and express opinions (ALBRIGHT et al., 2004). Speech acts constitute the basic unit of verbal communication and are well studied in linguistics and computational linguistics (AUSTIN, 1962; SEARLE, 1969; CORE; ALLEN, 1997; TRAUM, 2000). Traditionally, speech acts have been analysed in the action view initiated by Austin (AUSTIN, 1962) in which they are treated as a part of the theory of actions (HOLTGRAVES, 2002). Such as actions influence the state of affairs in the physical world, speech acts affect the cognitive state of participants on a dialogue. To some researchers, speech acts represent the minimal and primitive unit of linguistic communication (SEARLE, 1969); others refer to dialogue acts as more complex events involving interaction between the mental states of the participants on a dialogue (COHEN; LEVESQUE, 1990). However, there is a general consensus in the research community on modeling the whole linguistic communication as a complex phenomenon involving communicative intentions (on the speaker side) and their recognition (on the hearer side) (ALBRIGHT et al., 2004).

In the linguistic community, there is a general consensus on identifying the speech act associated to an utterance with the communicative intention of the speaker, *e.g.*, ask a question, criticise, impress, comfort or self-disclose through objective statements or opinions. Nevertheless, there is no simple way to relate a specific conversational statement and the speech act it represents (FUSSELL; KREUZ, 1998; KRAUSS et al., 1981). For instance, the attempt to establish communication might be expressed by the question "How are you today?". However, the very same intention (speech act) might have been expressed using different words. Moreover, the very same words could suggest a different communicative intention, depending on pragmatics and contextual information: *e.g.*, when a doctor asks a patient the very same question, it is more likely that they are inquiring into their patient's health rather than merely greeting them.

Hence, for the communication to be effective, all persons involved in the conversation must agree on the intention of the message (ŽEGARAC; CLARK, 1999). Indeed, overt communication can be seen as an error-prevention strategy (CROOK, 2004) and misinterpretations can arise from misunderstanding of the communicative intention (HAUGH, 2012). When such misunderstandings are perceived as soon as they occur, conversational repair can be employed as an error recovery strategy (GELUYKENS, 1994). In this study, we focus on the communicative intentions conveyed by developers during code review. Specifically, we analyse their communicative goals: the *illocutionary force* (AUSTIN, 1962) (*i.e.*, the speaker's intention) of the questions asked during code review.

5.3 METHODOLOGY

We study the communicative intentions conveyed by comments contributed by developers during code review. Specifically, we focus on questions as they can be used to convey a multitude of communicative intentions, such as requests for clarification, discussions of hypothetical scenarios and suggestions for improvements.

Developers' questions have been investigated in previous software engineering research, with specific focus on technical Q&A sites (TREUDE; BARZILAY; STOREY, 2011; BAJAJ; PATTABIRAMAN; MESBAH, 2014; CALEFATO; LANUBILE; NOVIELLI, 2018b). However, *a priori* it is not clear how frequent are the questions in *code reviews*. Indeed, if questions are infrequent in code reviews, further classification of communicative intentions expressed in questions might be irrelevant. Hence, we first ask:

RQ1. *How frequent are questions in code reviews?*

Next, we take a closer look at the communicative intentions expressed by these questions:

RQ2. *What are the communicative intentions expressed in the developers' questions in code reviews?*

5.3.1 Case Study Subject: Android

To answer our research questions, we conduct an exploratory case study (RUNESON; HÖST, 2009). Indeed, **RQ2** is exploratory in nature and we aim at formulating hypotheses that can be confirmed or refuted by follow-up studies (EASTERBROOK et al., 2008).

As a starting point, to answer the **RQ1**, we leverage the our full datasets from Chapter 3 of general and inline comments from the entire ANDROID ecosystem. The total number of questions found in the general and inline comments are 12,686 and 37,712, respectively. For **RQ2**, we extracted the annotation sample only from the inline comments. We decided to consider only inline comments because, as explained earlier, we believe they tend to be more similar to dialogues than the general ones. We considered only the **questions** group of our ANDROID dataset to answer **RQ2**. Our annotation sample contains 499 questions extracted from randomly selected 399 inline comments (corresponding to the confidence interval of less than 5% and confidence level of 95% from a population of 33,711 questions).

5.3.2 Identifying Questions

To answer **RQ1** and lay foundations to answering **RQ2**, one should be able to distinguish between questions and non-questions. A naive approach would be to identify questions based on the presence of question marks "?". However, not all questions would contain the question mark and not all question marks introduce a question. Indeed, on the one hand, questions might be implicit, *e.g.*, "*Would you point me to a an example of one of these test stanzas you're talking about.*" and "*can we bring quick_ into the naming here.*" Moreover, the writing style of code review comments tends to be quite informal and developers might omit the question mark when asking a question (cf. discussion of punctuation omission in instant messaging (ZHOU; ZHANG, 2005)). On the other hand, question marks in the comments do not necessarily indicate questions as they might be part of source code fragments or URLs. As an alternative to the naive approach, we use the STANFORDNLP API (MANNING et al., 2014) to parse the comments.

For example, the following questions:

- "*Don't we need to increment 'i' in the else case here to avoid an infinite loop?*"

and

- "*are you sure you want to include this source file directly? Why not create a static library?*"[2]

both express suggestion, even though the first is in the inverted yes/no question and the latter is a wh-question.

[2] From now on, in comments containing multiple sentences we highlight the question we are referring to by underlining it.

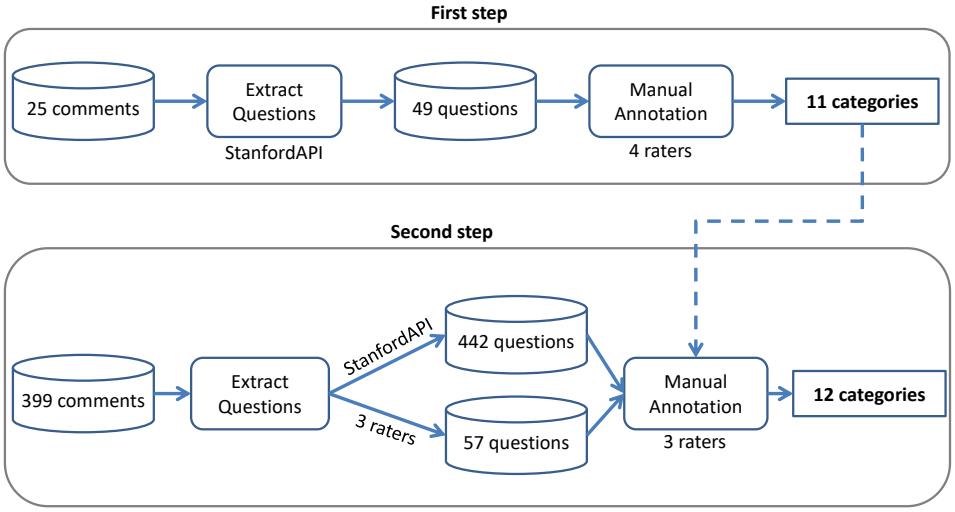

Figure 16 – Methodology adopted in the manual labeling.

5.3.3 Manual Labeling

To answer **RQ2**, we performed a manual annotation study on questions extracted from code review comments (see Fig. 16). For this manual annotation, four raters, holding at least a master's degree in Computer Science, classified the questions in the first step. For the second step, only three raters (from those four) participated in the annotation process.

The first step aimed at defining the coding schema using an open coding methodology, *i.e.*, without predefined categories (STRAUSS; CORBIN, 1990; ZIMMERMANN, 2016). During this step, the four raters individually annotated the 49 questions extracted from 25 randomly sampled comments. The annotation was performed at the sentence level, *i.e.*, the individual questions rather than comment as a whole are used as unit of analysis. However, the whole comment was presented to provide context for the annotation. The raters were requested to assign a single label to each question indicating the communicative intention it conveyed. After the annotators completed their assignments, the results of the individual labeling were discussed to solve cases of disagreement. Based on the first annotation step, we designed an initial taxonomy including 11 question categories that were used as guidelines for annotation during the second step.

In the second step, the three raters were requested to individually label the communicative intention of 442 questions. They were instructed to perform the labeling based on the 11 categories in the initial taxonomy. However, they could suggest any missing categories they found relevant. Once the individual annotation was completed, we compared the labels from each rater and compiled them into one final set. As a result, one new category was added to the initial ones, *i.e.*, criticism, leading to the final list of 12 categories. Once again, the single question (sentence) was considered as a unit of analysis. Notwithstanding, the whole comment was provided as context to the rater. We used Fleiss' kappa (FLEISS, 1971) to measure the agreement, and majority voting to resolve

the disagreements. When majority vote was not possible, *i.e.*, when all the votes were different, the disagreement was resolved through discussion: the raters have jointly reviewed the cases and decided upon the most appropriate label.

During the manual annotation of the 442 questions, the raters found 57 additional questions not identified by the StanfordNLP API. These questions were included in our analysis of communicative intentions, thus resulting in 499 questions overall in our manually annotated dataset. However, the annotation of those 57 questions did not include any new question category. During the joint discussion, the raters unanimously identified and discarded 2 declarative sentences that were erroneously classified by the StanfordNLP API as questions. Furthermore, 1% (5) of the questions were discarded given the inability of the raters to reach an agreement on the label to assign to them. For instance, questions as "*Is this line to be removed?*" and "*Do we need two timings here?*", can be plausibly interpreted both as suggestions, *i.e.*, "Remove this line" and "Use only one timing", and as requests to the code change author to confirm the reviewer's understanding of the author's intention. Such cases were discarded given the inability of the raters to reach an agreement on the label to assign to them. In the end, our sample included 492 questions. The dataset of questions with corresponding communicative intention labels is publicly available for research purposes (ebert, 2019).

5.4 RESULTS

In this section, we organise the results according to each research question.

5.4.1 RQ1: How frequent are questions in code reviews?

The number of questions is three times higher in inline comments than in the general ones. Table 12 shows that inline comments contains 75% (37,712) of the questions, while general comments represent only 25% (12,686). We show the number of general and inline comments with at least one question in Table 13. We can observe that 1.65% of the general and 14.5% of the inline comments contain at least one question. These results suggest that developers ask more questions in the inline comments.

Table 12 – Distribution of questions.

Comment	Number of Questions
General	12,686 (25%)
Inline	37,712 (75%)
Total	50,398 (100%)

Table 13 – Distribution of comments with questions.

Comment	With Questions	Without Questions	Total
General	10,965 (1.65%)	649,880 (98%)	660,845
Inline	33,711 (14.50%)	198,760 (85%)	232,471

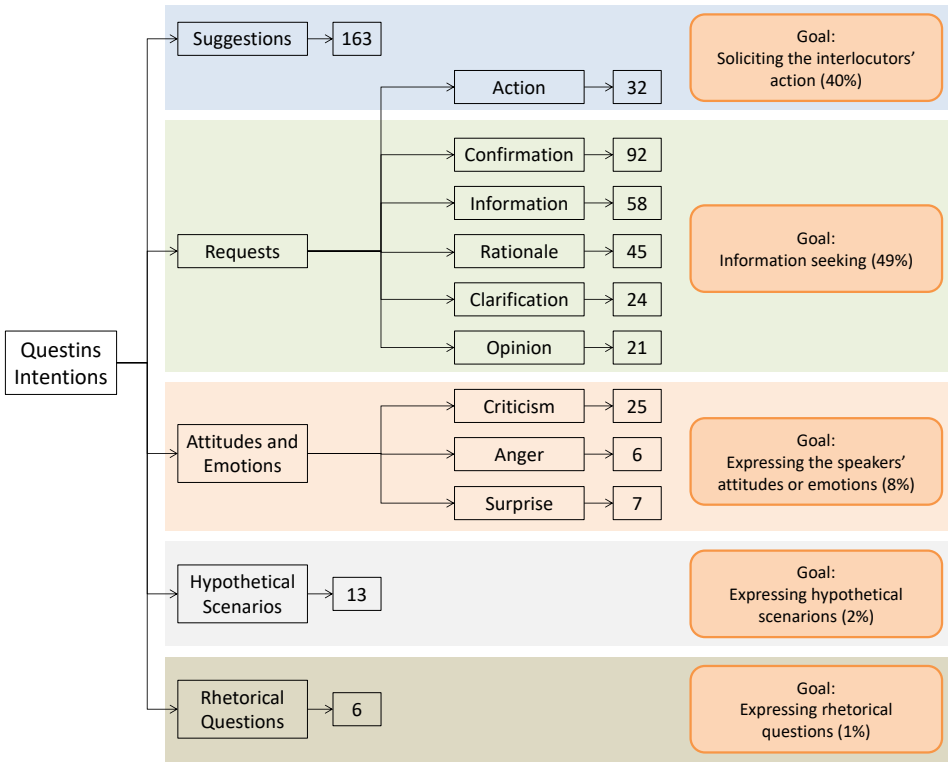

Figure 17 – Questions intention classification tree.

5.4.2 RQ2: What are the communicative intentions expressed in the developers' questions in code reviews?

The manual labeling agreement between the three raters measured with Fleiss' kappa is 0.40, indicating a moderate agreement (LANDIS; KOCH, 1977). Figure 17 presents the questions' category classification tree. In the following, we discuss in details the classification of questions intentions. We group and discuss the intentions according to their communicative intention as follows: i) suggestions and requests for actions are discussed in Section 5.4.2.1, ii) all kinds of information seeking questions are introduced in Section 5.4.2.2 (requests for confirmation, information, rationale, clarification and opinion), iii) the questions expressing the speaker's cognitive and affective states, *i.e.*, attitudes such as criticism or emotions, such as anger and surprise, are discussed in Section 5.4.2.3, and iv) hypothetical scenarios and v) rhetorical questions are discussed in Section 5.4.2.4 and 5.4.2.5, respectively.

5.4.2.1 Soliciting the interlocutor's action

A considerable amount of questions is used to give a suggestion or to request an action, *i.e.*, to elicit an action of the interlocutor, albeit with a different degree of strength, from suggestions to direct requests for action (see the examples reported below). Looking at Figure 17, we observe that triggering the interlocutors/'peers' action is the actual speaker's goal in 40% of questions, based on the sum of suggestions (163) and request for action (32).

Suggestions. More than 33% (163) of the questions actually convey the intention of suggesting something, *i.e.*, an alternative solution or a different implementation choice. The developers usually try to be polite when making suggestions to their peers by using questions instead of direct affirmative sentences (DANESCU-NICULESCU-MIZIL et al., 2013):

- "*are you sure you want to include this source file directly? Why not create a static library?*";
- "*Maybe introduce an additional line between 'abc' and 'def'?*";
- "*Do we need the conditional? How about assigning is_success directly?*".

In some cases, suggestions consists of merely one word, suggesting an action or describing the desired source code modification:

- "*const?*";
- "*Remove?*";
- "*Multicatch?*".

Request for action. In 7% (32) of cases, reviewers rather make a direct request for action, *i.e.*, they directly ask the change authors to perform a specific action. The intention is to induce the interlocutor to perform a specific action. As opposed to suggestions, requests for action explicitly refer to the individual that is expected to take action:

- "*keeping with the other functions in this file, can you put each argument on its own line?*";
- "*Can you make these different? For example, can this one be 2001:db8:1::13?*";
- "*Can you move this inside _nvmap _page _pool _fill _lots _locked()?*".

5.4.2.2 Information seeking

The majority of questions (49%) serve information seeking goals. This communicative intention is expected because it is intrinsic to the code review process. Looking at Figure 17, we observe that all requests together sum up to 240, if we exclude requests for action, whose purpose is the same communication goal of suggestions. Conversely, all other request categories serve the speaker's goal to elicit information sharing action from the interlocutor. Specifically, developers try to fulfill different information needs while reviewing code changes, *i.e.*, they ask questions to clarify their understanding of the code change. To accomplish that, they usually need to request some kind of information.

Request for confirmation. More than 18% (92) of questions are request for confirmation, *i.e.*, the reviewer has a certain degree of understanding but expresses doubt and seeks approval from the code change author. Usually they have the format of yes/no questions:

- "*shouldn't this just be a failure? [...]*";
- "*This size_ includes the size of the header, correct? [...]*";
- "*Is request cloning still necessary?*".

Request for information. In 11% (58) of cases, reviewers perform requests for information, *i.e.*, reviewers have partial understanding of the problem and ask a question aimed at obtaining missing technical details required to complement the understanding:

- "*When can this be null? [...]*";
- "*Where are these entries populated?*";
- "*what's the exact format? [...]*".

Request for rationale. In 9% (45) of cases, the developer performs a request for rationale to understand the reason behind an implementation choice:

- "*Why is this included? [...]*";
- "*why are you releasing peristent mem buffers here?*";
- "*Why do you add 2 attribute list end marker?*".

Requests for clarification. About 4% (24) of the questions represent developers trying to clarify something from the code change. We exclude the requests to clarify the rationale behind the change as requests for rationale are relatively common and constitute their own category.

- "*What's happening here?*";
- "*Can you clarify what you mean?*";
- "*How is this different from the one above?*".

Request for opinion. Developers also make comments in the code review asking for others' opinions when they do not know how to proceed with something. We found that 4% (21) of the questions have such intention:

- "*Is there a cleaner way of expressing this? This works, but doesn't feel right.*";
- "*[...] What do you think? Thanks*";
- "*Which name do you suggest?*".

5.4.2.3 Expressing attitudes and emotions

In 8% of questions, the actual developers' communicative intention is to express their attitudes, opinions or emotions. Developers express their attitude of doubt through criticisms, and even share such attitude of perplexity or disagreement by communicating their emotions with different degrees of strength, from surprise to anger. In all these cases, the final goal of the speaker is to express their own cognitive and emotional state to induce critical reflection in the interlocutor about the topic being discussed.

Criticism. About 5% (25) of the questions contain some level of negativity, but could not be classified as expressing anger; they rather express criticism towards an implementation choice made by the interlocutor:

- "*And burning a register? :-)*";
- "*Do you really want to return the address of a local variable here?*";
- "*why do blank lines show up as code changes? Pls avoid this.*".

Anger and Surprise. We also observe few cases where developers expressed their emotions through questions. They represent 2% (13) of our question dataset, of which 1.22% (6) were classified as **anger** and 1.42% (7) as **surprise**.

For anger, there is one case where the direction of the anger is the speaker itself:

- "*free does not affect errno, what was I thinking?*".

The other questions have the anger targeted interlocutors or objects/situations:

- "*wtf? you _really_ want reflection here.*";
- "*What's all this?*";

- "*Agree. What the heck is this. Max jitter? If so, please use jitter*".

Finally, in some cases developers express their surprise with a question:

- "*is this true? that seems mildly surprising. did you run the tests against glibc too?*";
- "*Is this true? Shouldn't it be /dev/pmsgX and /sys/fs/pstore/pmsg-ramoops-X instead?*".

In the other case the developer is surprised by the compiler:

- "*remove? (i'm surprised clang doesn't complain about this?)*".

5.4.2.4 Hypothetical scenarios

Questions describing hypothetical scenarios (2%) may serve different communication purposes related to *what-if* scenarios, from information seeking to expression of criticism or doubt. Since context is needed to disambiguate the speakers' intention, we decided not to map *a priori* the hypothetical scenarios to categories discussed above. Developers tend to use wh-questions in combination with conditional conjunctions to describe hypothetical scenarios:

- "*What about if an already Jack server is running?*";
- "*what happens if r1 is sample and r2 is not?*";
- "*what if you see a sequence like c2 20 split across two calls?*".

5.4.2.5 Rhetorical questions

These are usually used when developers raise a question to answer it later themselves. Such questions serve the communicative goal of providing evidence and argument to support the claim made right after the question. They constitute only 1.22% (6) of the questions, some examples are:

- "*Isn't the case that you illustrated (0.9ms being decremented as 0) applicable in both solutions? Yes, the solution you offered ensures that the loop doesn't spend time on the extra instruction sets [...]*";
- "*[...] i asked rhetorically in the checkin comment then "what is this for?"*";
- "*[...] However, we have library that depends on this one that seems to compile either way. What gives? I would have expected some paths to be wrong.*".

5.5 DISCUSSIONS

We observed that developers ask more questions in the inline comments than in the general ones. This observation concurs with our expectations that the inline comments tend to be more similar to a dialog than the general comments.

In line with findings from previous research suggesting that code review also serves knowledge-transfer purposes (BACCHELLI; BIRD, 2013), we observed that the majority of questions are information seeking requests, including requests for confirmation, information, rationale, clarification, and opinion. This suggests a rich variety of information needs experienced by developers during code review. While representing the majority of questions, information seeking requests are actually less than half of the annotated sentences in our dataset (49%). In particular, the second most frequent goal addressed by developers' questions is to elicit a reaction of the interlocutor (40%). This evidence suggests that we should not consider questions as purely information seeking activity. Conversely, developers use sentences in form of questions to serve a wider variety of communication goals.

The second most frequent goal addressed by developers' questions is to elicit a reaction of the interlocutor (40%), *i.e.*, by either making a polite suggestion (33%) or directly performing a request for action using an imperative sentence to give a direct command (7%). The use of indirect language to perform a suggestion might be seen as an indication of politeness, as indirect language is considered as a clear rhetorical tool for conveying a kind attitude towards the interlocutors (DANESCU-NICULESCU-MIZIL et al., 2013).

Finally, developers use questions to express doubt or disagreement with different degrees of intensity, *e.g.*, with surprise, criticism expressions, and anger. This evidence indicates that developers also express their attitude of doubt and perplexity, and tend to disagree with the implementation choices of their peers with different degrees of intensity. Such intensity ranges from: i) surprise (1.42%) about unexpected code behavior, generally accompanied by a neutral attitude (*e.g.*, "*I am surprised clang doesn't complain about this*"), ii) criticism expressions (5%) to convey both perplexity and either a slightly negative attitude towards the interlocutors' or a slightly negative evaluation of their implementation choice (*e.g.*, "*Do you really want to return the address of a local variable here?*"), and iii) actual anger (1.22%) to communicate both strong disagreement and negative attitude towards the interlocutor (*e.g.*, "*wtf? you _really_ want reflection here*"). In all those cases, the actual goal of questions belonging to these categories is to express the speakers' own cognitive and emotional state to induce a critical reflection in the interlocutor.

One may argue that the percentage of questions conveying anger is quite limited to justify relevance of follow-up studies. However, it is comparable to that reported in previous research on emotions in developers' comments in issue tracking systems (MURGIA et al., 2014). Specifically, we found that *anger* is expressed in form of questions to convey

frustration for own errors, hostile attitude towards the interlocutor or strong dislike towards a code change. This is consistent with categorisation of anger towards *self*, *others*, and *object* described by previous research on negative emotion in issue tracking comments (GACHECHILADZE et al., 2017). This confirms previous evidence that developers express emotions during daily programming tasks (MURGIA et al., 2014).

Overall, our findings suggest that questions in code reviews do not seek exclusively to obtain information, *i.e.*, developers use questions to serve a wide variety of communication goals. In the following, we attempt to suggest avenues for possible implications for practitioners as well as further research.

5.5.1 Implications for Practitioners

In line with the recommendations for good practices provided by Efstathiou and Spinellis (EFSTATHIOU; SPINELLIS, 2018), we advocate that tools should support the automatic analysis of fine-grained communicative intentions in code review comments. For example, understanding the information needs of reviewers and the way authors and reviewers interact during code review can provide empirically-driven guidelines for change authors to anticipate the reviewers information needs, thus making the code review process faster and more effective. We believe that, as a consequence of understanding the communicative intention of the questions in code reviews, developers will have a better idea as to what other developers really want to say. Such practices can be complemented by the use of automatic tools providing feedback for improvement during code review. For example, we envision tools for augmented writing of comments to support neurodiverse developers to achieve an efficient and explicit communication during code review.

Furthermore, fine-grained analysis of the argumentation structure of code review discussions can be leveraged to identify long-lasting value knowledge to be shared among the contributors of a project. For example, the correct identification of requests for rationale may help identify design-related discussion, as proposed by Viviani *et al.* (VIVIANI et al., 2018a).

This study also provides further evidence of expressions of negative emotions in collaborative development (GACHECHILADZE et al., 2017; FORD; PARNIN, 2015). Early detection of negative emotions might benefit community management and reduce undesired turnover, *i.e.*, by preventing burnout and loss of productivity (MÄNTYLÄ et al., 2016) or timely addressing code of conduct violations and community smells (*i.e.*, sub-optimal organisational and socio-technical patterns in the organisational structure of the software community) (TOURANI; ADAMS; SEREBRENIK, 2017).

Finally, real-time identification of the communicative goal of questions in code review could enhance the effectiveness of the code review process itself. Requests for clarification or rationale could be detected in order to support programmers experiencing confusion, *e.g.*, by soliciting the author of the change to provide the required information (EBERT

et al., 2017) or by triggering an expert intervention. Conversely, requests for action or suggestions might be leveraged to notify the colleagues, *i.e.*, by notifying them that a change in the code is recommended.

5.5.2 Implication for Researchers

As befitting from the exploratory case study (RUNESON; HÖST, 2009), we propose three hypotheses about the intention of developers' questions in code reviews. Investigating those hypotheses should be a subject of a follow-up study.

H1 (cf. Section 5.4.2.1). *Questions from the inline comments are frequently used to trigger an action of the interlocutor (rather than to satisfy information needs).* Suggestions and requests for action jointly form the second most frequent communication goal: elicit an action of peers. We would like to investigate whether this same trend occurs in the general comments. Furthermore, this evidence indicates the intention of adopting a polite style of communication by making requests to perform code changes through indirect questions, rather than direct imperative statements. It would be interesting to enhance the evidence provided by related research about politeness and productivity in bug fixing (ORTU et al., 2015) by investigating this relationship also in code review.

H2 (cf. Section 5.4.2.2). *While the largest group of questions from the code reviews aim at satisfying information needs, such as request for information, rationale, or clarification, this group constitutes less than half of all questions.* One could further investigate to what extent the various types of requests are related to the presence of confusion in code reviews (EBERT et al., 2017).

H3 (cf. Section 5.4.2.3). *Developers use questions to express their own cognitive and emotional state in order to induce critical reflection in the interlocutor.* They express criticism and emotions (*e.g.*, anger or surprise), and convey doubt, perplexity and disagreement with different degrees of strength.

The questions' intentions classification can help researchers to understand how communication happens in code reviews. Our findings indicate that interaction during code reviews might serve several communication goals, beyond simply finding defects in the code. Beyond the specific hypotheses, a replication of this study with a broader set of questions, general comments and, possibly, with an automatic approach to speech act analysis, will bring full understanding of the role played by questions in code reviews. We believe that such understanding will shed new light on the informational needs of developers, the antecedents to confusion they might experience, and the way they communicate with each other to make suggestions or requests.

Furthermore, questions expressing different communicative intentions might have different impact on software quality, development time or project evolution: *e.g.*, hypothetical scenarios might be successful in revealing hidden bugs. Yet another perspective would be to combine the study of communicative intentions with social aspects of developers' com-

munities, *e.g.*, who is asking questions with different communicative intentions and who is being asked. One might also wonder whether the observation of Gachechilade *et al.* (that anger in Jira issues is usually directed at objects rather than the speaker themselves or the interlocutors (GACHECHILADZE et al., 2017)) is also valid for the code reviews. Finally, we would like to verify the relation between directionality of emotions and the outcome of code reviews (merge or abandon).

5.6 THREATS TO VALIDITY

In this section, we discuss the threats to validity following the guidelines of Runeson *et al.* (RUNESON et al., 2012), with focus on construct, internal, and external validity.

Construct validity. The threats to construct validity concern the relation between the theory and the observation. We used a state-of-the-art NLP tool, *i.e.*, the StanfordNLP API (MANNING et al., 2014), to extract questions included in our sample set. During the manual annotation, we observed that two sentences were erroneously classified as questions (0.4%) by the tool. Thus, we still consider its results precise enough for our exploratory study we conduct.

Internal validity. The threats to internal validity concern external factors we did not consider that could affect the variables and the relations being investigated. The manual annotation is an error-prone activity whose output depends on the subjective evaluation of the raters. To mitigate such threats, the raters were recruited based on their background in computer science and expertise with labeling. We solved disagreements either through majority voting or discussions, to address threats due to subjectivity in annotation.

External validity. The threats to external validity are related to the generalisability of the study results. Due to the exploratory character of our study, we do not claim generalisability of our observations, but use them to derive hypotheses (Section 5.5.2) to be confirmed or refuted in a follow-up study.

5.7 RELATED WORK

In the following, we report the related work on the study of communicative intentions of questions from different perspectives. We start discussing the intention of questions from the general dialogues perspective (Section 5.7.1), and then from the point of view of software developers (Section 5.7.2).

5.7.1 Analysis of Communicative Intentions in Dialogues

Speech acts are well studied in linguistics and computational linguistics research since long (AUSTIN, 1962; SEARLE, 1969; KEARSLEY, 1976). In computational linguistics, the task of automatic speech act recognition has been addressed with good results leveraging both supervised (STOLCKE et al., 2000; VOSOUGHI; ROY, 2016b) and unsupervised

approaches (NOVIELLI; STRAPPARAVA, 2011). This interest is justified by the large number of applications that could benefit from automatic speech act annotation of natural language interactions. For instance, simulation of natural dialogues with embodied conversational agents (KLÜWER, 2011), conversational interfaces for smart devices (MCTEAR; CALLEJAS; GRIOL, 2016) and the Internet of Things (KAR; HALDAR, 2016), detection of rumors in microblogging (VOSOUGHI; ROY, 2016a), and identification of roles in group chats in computer-mediated decision-making tasks (BARLOW, 2013).

Specifically, the communicative intentions conveyed by questions have been object of dedicated studies. Stivers and Enfield (STIVERS; ENFIELD, 2010) defined a coding scheme for questions and responses from ordinary conversations. Their intention was to publish the step-by-step methodology for coding questions and responses.

Ilie (ILIE, 1999) proposed a pragmatic framework for the interpretation of communicative intentions in questions occurring in talk shows. The study showed how mixed types of questions are often combined to fulfill several communication goals simultaneously. In particular, the author studied three types of argumentative questions, namely *expository questions*, *rhetorical questions*, and *echo questions*, and how they are used to elicit reactions from the interlocutor or from the audience. The three categories of questions have been analysed and categorised based on how they serve specific communicative goals in different argumentation framework, *i.e.*, to elicit an argument from the interlocutor (*argument-eliciting* questions), to preface the speaker's own argument to the audience (*argument-prefacing* questions), and to provide argument in the form of rhetorical questions (*argument-supplying* questions).

Chen *et al.* (CHEN; ZHANG; MARK, 2012) proposed a semi-supervised approach for classification of the user intent in community-based Question Answering (Q&A). They built a predictive model through machine learning based on both text and meta-data features for classification of requests on YAHOO! ANSWERS[3]. Questions are classified into three categories accordingly to the user intent, namely *subjective*, *objective*, and *social*, consistently with their long-term goal of enhancing information retrieval in Q&A sites.

5.7.2 Analysis of Developers' Questions

Thanks to the popularity of STACKOVERFLOW, which has made available an enormous amount of natural language interactions, researchers in software engineering also started to investigate how developers formulate their requests in information seeking tasks. Treude *et al.* (TREUDE; BARZILAY; STOREY, 2011) analysed which kinds of questions are asked on STACKOVERFLOW. By conducting a qualitative coding of questions asked by developers on STACKOVERFLOW, they categorised the questions, such as *how-to*, *discrepancy*, *environment*, *error*, and *review* (*i.e.*, questions that ask for a code review). The authors also verified which questions are answered well and which ones remain unanswered. Their

[3] http://answer.yahoo.com

preliminary findings indicate that StackOverflow is particularly successful in replying to *how-to* questions posed by new community members.

Bajaj *et al.* (BAJAJ; PATTABIRAMAN; MESBAH, 2014) also investigated the questions and answers from StackOverflow to understand the challenges and misconceptions among web developers. They used Latent Dirichlet Allocation (LDA) (BLEI; NG; JORDAN, 2003) to categorise the discussions from developers. As of the main results, they found that Cross Browser discussions are the most popular among web developers, followed by DOM and Canvas related discussions.

Calefato *et al.* (CALEFATO; LANUBILE; NOVIELLI, 2018b) investigated how information seekers can increase the chance of eliciting a successful answer to their requests for technical help on StackOverflow. Specifically, they investigate the impact of actionable factors, namely *affect*, *presentation quality*, and *time*, on the success of questions. Based on their findings, they provided evidence-based guidelines that information seekers can follow to increase the chance of getting help, *e.g.*, add example code snippets to enhance clarity, be concise, avoid unnecessary use of uppercase characters, and adopt a neutral writing style.

While relevant for the software development domain, the above mentioned studies focused either on the topics or on the writing style of the questions from Q&A websites, where the intention of questions is assumed to be the same for all StackOverflow posts, *i.e.*, asking for technical help (TREUDE; BARZILAY; STOREY, 2011).

Other studies aimed at categorising the type of information need expressed by developers' questions. It is the case of Fritz and Murphy (FRITZ; MURPHY, 2010) who conducted interviews with eleven developers and identified 78 questions they ask in their daily jobs. They classified the information needs expressed by developers into several categories, such as code change specific, people specific, and work-item progress. Furthermore, the channels where such information could be found include the source code, change sets, teams, work items, and informal comments.

Sillito *et al.* (SILLITO; MURPHY; VOLDER, 2006) conducted a study with developers performing code changes to understand what information they need to know about a code base and how they find it. They identified 44 different questions developers ask. Those questions are categorised into four groups based on the characteristics of the source code graph capturing the information needed for answering a given question: those aimed at finding initial focus points, those aimed at building on such points, those aimed at understanding a subgraph, and those over such subgraphs.

5.8 SUMMARY

In this chapter, we presented the study still related to the problem of *lack of knowledge* in code reviews, but with a slight different point of view: from the intention of developers questions. We believe this is an important topic which is strong related to confusion, the

analysis of this relation is part of our future work. We conducted a case study to investigate the communicative intention of developers' questions during the code review process. We observed that questions are more present in the inline comments than general ones. We also manually labelled 499 questions from 399 ANDROID code review comments. We found that, while representing the majority of requests, information seeking questions are still less than half of all questions in our sample. This evidence suggests that questions are actually used by developers in code review to serve a wider variety of communicative purposes. Specifically, we found that questions are extensively used by developers to convey suggestions. Developers also express their cognitive and affective states in code review comments, such as attitude of doubts and criticisms or emotions like anger and surprise. Based on this case study, we formulate three hypotheses about the intention of developers' questions in the code review process.

6 CONCLUSIONS

In this chapter, we review the problem stated, the solution proposed and the main contributions of this thesis. We also discuss how we plan to extend this study in the future work.

6.1 THE PROBLEM

This thesis tackles two important and timely problems:

1. The *lack of knowledge* about confusion faced by developers in code reviews;
2. The *lack of tools* for confusion identification in code review comments.

We conducted three different studies aiming the problem of *lack of knowledge*. We believe we provided a in-depth understanding of this problem, *i.e.*, how confusion is related to code reviews. As for the problem of *lack of tools*, we believe we provided the first step towards its solution. Three classifiers with a reasonable performance are available for practitioners and researchers to use it to identify confusion in code reviews comments.

6.2 REVIEW OF MAIN CONTRIBUTIONS

This thesis offers the following contributions:

1. A *confusion coding scheme*, together with an automated approach for detecting confusion in code review comments;
2. A gold standard set with 1,542 general and 1,190 inline code review comments labelled as *confusion* and *no confusion*;
3. A model of *confusion in context*, with the reasons and impacts of confusion in code reviews, as well as the strategies developers adopt to cope with confusion;
4. A gold standard set of our *confusion in context* model, comprising the reasons, impacts, and coping strategies related to *confusion* in code reviews;
5. A series of practical and actionable suggestions to improve code review tools based on our *confusion in context* model;
6. A classification of the communicative intentions expressed in the developers' questions in code reviews;
7. A gold standard set of 499 questions from code reviews with corresponding communicative intentions.

6.3 FUTURE WORK

We intend to extend this study with the following future work.

As for the **confusion detection** study (cf. Chapter 3):

- Firstly, we want to analyse the inline comments of the **remaining** group;
- We think of experimenting other scenarios within our models, *e.g.*, assessing the impact of the handling of negation and the undersampling (DRUMMOND; HOLTE, 2003) of the majority class (*i.e.*, sampling a smaller set of the *no confusion* class);
- We consider experimenting different NLP libraries, other than StanfordNLP, to the identification of questions since we observed a small number of questions not detected by it;
- The error analysis of the balanced models is another step we plan to conduct in order to understand the reason of the misclassification within those models;
- Finally, we plan to replicate this study with other projects to assess the evaluation of our models of confusion identification, as well as to increase our gold standard sets of confusion comments.

As regarding the **confusion in context** study (cf. Chapter 4):

- Firstly, we want to investigate *how* and *when* confusion affects the relations between code review metrics (*e.g.*, review time, number of comments, code change size, number of reviewers, etc.) and the code review outcome (*e.g.*, accepted or rejected). The former is established by the *mediation* analysis, while the latter is with the *moderation* analysis (HAYES, 2013);
- We intend to automate detection of different *confusion* dimensions (*e.g.*, from the review process, artifact, and developers) in code review comments by using machine learning techniques;
- We plan to study possible relations between *confusion* and information needs (PASCARELLA et al., 2018), as well as between *confusion* and migration of code review discussions to off-line channels;
- In our model, several reasons for confusion do not have a coping strategy related. Thus, we aim at investigating how those reasons could be addressed in order to support developers in code reviews;
- We will provide a series of guidelines on how to avoid and deal with confusion during code reviews based on our *confusion in context* model;

- Lastly, we consider verifying whether the perceived impact of *confusion* can be observed in the code review artifacts.

As for the **communicative intentions of questions** study (cf. Chapter 5):

- We consider confirming/rejecting the hypotheses first in the general comments of Android, and next on other OSS projects which conduct code reviews;
- We plan to study the communicative intention of the code review comment as a whole, as the combination of different sentence intentions in the same comment might have another intention;
- We intend to survey developers to validate our findings about their communicative intentions in code reviews;
- Finally, we plan to investigate the relations between the questions' intentions and confusion in code review comments.

REFERENCES

AHMED, T.; BOSU, A.; IQBAL, A.; RAHIMI, S. SentiCR: A customized sentiment analysis tool for code review interactions. In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2017. (ASE 2017), p. 106–111. ISBN 978-1-5386-2684-9. Available at: <http://dl.acm.org/citation.cfm?id=3155562.3155579>.

ALBRIGHT, L.; COHEN, A. I.; MALLOY, T. E.; CHRIST, T.; BROMGARD, G. Judgments of communicative intent in conversation. *Journal of Experimental Social Psychology*, v. 40, n. 3, p. 290–302, 2004.

ANDERSON, M. J. A new method for non-parametric multivariate analysis of variance. *Austral Ecology*, v. 26, n. 1, p. 32–46, 2001. Available at: <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1442-9993.2001.01070.pp.x>.

ARANDA, J.; VENOLIA, G. The secret life of bugs: Going past the errors and omissions in software repositories. In: *Proceedings of the 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009. (ICSE '09), p. 298–308. ISBN 978-1-4244-3453-4. Available at: <http://dx.doi.org/10.1109/ICSE.2009.5070530>.

ARIMA, R.; HIGO, Y.; KUSUMOTO, S. A study on inappropriately partitioned commits: How much and what kinds of ip commits in java projects? In: *Proceedings of the 15th International Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2018. p. 336–340. ISBN 978-1-4503-5716-6. Available at: <http://doi.acm.org/10.1145/3196398.3196406>.

ARMOUR, P. G. The five orders of ignorance. *Commun. ACM*, ACM, New York, NY, USA, v. 43, n. 10, p. 17–20, Oct. 2000. ISSN 0001-0782. Available at: <http://doi.acm.org/10.1145/352183.352194>.

ASUNDI, J.; JAYANT, R. Patch review processes in open source software development communities: A comparative case study. In: *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*. Washington, DC, USA: IEEE Computer Society, 2007. (HICSS '07), p. 166c–. ISBN 0-7695-2755-8. Available at: <https://doi.org/10.1109/HICSS.2007.426>.

AUSTIN, J. L. *How to do things with words*. New York, USA: Oxford University Press, 1962. (William James Lectures).

BACCHELLI, A.; BIRD, C. Expectations, outcomes, and challenges of modern code review. In: *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 712–721. ISBN 978-1-4673-3076-3. Available at: <http://dl.acm.org/citation.cfm?id=2486788.2486882>.

BAJAJ, K.; PATTABIRAMAN, K.; MESBAH, A. Mining questions asked by web developers. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2014. (MSR 2014), p. 112–121. ISBN 978-1-4503-2863-0. Available at: <http://doi.acm.org/10.1145/2597073.2597083>.

BAKER JR., R. A. Code reviews enhance software quality. In: *Proceedings of the 19th International Conference on Software Engineering*. New York, NY, USA: ACM, 1997. (ICSE '97), p. 570–571. ISBN 0-89791-914-9. Available at: <http://doi.acm.org/10.1145/253228.253461>.

BAKER, R. S. J. de; GOWDA, S. M.; WIXON, M.; KALKA, J.; WAGNER, A. Z.; SALVI, A.; ALEVEN, V.; KUSBIT, G.; OCUMPAUGH, J.; ROSSI, L. M. Sensor-free automated detection of affect in a cognitive tutor for algebra. In: *Proceedings of the 5th International Conference on Educational Data Mining*. Chania, Greece: www.educationaldatamining.org, 2012.

BARLOW, J. B. Emergent roles in decision-making tasks using group chat. In: *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*. New York, NY, USA: ACM, 2013. (CSCW '13), p. 1505–1514. ISBN 978-1-4503-1331-5. Available at: <http://doi.acm.org/10.1145/2441776.2441948>.

BARNETT, M.; BIRD, C.; BRUNET, J. A.; LAHIRI, S. K. Helping developers help themselves: Automatic decomposition of code review changesets. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. Piscataway, NJ, USA: IEEE Press, 2015. (ICSE '15), p. 134–144. ISBN 978-1-4799-1934-5. Available at: <http://dl.acm.org/citation.cfm?id=2818754.2818773>.

BAVOTA, G.; RUSSO, B. 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)*. Bremen, Germany: IEEE Computer Society, 2015. p. 81–90.

BERGSTRA, J.; BENGIO, Y. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, JMLR.org, v. 13, p. 281–305, Feb. 2012. ISSN 1532-4435. Available at: <http://dl.acm.org/citation.cfm?id=2188385.2188395>.

BERNHART, M.; MAUCZKA, A.; GRECHENIG, T. Adopting code reviews for agile software development. In: *2010 Agile Conference*. Orlando, FL, USA: IEEE Computer Society, 2010. p. 44–47.

BLEI, D. M.; NG, A. Y.; JORDAN, M. I. Latent dirichlet allocation. *J. Mach. Learn. Res.*, JMLR.org, v. 3, p. 993–1022, mar 2003. ISSN 1532-4435. Available at: <http://dl.acm.org/citation.cfm?id=944919.944937>.

BOEHM, B.; BASILI, V. R. Top 10 list [software development]. *Computer*, v. 34, n. 1, p. 135–137, Jan 2001. ISSN 0018-9162.

BOSU, A.; CARVER, J. C.; BIRD, C.; ORBECK, J.; CHOCKLEY, C. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. *IEEE Transactions on Software Engineering*, v. 43, n. 1, p. 56–75, Jan 2017. ISSN 0098-5589.

BOSU, A.; GREILER, M.; BIRD, C. Characteristics of useful code reviews: An empirical study at microsoft. In: *Proceedings of the 12th Working Conference on Mining Software Repositories*. Piscataway, NJ, USA: IEEE Press, 2015. (MSR '15), p. 146–156. ISBN 978-0-7695-5594-2. Available at: <http://dl.acm.org/citation.cfm?id=2820518.2820538>.

BUCHAN, J.; PEARL, M. Leveraging the mob mentality: An experience report on mob programming. In: *Proceedings of the 22Nd International Conference on Evaluation and Assessment in Software Engineering 2018*. New York, NY, USA: ACM, 2018. (EASE'18), p. 199–204. ISBN 978-1-4503-6403-4. Available at: <http://doi.acm.org/10.1145/3210459.3210482>.

CALEFATO, F.; LANUBILE, F.; MAIORANO, F.; NOVIELLI, N. Sentiment polarity detection for software development. *Empirical Software Engineering*, Kluwer Academic Publishers, Norwell, MA, USA, v. 23, n. 3, p. 1352–1382, Jun. 2018. ISSN 1382-3256. Available at: <https://doi.org/10.1007/s10664-017-9546-9>.

CALEFATO, F.; LANUBILE, F.; NOVIELLI, N. An empirical assessment of best-answer prediction models in technical Q&A sites. *Empirical Software Engineering*, 08 2018.

CALEFATO, F.; LANUBILE, F.; NOVIELLI, N. How to ask for technical help? evidence-based guidelines for writing questions on stack overflow. *Information and Software Technology*, v. 94, p. 186 – 207, 2018. ISSN 0950-5849. Available at: <http://www.sciencedirect.com/science/article/pii/S0950584917301167>.

CAMELO, M. *Além Do Que Se Vê*. 2003. Ventura.

CHAHBOUN, S.; VULCHANOV, V.; SALDAÑA, D.; ESHUIS, H.; VULCHANOVA, M. Can you play with fire and not hurt yourself? a comparative study in figurative language comprehension between individuals with and without autism spectrum disorder. *PLOS ONE*, Public Library of Science, v. 11, n. 12, p. 1–24, 12 2016. Available at: <https://doi.org/10.1371/journal.pone.0168571>.

CHAWLA, N. V.; BOWYER, K. W.; HALL, L. O.; KEGELMEYER, W. P. SMOTE: synthetic minority over-sampling technique. *Journal Of Artificial Intelligence Research*, abs/1106.1813, p. 321–357, 2002.

CHEN, L.; ZHANG, D.; MARK, L. Understanding user intent in community question answering. In: *Proceedings of the 21st International Conference on World Wide Web*. New York, NY, USA: ACM, 2012. (WWW '12 Companion), p. 823–828. ISBN 978-1-4503-1230-1. Available at: <http://doi.acm.org/10.1145/2187980.2188206>.

CLARKE, K. R. Non-parametric multivariate analysis of changes in community structure. *Australian Journal of Ecology*, v. 18, p. 117–143, 1993.

COHEN, J.; TELEKI, S.; BROWN, E. *Best Kept Secrets of Peer Code Review*. Massachusetts, EUA: Smart Bear Inc., 2006.

COHEN, P.; LEVESQUE, H. J. *Rational interaction as the basis for communication*. Massachusetts , USA: MIT Press, 1990. 221–255 p.

CORE, M. G.; ALLEN, J. F. Coding dialogs with the DAMSL annotation scheme. In: *Working Notes of the AAAI Fall Symposium on Communicative Action in Humans and Machines*. Cambridge, MA: AAAI Fall Symposium on Communicative Action in Humans and Machines, 1997. p. 28–35. Available at: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.7024>.

CROOK, J. On covert communication in advertising. *Journal of Pragmatics*, v. 36, n. 4, p. 715–738, 2004. ISSN 0378-2166.

DANESCU-NICULESCU-MIZIL, C.; SUDHOF, M.; JURAFSKY, D.; LESKOVEC, J.; POTTS, C. A computational approach to politeness with application to social factors. *CoRR*, abs/1306.6078, 2013. Available at: <http://arxiv.org/abs/1306.6078>.

DAS, S.; CHEN, M. Yahoo! for amazon: extracting market sentiment from stock message boards. *Manage. Sci.*, v. 539, 01 2001.

DIAS, M.; BACCHELLI, A.; GOUSIOS, G.; CASSOU, D.; DUCASSE, S. Untangling fine-grained code changes. In: GUÉHÉNEUC, Y.; ADAMS, B.; SEREBRENIK, A. (Ed.). *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*. Montreal, QC, Canada: IEEE Computer Society, 2015. p. 341–350.

D'MELLO, S.; CRAIG, S.; JOHNSON, A.; MCDANIEL, B.; GRAESSER, A. Automatic detection of learner's affect from conversational cues. *User Modeling and User-Adapted Interaction*, Springer Netherlands, v. 18, n. 1-2, p. 45–80, 2 2008. ISSN 0924-1868.

D'MELLO, S.; GRAESSER, A. Confusion and its dynamics during device comprehension with breakdown scenarios. *Acta Psychologica*, v. 151, p. 106–116, 2014. ISSN 0001-6918. Available at: <http://www.sciencedirect.com/science/article/pii/S0001691814001504>.

D'MELLO, S.; LEHMAN, B.; PEKRUN, R.; GRAESSER, A. Confusion can be beneficial for learning. *Learning and Instruction*, v. 29, p. 153 – 170, 2014. ISSN 0959-4752.

DRUMMOND, C.; HOLTE, R. C. C4.5, class imbalance, and cost sensitivity: Why under-sampling beats over-sampling. In: *Working Notes ICML Workshop Learning Imbalanced Data Sets*. Washington DC, USA: AAAI Press, 2003. p. 1–8.

DRUMMOND, C.; HOLTE, R. C. Cost curves: An improved method for visualizing classifier performance. *Machine Learning*, v. 65, n. 1, p. 95–130, Oct 2006. ISSN 1573-0565. Available at: <https://doi.org/10.1007/s10994-006-8199-5>.

EASTERBROOK, S.; SINGER, J.; STOREY, M.-A.; DAMIAN, D. Selecting empirical methods for software engineering research. In: _____. *Guide to Advanced Empirical Software Engineering*. London: Springer London, 2008. p. 285–311. ISBN 978-1-84800-044-5. Available at: <https://doi.org/10.1007/978-1-84800-044-5_11>.

EBERT, F. *Material of the PhD thesis "Understanding Confusion in Code Reviews"*. 2019. Permanent link: <https://github.com/felipeebert/confusion-in-code-reviews>. Available at: <https://doi.org/10.5281/zenodo.2541203>.

EBERT, F.; CASTOR, F.; NOVIELLI, N.; SEREBRENIK, A. Confusion detection in code reviews. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Shanghai, China: IEEE Computer Society, 2017. p. 549–553.

EBERT, F.; CASTOR, F.; NOVIELLI, N.; SEREBRENIK, A. Communicative intention in code review questions. In: *The 34th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Madrid, Spain: IEEE Computer Society, 2018. p. 519–523. ISSN 2576-3148.

EBERT, F.; CASTOR, F.; NOVIELLI, N.; SEREBRENIK, A. Confusion in code reviews: Reasons, impacts and coping strategies. In: *The 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'2019)*. Hangzhou, China: IEEE Computer Society, 2019.

EBERT, F.; CASTOR, F.; SEREBRENIK, A. An exploratory study on exception handling bugs in java programs. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 106, n. C, p. 82–101, aug 2015. ISSN 0164-1212. Available at: <http://dx.doi.org/10.1016/j.jss.2015.04.066>.

EFSTATHIOU, V.; SPINELLIS, D. Code review comments: Language matters. In: *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. New York, NY, USA: ACM, 2018. (ICSE-NIER '18), p. 69–72. ISBN 978-1-4503-5662-6. Available at: <http://doi.acm.org/10.1145/3183399.3183411>.

ESTABROOKS, A.; JO, T.; JAPKOWICZ, N. A multiple resampling method for learning from imbalanced data sets. *Computational Intelligence*, v. 20, n. 1, p. 18–36, 2004. Available at: <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.0824-7935.2004.t01-1-00228.x>.

FAGAN, M. E. Design and code inspections to reduce errors in program development. *IBM Syst. J.*, IBM Corp., Riverton, NJ, USA, v. 15, n. 3, p. 182–211, Sep. 1976. ISSN 0018-8670. Available at: <http://dx.doi.org/10.1147/sj.153.0182>.

FINFGELD-CONNETT, D. Use of content analysis to conduct knowledge-building and theory-generating qualitative systematic reviews. *Qualitative Research*, v. 14, n. 3, p. 341–352, 2014. Available at: <https://doi.org/10.1177/1468794113481790>.

FLEISS, J. Measuring nominal scale agreement among many raters. *Psychological Bulletin*, v. 76, p. 378–382, 1971.

FLYVBJERG, B. *Five Misunderstandings about Case-Study Research*. California, USA: Sage, 2007. 390–404 p.

FODDY, W. H. *Constructing questions for interviews and questionnaires: theory and practice in social research*. New York, NY, USA: Cambridge University Press Cambridge, 1993. ISBN 0521467330 0521420091.

FORD, D.; PARNIN, C. Exploring causes of frustration for software developers. In: *Proceedings of the Eighth International Workshop on Cooperative and Human Aspects of Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2015. (CHASE '15), p. 115–116. Available at: <http://dl.acm.org/citation.cfm?id=2819321.2819346>.

FORMAN, G. An extensive empirical study of feature selection metrics for text classification. *The Journal of Machine Learning Research*, JMLR.org, v. 3, p. 1289–1305, Mar. 2003. ISSN 1532-4435. Available at: <http://dl.acm.org/citation.cfm?id=944919.944974>.

FORMAN, G.; COHEN, I. Learning from little: Comparison of classifiers given little training. In: BOULICAUT, J.-F.; ESPOSITO, F.; GIANNOTTI, F.; PEDRESCHI, D. (Ed.). *Knowledge Discovery in Databases: PKDD 2004*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 161–172.

FRANK, E.; HALL, M. A.; WITTEN, I. H. *The WEKA Workbench*. Morgan Kaufmann, 2016. Appendix for "Data Mining: Practical Machine Learning Tools and Techniques". Available at: <http://www.cs.waikato.ac.nz/ml/weka/Witten_et_al_2016_appendix.pdf>.

FRITZ, T.; MURPHY, G. C. Using information fragments to answer the questions developers ask. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. New York, NY, USA: ACM, 2010. (ICSE '10), p. 175–184. ISBN 978-1-60558-719-6. Available at: <http://doi.acm.org/10.1145/1806799.1806828>.

FUSSELL, S. R.; KREUZ, R. J. *Social and Cognitive Approaches to Interpersonal Communication: Introduction and overviem*. Mahwah, New Jersey, USA: Lawrence Erlbaum Associates, 1998. 3–17 p.

GACHECHILADZE, D.; LANUBILE, F.; NOVIELLI, N.; SEREBRENIK, A. Anger and its direction in collaborative software development. In: *Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Results Track*. Piscataway, NJ, USA: IEEE Press, 2017. (ICSE-NIER '17), p. 11–14. ISBN 978-1-5386-2675-7. Available at: <https://doi.org/10.1109/ICSE-NIER.2017.18>.

GELUYKENS, R. *The Pragmatics of Discourse Anaphora in English: Evidence from Conversational Repair*. New York, USA: Walter de Gruyter, Berlin, 1994.

GHOTRA, B.; MCINTOSH, S.; HASSAN, A. E. Revisiting the impact of classification techniques on the performance of defect prediction models. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. Piscataway, NJ, USA: IEEE Press, 2015. (ICSE '15), p. 789–800. ISBN 978-1-4799-1934-5. Available at: <http://dl.acm.org/citation.cfm?id=2818754.2818850>.

GLASER, B. G.; STRAUSS, A. L. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. New York, NY: Aldine de Gruyter, 1967.

GOPSTEIN, D.; IANNACONE, J.; YAN, Y.; DELONG, L.; ZHUANG, Y.; YEH, M. K.-C.; CAPPOS, J. Understanding misunderstandings in source code. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: ACM, 2017. (ESEC/FSE 2017), p. 129–139. ISBN 978-1-4503-5105-8. Available at: <http://doi.acm.org/10.1145/3106237.3106264>.

GREILER, M. *On to code review: Lessons learned @ Microsoft*. 2016. Keynote for QUATIC 2016 - the 10th International Conference on the Quality of Information and Communication Technology. Available at: <https://pt.slideshare.net/mgreiler/on-to-code-review-lessons-learned-at-microsoft>.

GROVES, R. M.; FOWLER, F. J.; COUPER, M. P.; LEPKOWSKI, J. M.; SINGER, E.; TOURANGEAU, R. *Survey Methodology*. 2nd. ed. New Jersey, USA: Wiley, 2009.

GUYON, I.; ELISSEEFF, A. An introduction to variable and feature selection. *The Journal of Machine Learning Research*, JMLR.org, v. 3, p. 1157–1182, Mar. 2003. ISSN 1532-4435. Available at: <http://dl.acm.org/citation.cfm?id=944919.944968>.

GUZZI, A.; BACCHELLI, A.; LANZA, M.; PINZGER, M.; DEURSEN, A. van. Communication in open source software development mailing lists. In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. San Francisco, Californi, USA: IEEE Computer Society, 2013. p. 277–286. ISSN 2160-1860.

HALL, T.; BEECHAM, S.; BOWES, D.; GRAY, D.; COUNSELL, S. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, v. 38, n. 6, p. 1276–1304, Nov 2012. ISSN 0098-5589.

HAMASAKI, K.; KULA, R. G.; YOSHIDA, N.; CRUZ, A. E. C.; FUJIWARA, K.; IIDA, H. Who does what during a code review? datasets of oss peer review repositories. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. Piscataway, NJ, USA: IEEE Press, 2013. (MSR '13), p. 49–52. ISBN 978-1-4673-2936-1. Available at: <http://dl.acm.org/citation.cfm?id=2487085.2487096>.

HAUGH, M. On understandings of intention: A response to Wedgwood. *Intercultural Pragmatics*, v. 9, n. 2, p. 161–194, 2012. ISSN 1613-365X.

HAYES, A. F. *Introduction to Mediation, Moderation, and Conditional Process Analysis: A Regression-Based Approach*. New York, NY, USA: Guilford Press, 2013. ISBN 9781609182304.

HE, H.; GARCIA, E. A. Learning from imbalanced data. *IEEE Trans. on Knowl. and Data Eng.*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 21, n. 9, p. 1263–1284, sep 2009. ISSN 1041-4347.

HENTSCHEL, M.; HÄHNLE, R.; BUBEL, R. Can formal methods improve the efficiency of code reviews? In: *Proceedings of the 12th International Conference on Integrated Formal Methods - Volume 9681*. Berlin, Heidelberg: Springer-Verlag, 2016. (IFM 2016), p. 3–19. ISBN 978-3-319-33692-3. Available at: <https://doi.org/10.1007/978-3-319-33693-0_1>.

HOLMES, J. Expressing doubt and certainty in english. *RELC Journal*, v. 13, n. 2, p. 9–28, 1982. Available at: <https://doi.org/10.1177/003368282013002O2>.

HOLTGRAVES, T. M. *Language as social action: Social psychology and language use*. New Jersey, USA: Lawrence Erlbaum Associates Publishers, 2002.

HOSMER, D. W.; LEMESHOW, S. Book; Book/Illustrated. *Applied logistic regression*. 2nd ed. ed. New York : Wiley, 2000. "A Wiley-Interscience publication.". ISBN 0471356328 (cloth : alk. paper). Available at: <http://gateway.library.qut.edu.au/login?url=https://onlinelibrary.wiley.com/doi/book/10.1002/0471722146>.

HUANG, Y.; ZHENG, Q.; CHEN, X.; XIONG, Y.; LIU, Z.; LUO, X. Mining version control system for automatically generating commit comment. In: *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. Piscataway, NJ, USA: IEEE Press, 2017. p. 414–423. ISBN 978-1-5090-4039-1. Available at: <https://doi.org/10.1109/ESEM.2017.56>.

ILIE, C. Question-response argumentation in talk shows. *Journal of Pragmatics*, v. 31, n. 8, p. 975 – 999, 1999. ISSN 0378-2166. Available at: <http://www.sciencedirect.com/science/article/pii/S0378216699000569>.

JEAN, P.-A.; HARISPE, S.; RANWEZ, S.; BELLOT, P.; MONTMAIN, J. Uncertainty detection in natural language: A probabilistic model. In: *International Conference on Web Intelligence, Mining and Semantics*. New York, NY, USA: ACM, 2016. p. 10:1–10:10. ISBN 978-1-4503-4056-4.

JIANG, Y.; CUKIC, B.; MENZIES, T. Can data transformation help in the detection of fault-prone modules? In: *Proceedings of the 2008 Workshop on Defects in Large Software Systems*. New York, NY, USA: ACM, 2008. (DEFECTS '08), p. 16–20. ISBN 978-1-60558-051-7. Available at: <http://doi.acm.org/10.1145/1390817.1390822>.

JOACHIMS, T. Text categorization with support vector machines: Learning with many relevant features. In: NÉDELLEC, C.; ROUVEIROL, C. (Ed.). *Machine Learning: ECML-98*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998. p. 137–142.

JORDAN, M. E.; CHENG, A.-C. J.; SCHALLERT, D.; SONG, K.; LEE, S.; PARK, Y. "I guess my question is": What is the co-occurrence of uncertainty and learning in computer-mediated discourse? *International Journal of Computer-Supported Collaborative Learning*, v. 9, n. 4, p. 451–475, Dec 2014. ISSN 1556-1615. Available at: <https://doi.org/10.1007/s11412-014-9203-x>.

JORDAN, M. E.; JR., R. R. M. Managing uncertainty during collaborative problem solving in elementary school teams: The role of peer influence in robotics engineering activity. *Journal of the Learning Sciences*, Routledge, v. 23, n. 4, p. 490–536, 2014. Available at: <https://doi.org/10.1080/10508406.2014.896254>.

JORDAN, M. E.; SCHALLERT, D. L.; PARK, Y.; LEE, S.; CHIANG, Y. hui V.; CHENG, A.-C. J.; SONG, K.; CHU, H.-N. R.; KIM, T.; LEE, H. Expressing uncertainty in computer-mediated discourse: Language as a marker of intellectual work. *Discourse Processes*, Routledge, v. 49, n. 8, p. 660–692, 2012. Available at: <https://doi.org/10.1080/0163853X.2012.722851>.

KAR, R.; HALDAR, R. Applying chatbots to the internet of things: Opportunities and architectural elements. *CoRR*, abs/1611.03799, 2016. Available at: <http://arxiv.org/abs/1611.03799>.

KAREGOWDA, A. G.; M.A.JAYARAM; MANJUNATH, A. Feature subset selection problem using wrapper approach in supervised learning. *International Journal of Computer Applications*, Foundation of Computer Science, v. 1, n. 7, p. 13–17, February 2010. Published By Foundation of Computer Science.

KEARSLEY, G. P. Questions and question asking in verbal discourse: A cross-disciplinary review. *Journal of Psycholinguistic Research*, v. 5, n. 4, p. 355–375, Oct 1976. ISSN 1573-6555. Available at: <https://doi.org/10.1007/BF01079934>.

KITCHENHAM, B.; PFLEEGER, S. L. Personal opinion surveys. In: SHULL, F.; SINGER, J.; SJOBERG, D. I. K. (Ed.). *Guide to Advanced Empirical Software Engineering*. London, UK: Springer, London, 2008. p. 63–92.

KLÜWER, T. "i like your shirt" - dialogue acts for enabling social talk in conversational agents. In: VILHJÁLMSSON, H. H.; KOPP, S.; MARSELLA, S.; THÓRISSON, K. R. (Ed.). *Intelligent Virtual Agents*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 14–27. ISBN 978-3-642-23974-8.

KOHAVI, R.; JOHN, G. H. Wrappers for feature subset selection. *Artificial Intelligence*, v. 97, n. 1-2, p. 273–324, 1997. ISSN 0004-3702. Special issue on relevance.

KONONENKO, O.; BAYSAL, O.; GUERROUJ, L.; CAO, Y.; GODFREY, M. W. Investigating code review quality: Do people and participation matter? In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Bremen, Germany: IEEE Computer Society, 2015. p. 111–120.

KRAUSS, R. M.; APPLE, W.; MORENCY, N.; WENZEL, C.; WINTON, W. Verbal, vocal, and visible factors in judgments of another's affect. *Journal of Personality and Social Psychology*, v. 40, p. 312–320, 02 1981.

LAKOFF, G. Hedges: A study in meaning criteria and the logic of fuzzy concepts. In: _____. Dordrecht, The Netherlands: Springer, Dordrecht, 1975. p. 221–271.

LANDIS, J. R.; KOCH, G. G. The measurement of observer agreement for categorical data. *Biometrics*, International Biometric Society, v. 33, n. 1, 1977.

LATOZA, T. D.; VENOLIA, G.; DELINE, R. Maintaining mental models: A study of developer work habits. In: *Proceedings of the 28th International Conference on Software Engineering*. New York, NY, USA: ACM, 2006. (ICSE '06), p. 492–501. ISBN 1-59593-375-1. Available at: <http://doi.acm.org/10.1145/1134285.1134355>.

LEE, A.; CARVER, J. C.; BOSU, A. Understanding the impressions, motivations, and barriers of one time code contributors to FLOSS projects: a survey. In: UCHITEL, S.; ORSO, A.; ROBILLARD, M. P. (Ed.). *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017*. Buenos Aires, Argentina: IEEE / ACM, 2017. p. 187–197.

LENBERG, P.; FELDT, R.; TENGBERG, L. G. W.; TIDEFORS, I.; GRAZIOTIN, D. Behavioral software engineering - guidelines for qualitative studies. *CoRR*, abs/1712.08341, 2017. Available at: <http://arxiv.org/abs/1712.08341>.

LESSMANN, S.; BAESENS, B.; MUES, C.; PIETSCH, S. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, v. 34, n. 4, p. 485–496, July 2008. ISSN 0098-5589.

LIN, B.; ROBLES, G.; SEREBRENIK, A. Developer turnover in global, industrial open source projects: Insights from applying survival analysis. In: *Proceedings of the 12th International Conference on Global Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2017. (ICGSE '17), p. 66–75. ISBN 978-1-5386-1587-4. Available at: <https://doi.org/10.1109/ICGSE.2017.11>.

LINARES-VÁSQUEZ, M.; CORTÉS-COY, L. F.; APONTE, J.; POSHYVANYK, D. Changescribe: A tool for automatically generating commit messages. In: *Proceedings of the 37th International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2015. v. 2, p. 709–712.

LÓPEZ, V.; FERNáNDEZ, A.; GARCÍA, S.; PALADE, V.; HERRERA, F. An insight into classification with imbalanced data: Empirical results and current trends on using data intrinsic characteristics. *Information Sciences*, v. 250, p. 113 – 141, 2013. ISSN 0020-0255. Available at: <http://www.sciencedirect.com/science/article/pii/S0020025513005124>.

MANDLER, G. *Mind and Body: Psychology of Emotion and Stress*. New York, USA: W.W. Norton and Company, 1984.

MANDLER, G. Interruption (discrepancy) theory: review and extensions. In: _____. *On the move: The psychology of change and transition*. Chichester, UK: Wiley, 1990. p. 13–32.

MANNING, C. D.; SURDEANU, M.; BAUER, J.; FINKEL, J.; BETHARD, S. J.; MCCLOSKY, D. The Stanford CoreNLP natural language processing toolkit. In: *Association for Computational Linguistics (ACL) System Demonstrations*. Baltimore, Maryland: Association for Computational Linguistics, 2014. p. 55–60.

MÄNTYLÄ, M.; ADAMS, B.; DESTEFANIS, G.; GRAZIOTIN, D.; ORTU, M. Mining valence, arousal, and dominance: Possibilities for detecting burnout and productivity? In: *Proceedings of the 13th International Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2016. (MSR '16), p. 247–258. ISBN 978-1-4503-4186-8. Available at: <http://doi.acm.org/10.1145/2901739.2901752>.

MÄNTYLÄ, M. V.; LASSENIUS, C. What types of defects are really discovered in code reviews? *TSE*, v. 35, n. 3, p. 430–448, 2009.

MARTIN, R. C. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003. ISBN 0135974445.

MCARDLE, B. H.; ANDERSON, M. J. Fitting multivariate models to community data: A comment on distance-based redundancy analysis. *Ecology*, v. 82, n. 1, p. 290–297, 2001.

MCINTOSH, S.; KAMEI, Y.; ADAMS, B.; HASSAN, A. E. 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*. New York, NY, USA: ACM, 2014. (MSR 2014), p. 192–201. ISBN 978-1-4503-2863-0. Available at: <http://doi.acm.org/10.1145/2597073.2597076>.

MCINTOSH, S.; KAMEI, Y.; ADAMS, B.; HASSAN, A. E. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, v. 21, n. 5, p. 2146–2189, Oct 2016. ISSN 1573-7616. Available at: <https://doi.org/10.1007/s10664-015-9381-9>.

MCTEAR, M.; CALLEJAS, Z.; GRIOL, D. *The Conversational Interface: Talking to Smart Devices*. 1st. ed. Switzerland: Springer Publishing Company, Incorporated, 2016. ISBN 3319329650, 9783319329659.

MENDE, T.; KOSCHKE, R. Revisiting the evaluation of defect prediction models. In: *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*. New York, NY, USA: ACM, 2009. (PROMISE '09), p. 7:1–7:10. ISBN 978-1-60558-634-2. Available at: <http://doi.acm.org/10.1145/1540438.1540448>.

MENZIES, T. "how not to do it": Anti-patterns for data science in software engineering. In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion(ICSE-C)*. Austin, TX, USA: ACM, 2016. v. 00, p. 887. Available at: <doi.ieeecomputersociety.org/>.

MENZIES, T.; DEKHTYAR, A.; DISTEFANO, J.; GREENWALD, J. Problems with precision: A response to "comments on 'data mining static code attributes to learn defect

predictors"'. *IEEE Transactions on Software Engineering*, v. 33, n. 9, p. 637–640, Sep. 2007. ISSN 0098-5589.

MENZIES, T.; GREENWALD, J.; FRANK, A. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 33, n. 1, p. 2–13, Jan. 2007. ISSN 0098-5589. Available at: <https://doi.org/10.1109/TSE.2007.10>.

METCALFE, C. Biostatistics: A foundation for analysis in the health sciences. *Statistics in Medicine*, v. 20, n. 2, p. 324–326, 2001. Available at: <https://onlinelibrary.wiley.com/doi/abs/10.1002/1097-0258%2820010130%2920%3A2%3C324%3A%3AAID-SIM635%3E3.0.CO%3B2-O>.

MORALES, R.; MCINTOSH, S.; KHOMH, F. 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)*. Montreal, QC, Canada: IEEE Computer Society, 2015. p. 171–180. ISSN 1534-5351.

MORRIS, M. R.; BEGEL, A.; WIEDERMANN, B. Understanding the challenges faced by neurodiverse software engineering employees: Towards a more inclusive and productive technical workforce. In: *Proceedings of the 17th International ACM SIGACCESS Conference on Computers & Accessibility*. New York, NY, USA: ACM, 2015. (ASSETS '15), p. 173–184. ISBN 978-1-4503-3400-6. Available at: <http://doi.acm.org/10.1145/2700648.2809841>.

MOURA, I.; PINTO, G.; EBERT, F.; CASTOR, F. Mining energy-aware commits. In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. Florence, Italy: IEEE Computer Society, 2015. p. 56–67. ISSN 2160-1852.

MUKADAM, M.; BIRD, C.; RIGBY, P. C. Gerrit software code review data from android. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. Piscataway, NJ, USA: IEEE Press, 2013. (MSR '13), p. 45–48. ISBN 978-1-4673-2936-1. Available at: <http://dl.acm.org/citation.cfm?id=2487085.2487095>.

MUNAIAH, N.; MEYERS, B. S.; ALM, C. O.; MENEELY, A.; MURUKANNAIAH PRADEEP K. SAND PRUD'HOMMEAUX, E.; WOLFF, J.; YU, Y. Natural language insights from code reviews that missed a vulnerability. In: BODDEN, E.; PAYER, M.; ATHANASOPOULOS, E. (Ed.). *Engineering Secure Software and Systems*. Cham: Springer International Publishing, 2017. p. 70–86.

MURGIA, A.; TOURANI, P.; ADAMS, B.; ORTU, M. Do developers feel emotions? an exploratory analysis of emotions in software artifacts. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2014. (MSR 2014), p. 262–271. ISBN 978-1-4503-2863-0. Available at: <http://doi.acm.org/10.1145/2597073.2597086>.

NOVIELLI, N.; GIRARDI, D.; LANUBILE, F. A benchmark study on sentiment analysis for software engineering research. *CoRR*, abs/1803.06525, 2018.

NOVIELLI, N.; STRAPPARAVA, C. *Dialogue act classification exploiting lexical semantics*. Hershey, Pennsylvania, USA: IGI Global Disseminator of Knowledge, 2011. 80-106 p.

NUROLAHZADE, M.; NASEHI, S. M.; KHANDKAR, S. H.; RAWAL, S. The role of patch review in software evolution: An analysis of the mozilla firefox. In: *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*. New York, NY, USA: ACM, 2009. (IWPSE-Evol '09), p. 9–18. ISBN 978-1-60558-678-6. Available at: <http://doi.acm.org/10.1145/1595808.1595813>.

OMRAN, F. N. A. A.; TREUDE, C. Choosing an NLP library for analyzing software documentation: A systematic literature review and a series of experiments. In: *Proceedings of the 14th International Conference on Mining Software Repositories*. Piscataway, NJ, USA: IEEE Press, 2017. (MSR '17), p. 187–197. ISBN 978-1-5386-1544-7. Available at: <https://doi.org/10.1109/MSR.2017.42>.

ORTU, M.; DESTEFANIS, G.; KASSAB, M.; COUNSELL, S.; MARCHESI, M.; TONELLI, R. Would you mind fixing this issue? In: LASSENIUS, C.; DINGSØYR, T.; PAASIVAARA, M. (Ed.). *Agile Processes in Software Engineering and Extreme Programming*. Cham: Springer International Publishing, 2015. p. 129–140. ISBN 978-3-319-18612-2.

PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; POSHYVANYK, D.; LUCIA, A. D. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, v. 41, n. 5, p. 462–489, May 2015. ISSN 0098-5589.

PALOMBA, F.; TAMBURRI, D. A.; SEREBRENIK, A.; ZAIDMAN, A.; FONTANA, F. A.; OLIVETO, R. How do community smells influence code smells? In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. New York, NY, USA: ACM, 2018. (ICSE '18), p. 240–241. ISBN 978-1-4503-5663-3. Available at: <http://doi.acm.org/10.1145/3183440.3194950>.

PANG, B.; LEE, L.; VAITHYANATHAN, S. Thumbs up?: Sentiment classification using machine learning techniques. In: *Proceedings of the ACL-02 Conference on Empirical Methods in Natural Language Processing - Volume 10*. Stroudsburg, PA, USA: Association for Computational Linguistics, 2002. (EMNLP '02), p. 79–86. Available at: <https://doi.org/10.3115/1118693.1118704>.

PANGSAKULYANONT, T.; THONGTANUNAM, P.; PORT, D.; IIDA, H. Assessing MCR discussion usefulness using semantic similarity. In: *2014 6th International Workshop on Empirical Software Engineering in Practice*. Osaka, Japan: IEEE Computer Society, 2014. p. 49–54.

PANICHELLA, S. Summarization techniques for code, change, testing, and user feedback (invited paper). In: *IEEE Workshop on Validation, Analysis and Evolution of Software Tests*. Campobasso, Italy: IEEE, 2018. p. 1–5.

PARMANTO, B.; MUNRO, P. W.; DOYLE, H. R. Improving committee diagnosis with resampling techniques. In: TOURETZKY, D. S.; MOZER, M. C.; HASSELMO, M. E. (Ed.). *Advances in Neural Information Processing Systems 8*. MIT Press, 1996. p. 882–888. Available at: <http://papers.nips.cc/paper/1075-improving-committee-diagnosis-with-resampling-techniques.pdf>.

PASCARELLA, L.; SPADINI, D.; PALOMBA, F.; BRUNTINK, M.; BACCHELLI, A. Information needs in contemporary code review. In: . New York, NY, USA:

ACM, 2018. v. 2, n. CSCW, p. 135:1–135:27. ISSN 2573-0142. Available at: <http://doi.acm.org/10.1145/3274404>.

PROVOST, F.; FAWCETT, T. Analysis and visualization of classifier performance: Comparison under imprecise class and cost distributions. In: *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*. AAAI Press, 1997. (KDD'97), p. 43–48. Available at: <http://dl.acm.org/citation.cfm?id=3001392.3001400>.

PROVOST, F. J.; FAWCETT, T.; KOHAVI, R. The case against accuracy estimation for comparing induction algorithms. In: *Proceedings of the Fifteenth International Conference on Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998. (ICML '98), p. 445–453. ISBN 1-55860-556-8. Available at: <http://dl.acm.org/citation.cfm?id=645527.657469>.

RAHMAN, F.; POSNETT, D.; DEVANBU, P. Recalling the "imprecision" of cross-project defect prediction. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, 2012. (FSE '12), p. 61:1–61:11. ISBN 978-1-4503-1614-9. Available at: <http://doi.acm.org/10.1145/2393596.2393669>.

RAJARAMAN, A.; ULLMAN, J. D. Data mining. In: _____. *Mining of Massive Datasets*. Cambridge, UK: Cambridge University Press, 2011. p. 1—-17.

RAM, A.; SAWANT, A. A.; CASTELLUCCIO, M.; BACCHELLI, A. 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*. New York, NY, USA: ACM, 2018. (ESEC/FSE 2018), p. 201–212. ISBN 978-1-4503-5573-5. Available at: <http://doi.acm.org/10.1145/3236024.3236080>.

RAYMOND, E. S. *The Cathedral and the Bazaar*. 1st. ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1999. ISBN 1565927249.

REBOUÇAS, M.; PINTO, G.; EBERT, F.; TORRES, W.; SEREBRENIK, A.; CASTOR, F. An empirical study on the usage of the swift programming language. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Suita, Japan: IEEE Computer Society, 2016. v. 1, p. 634–638.

RIGBY, P.; CLEARY, B.; PAINCHAUD, F.; STOREY, M.; GERMAN, D. Contemporary peer review in action: Lessons from open source development. *IEEE Software*, v. 29, n. 6, p. 56–61, Nov 2012. ISSN 0740-7459.

RIGBY, P. C. *Understanding Open Source Software Peer Review: Review Processes, Parameters and Statistical Models, and Underlying Behaviours and Mechanisms*. Phd Thesis (PhD Thesis) — University of Victoria, Victoria, B.C., Canada, Canada, 2011. AAINR80365. Available at: <https://books.google.com.br/books?id=d87VtwEACAAJ>.

RIGBY, P. C.; BACCHELLI, A.; GOUSIOS, G.; MUKADAM, M. A mixed methods approach to mining code review data: Examples and a study of multicommit reviews and pull requests. In: BIRD, C.; MENZIES, T.; ZIMMERMANN,

T. (Ed.). *The Art and Science of Analyzing Software Data*. Boston: Morgan Kaufmann, 2015. p. 231–255. ISBN 978-0-12-411519-4. Available at: <http://www.sciencedirect.com/science/article/pii/B9780124115194000094>.

RIGBY, P. C.; BIRD, C. Convergent contemporary software peer review practices. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: ACM, 2013. (ESEC/FSE 2013), p. 202–212. ISBN 978-1-4503-2237-9. Available at: <http://doi.acm.org/10.1145/2491411.2491444>.

RIGBY, P. C.; GERMAN, D. M.; COWEN, L.; STOREY, M.-A. Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Trans. Softw. Eng. Methodol.*, ACM, New York, NY, USA, v. 23, n. 4, p. 35:1–35:33, Sep. 2014. ISSN 1049-331X. Available at: <http://doi.acm.org/10.1145/2594458>.

RIGBY, P. C.; STOREY, M.-A. Understanding broadcast based peer review on open source software projects. In: *Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA: ACM, 2011. (ICSE '11), p. 541–550. ISBN 978-1-4503-0445-0. Available at: <http://doi.acm.org/10.1145/1985793.1985867>.

RINGROSE, T.; HAND, D. Construction and assessment of classification rules. *Biometrics 53(3)*, 1997.

RUANGWAN, S.; THONGTANUNAM, P.; IHARA, A.; MATSUMOTO, K. The impact of human factors on the participation decision of reviewers in modern code review. *Empirical Software Engineering*, Sep 2018. ISSN 1573-7616. Available at: <https://doi.org/10.1007/s10664-018-9646-1>.

RUNESON, P.; HÖST, M. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, v. 14, n. 2, p. 131–164, Dec 2009. ISSN 1573-7616. Available at: <https://doi.org/10.1007/s10664-008-9102-8>.

RUNESON, P.; HÖST, M.; RAINER, A.; REGNELL, B. *Case Study Research in Software Engineering: Guidelines and Examples*. 1st. ed. New Jersey, USA: Wiley Publishing, 2012. ISBN 1118104358, 9781118104354.

SAIF, H.; FERNÁNDEZ, M.; HE, Y.; ALANI, H. On stopwords, filtering and data sparsity for sentiment analysis of twitter. In: *LREC 2014, Ninth International Conference on Language Resources and Evaluation. Proceedings.* European Language Resources Association (ELRA), 2014. p. 810–817. Available at: <http://oro.open.ac.uk/40666/>.

SEARLE, J. R. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge, London: Cambridge University Press, 1969.

SEREBRENIK, A. Emotional labor of software engineers. In: DEMEYER, S.; PARSAI, A.; LAGHARI, G.; BLADEL, B. van (Ed.). *Proceedings of the 16th edition of the BElgian-NEtherlands software eVOLution symposium*. Antwerp, Belgium: CEUR-WS.org, 2017. (CEUR Workshop Proceedings, v. 2047), p. 1–6.

SILLITO, J.; MURPHY, G. C.; VOLDER, K. D. Questions programmers ask during software evolution tasks. In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2006. (SIGSOFT '06/FSE-14), p. 23–34. ISBN 1-59593-468-5. Available at: <http://doi.acm.org/10.1145/1181775.1181779>.

SINGER, J.; VINSON, N. G. Ethical issues in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, v. 28, n. 12, p. 1171–1180, Dec 2002. ISSN 0098-5589.

SMITHSON, M. *Ignorance and uncertainty: Emerging paradigms*. New York, USA: Springer-Verlag New York, 1989.

STEELE, C. M.; ARONSON, J. Stereotype threat and the intellectual test performance of african americans. *Journal of personality and social psychology*, v. 69 5, p. 797–811, 1995.

STEIN, N.; LEVINE, L. Making sense out of emotion. In: _____. *Memories, thoughts, and emotions: Essays*. New Jersey, USA: Erlbaum, Hillsdale, NJ, 1991. p. 295–322.

STIVERS, T.; ENFIELD, N. A coding scheme for question-response sequences in conversation. *Journal of Pragmatics*, v. 42, n. 10, p. 2620 – 2626, 2010. ISSN 0378-2166.

STOL, K.-J.; RALPH, P.; FITZGERALD, B. Grounded theory in software engineering research: A critical review and guidelines. In: *Proceedings of the 38th International Conference on Software Engineering*. New York, NY, USA: ACM, 2016. (ICSE '16), p. 120–131. ISBN 978-1-4503-3900-1. Available at: <http://doi.acm.org/10.1145/2884781.2884833>.

STOLCKE, A.; COCCARO, N.; BATES, R.; TAYLOR, P.; ESS-DYKEMA, C. V.; RIES, K.; SHRIBERG, E.; JURAFSKY, D.; MARTIN, R.; METEER, M. Dialogue act modeling for automatic tagging and recognition of conversational speech. *Comput. Linguist.*, MIT Press, Cambridge, MA, USA, v. 26, n. 3, p. 339–373, sep 2000. ISSN 0891-2017. Available at: <https://doi.org/10.1162/089120100561737>.

STRAUSS, A.; CORBIN, J. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. Newbury Park, California: Sage Publications, 1990.

SUTHERLAND, A.; VENOLIA, G. Can peer code reviews be exploited for later information needs? In: *2009 31st International Conference on Software Engineering - Companion Volume*. Vancouver, BC, Canada: IEEE, 2009. p. 259–262.

SVOBODA, J. *Effectively Combining Static Code Analysis and Manual Code Reviews*. Master's Thesis (Master's thesis) — Masaryk University, Faculty of Informatics, Brno, 2014. Available at: <https://theses.cz/id/expvz4>.

TAO, Y.; DANG, Y.; XIE, T.; ZHANG, D.; KIM, S. How do software engineers understand code changes?: An exploratory study in industry. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. New York, NY, USA: ACM, 2012. (FSE '12), p. 51:1–51:11. ISBN 978-1-4503-1614-9. Available at: <http://doi.acm.org/10.1145/2393596.2393656>.

TAO, Y.; HAN, D.; KIM, S. Writing acceptable patches: An empirical study of open source project patches. In: *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*. Washington, DC, USA: IEEE Computer Society, 2014. (ICSME '14), p. 271–280. ISBN 978-1-4799-6146-7. Available at: <http://dx.doi.org/10.1109/ICSME.2014.49>.

TAO, Y.; KIM, S. Partitioning composite code changes to facilitate code review. In: *Proceedings of the 12th Working Conference on Mining Software Repositories*. Piscataway, NJ, USA: IEEE Press, 2015. (MSR '15), p. 180–190. ISBN 978-0-7695-5594-2. Available at: <http://dl.acm.org/citation.cfm?id=2820518.2820541>.

THONGTANUNAM, P.; MCINTOSH, S.; HASSAN, A. E.; IIDA, H. Review participation in modern code review - an empirical study of the android, qt, and openstack projects. *Empirical Software Engineering*, v. 22, n. 2, p. 768–817, 2017. Available at: <https://doi.org/10.1007/s10664-016-9452-6>.

THONGTANUNAM, P.; TANTITHAMTHAVORN, C.; KULA, R. G.; YOSHIDA, N.; IIDA, H.; MATSUMOTO, K. ichi. 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)*. Montreal, QC, Canada: IEEE Computer Society, 2015. p. 141–150. ISSN 1534-5351.

THONGTANUNAM, P.; YANG, X.; YOSHIDA, N.; KULA, R. G.; CRUZ, A. E. C.; FUJIWARA, K.; IIDA, H. ReDA: A web-based visualization tool for analyzing modern code review dataset. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. Victoria, BC, Canada: IEEE Computer Society, 2014. p. 605–608. ISSN 1063-6773.

THORNTON, C.; HUTTER, F.; HOOS, H. H.; LEYTON-BROWN, K. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In: *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA: ACM, 2013. (KDD '13), p. 847–855. ISBN 978-1-4503-2174-7. Available at: <http://doi.acm.org/10.1145/2487575.2487629>.

TICHY, W. F. Rcs — a system for version control. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 15, n. 7, p. 637–654, jul 1985. ISSN 0038-0644. Available at: <http://dx.doi.org/10.1002/spe.4380150703>.

TOURANI, P.; ADAMS, B.; SEREBRENIK, A. Code of conduct in open source projects. In: *24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*. United States: IEEE Computer Society, 2017. p. 24–33. ISBN 978-1-5090-5501-2.

TRAUM, D. R. 20 questions for dialogue act taxonomies. *Journal of Semantics*, v. 17, p. 7–30, 2000.

TREUDE, C.; BARZILAY, O.; STOREY, M.-A. How do programmers ask and answer questions on the web? (nier track). In: *Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA: ACM, 2011. (ICSE '11), p. 804–807. ISBN 978-1-4503-0445-0. Available at: <http://doi.acm.org/10.1145/1985793.1985907>.

TURHAN, B. On the dataset shift problem in software engineering prediction models. *Empirical Software Engineering*, v. 17, n. 1, p. 62–74, Feb 2012. ISSN 1573-7616. Available at: <https://doi.org/10.1007/s10664-011-9182-8>.

UWANO, H.; NAKAMURA, M.; MONDEN, A.; MATSUMOTO, K.-i. Analyzing individual performance of source code review using reviewers' eye movement. In: *Proceedings of the 2006 Symposium on Eye Tracking Research &Amp; Applications*. New

York, NY, USA: ACM, 2006. (ETRA '06), p. 133–140. ISBN 1-59593-305-0. Available at: <http://doi.acm.org/10.1145/1117309.1117357>.

VARTTALA, T. *Hedging in the scientifically oriented discourse. Exploring variation according to discipline and intended audience*. Phd Thesis (PhD Thesis) — University of Tampere, 2001. Available at: <https://tampub.uta.fi/handle/10024/67148>.

VASILESCU, B.; CAPILUPPI, A.; SEREBRENIK, A. Gender, representation and online participation: A quantitative study. *Interacting with Computers*, v. 26, n. 5, p. 488–511, 2014. Available at: <https://doi.org/10.1093/iwc/iwt047>.

VASILESCU, B.; FILKOV, V.; SEREBRENIK, A. Perceptions of diversity on github: A user survey. In: *Proceedings of the Eighth International Workshop on Cooperative and Human Aspects of Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2015. (CHASE '15), p. 50–56.

VASILESCU, B.; POSNETT, D.; RAY, B.; BRAND, M. G. van den; SEREBRENIK, A.; DEVANBU, P.; FILKOV, V. Gender and tenure diversity in github teams. In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, 2015. (CHI '15), p. 3789–3798. ISBN 978-1-4503-3145-6. Available at: <http://doi.acm.org/10.1145/2702123.2702549>.

VIVIANI, G.; JANIK-JONES, C.; FAMELIS, M.; MURPHY, G. C. The structure of software design discussions. In: *Proceedings of the 11th International Workshop on Cooperative and Human Aspects of Software Engineering*. New York, NY, USA: ACM, 2018. (CHASE '18), p. 104–107. ISBN 978-1-4503-5725-8. Available at: <http://doi.acm.org/10.1145/3195836.3195841>.

VIVIANI, G.; JANIK-JONES, C.; FAMELIS, M.; XIA, X.; MURPHY, G. C. What design topics do developers discuss? In: *Proceedings of the 26th Conference on Program Comprehension*. New York, NY, USA: ACM, 2018. (ICPC '18), p. 328–331. ISBN 978-1-4503-5714-2. Available at: <http://doi.acm.org/10.1145/3196321.3196357>.

VOSOUGHI, S.; ROY, D. A semi-automatic method for efficient detection of stories on social media. *CoRR*, abs/1605.05134, 2016. Available at: <http://arxiv.org/abs/1605.05134>.

VOSOUGHI, S.; ROY, D. Tweet acts: A speech act classifier for twitter. *CoRR*, abs/1605.05156, 2016. Available at: <http://arxiv.org/abs/1605.05156>.

WANG, J.; SHIH, P. C.; WU, Y.; CARROLL, J. M. Comparative case studies of open source software peer review practices. *Inf. Softw. Technol.*, Butterworth-Heinemann, Newton, MA, USA, v. 67, n. C, p. 1–12, Nov. 2015. ISSN 0950-5849. Available at: <https://doi.org/10.1016/j.infsof.2015.06.002>.

WESEL, P. van; LIN, B.; ROBLES, G.; SEREBRENIK, A. Reviewing career paths of the openstack developers. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME'2017)*. Shanghai, China: IEEE Computer Society, 2017. p. 544–548. Available at: <https://doi.org/10.1109/ICSME.2017.25>.

WIEGERS, K. E. *Peer Reviews in Software: A Practical Guide*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0-201-73485-0.

WILSON, A. Mob programming - what works, what doesn't. In: LASSENIUS, C.; DINGSØYR, T.; PAASIVAARA, M. (Ed.). *Agile Processes in Software Engineering and Extreme Programming*. Cham: Springer International Publishing, 2015. p. 319–325. ISBN 978-3-319-18612-2.

YANG, D.; WEN, M.; HOWLEY, I.; KRAUT, R.; ROSE, C. Exploring the effect of confusion in discussion forums of massive open online courses. In: *ACM Conference on Learning @ Scale*. Vancouver, BC, Canada: ACM, 2015. p. 121–130. ISBN 978-1-4503-3411-2.

YANG, X.; KULA, R. G.; YOSHIDA, N.; IIDA, H. Mining the modern code review repositories: A dataset of people, process and product. In: *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. New York, NY, USA: ACM, 2016. p. 460–463.

ŽEGARAC, V.; CLARK, B. Phatic interpretations and phatic communication. *Journal of Linguistics*, Cambridge University Press, v. 35, n. 2, p. 321–346, 1999.

ZHOU, L.; ZHANG, D. A heuristic approach to establishing punctuation convention in instant messaging. *IEEE Transactions on Professional Communication*, v. 48, n. 4, p. 391–400, Dec 2005. ISSN 0361-1434.

ZIMMERMANN, T. Card-sorting: From text to themes. In: MENZIES, T.; WILLIAMS, L.; ZIMMERMANN, T. (Ed.). *Perspectives on Data Science for Software Engineering*. Boston: Morgan Kaufmann, 2016. p. 137 – 141. ISBN 978-0-12-804206-9. Available at: <https://www.sciencedirect.com/science/article/pii/B9780128042069000271>.

APPENDIX A – LIST OF FEATURES FROM THE CONFUSION CODING SCHEME

Below we list all features from each category of our confusion coding scheme.

A.0.1 Hedges

could
may
maybe
might
might feel a bit
perhaps
seems
seems like
sort of
a real
a regular
a self-styled
a true
a veritable
actually
all but a
all but technically
almost
anything but a
as it were
basically
can be tooked upon as
can be viewed as
crypto
crypto-
details aside
especially
essentially
exceptionally
for the most part
he as much as
he calls himself a
he calls himself an
in a manner of speaking
in a real sense
in a sense
in a way
in an important sense
in essence
in name only
in one sense
kind of
largely
literally
loosely speaking
more or less
mostly
mutatis mutandis
nominally
often
par excellence
particularly
practically
pretty
pretty much
principally
pseudo
pseudo-
quintessential
quintessentially
rather
really
relatively
roughly
she as much as
she calls herself a
she calls herself an
so to say
somewhat
strictly speaking
technically
they as much as
they call themselves a
they call themselves an
typical
typically
very
virtually
about
allegation
allegations
allege
alleged
alleges
alleging
allude
alluded
alludes
alluding
alternative
alternatives
anticipate
anticipated
anticipates
anticipating
appear
appearance
appearances
appeared
appearing
appears
appreciable
approach
approaches
approximate
approximated
approximately
approximates
approximating
approximation
approximations
argue
argued
argues
arguing
argument
arguments
around
assert
asserted
asserting
assertion
assertions
asserts
assess
assessed
assesses
assessing
assessment
assessments
assume
assumed
assumes
assuming
assumption
assumptions
avenue
avenues
belief
beliefs

believe
believed
believes
believing
can
candidate
candidates
central
chance
chances
charge
charged
charges
charging
claim
claimed
claiming
claims
close
closely
clue
clues
conceive
conceived
conceives
conceiving
concept
concepts
conceptualization
conceptualizations
conceptualize
conceptualized
conceptualizes
conceptualizing
conclude
concluded
concludes
concluding
conclusion
conclusions
consider
considerable
considerably
considered
considering
considers
construct
constructs
contend
contended
contending
contends
contention
contentions
deduction
deductions
deem
deemed
deeming
deems
devastating
doubt
doubted
doubting
doubts
dramatically
drastically
envision
envisioned
envisioning
envisions
estimate
estimated
estimates
estimating
estimation
estimations
evaluate
evaluated
evaluates
evaluating
evaluation
evaluations
expect
expectancies
expectancy
expectation
expectations
expected
expecting
expects
extrapolate
extrapolated
extrapolates
extrapolating
fair
fairly
fantasies
fantasy
fear
feared
fearing
fears
feel
feeling
feels
felt
find
finding
finds
forecast
forecasting
forecasts
foresaw
foresee
foreseeing
foreseen
foresees
foreshadow
foreshadowed
foreshadowing
foreshadows
found
greatly
gross
guess
guesses
held
highly
hint
hinted
hinting
hints
hold
holding
holds
hope
hoped
hopes
hoping
hunch
hunches
hypotheses
hypothesis
hypothesize
hypothesized
hypothesizes
hypothesizing
idea
ideas
illusion
illusions
imagine
imagined
imagines
imagining
immense
implicate
implicated
implicates
implicating
implication
implications
implied
implies
imply
implying
impression
impressions
impressive
inclination
inclinations
indication
indications
infer
infered
inference
inferences
infering
inferred
inferring
interpret
interpretation
interpretations
interpreted
interpreting
interprets
judge
judged
judges
judging
judgment
judgments
just
large
likelihood
likelihoods
little
look
looked

illusion
illusions
imagine
imagined
imagines
imagining
immense
implicate
implicated
implicates
implicating
implication
implications
implied
implies
imply
implying
impression
impressions
impressive
inclination
inclinations
indication
indications
infer
infered
inference
inferences
infering
inferred
inferring
interpret
interpretation
interpretations
interpreted
interpreting
interprets
judge
judged
judges
judging
judgment
judgments
just
large
likelihood
likelihoods
little
look
looked
looking
looks
main
mainly
maintain
maintained
maintaining
maintains
major
marked
markedly
massively
moderate
moderately
modest
modestly
must
nearly
negligible
notable
noticeable
notion
notions
odds
opinion
opinions
opportunities
opportunity
option
options
partially
partly
perceive
perceived
perceives
perceiving
perception
perceptions
perspective
perspectives
philosophies
philosophy
point of view
point to
point toward
pointed to
pointed toward
pointing to
pointing toward
points of view
points to
points toward
portend
portended
portending
portends
posit
posited
positing
posits
possibilities
possibility
postulate
postulated
postulates
postulating
potential
potentials
predict
predicted
predicting
prediction
predictions
predicts
predominantly
predominately
premise
premises
presume
presumed
presumes
presuming
presumption
presumptions
primarily
primary
principal
probabilities
probability
project
projected
projecting
projection
projections
projects
promise
promises
proposal
proposals
propose
proposed
proposes
proposing
proposition
propositions
prospect
prospects
proximate
quite
reason
reasonable
reasonably
reasoned
reasoning
reasons
reassess
reassessed
reassesses
reassessing
reestimate
reestimated
reestimates
reestimating
regard
regarded
regarding
regards
relative
remarkable
rough
saw
scenario
scenarios
scheme
schemes
see
seeing
seem
seemed
seeming
seen
sees
shortly
should
sign
significant
significantly
signs
slight
slightly

small
some
sound
sounded
sounding
sounds
speculate
speculated
speculates
speculating
speculation
speculations
strongly
substantial
substantially
suggest
suggested
suggesting
suggestion
suggestions
suggests
suppose
supposed
supposes
supposing
supposition
suppositions
surmise
surmised
surmises
surmising
suspect
suspected
suspecting
suspects
suspicion
suspicions
tend
tended
tendencies
tendency
tending
tends
tenet
tenets
theories
theorize
theorized
theorizes
theorizing
theory
theses
thesis
think
thinking
thinkings
thinks
thought
thoughts
threat
threats
tiny
tremendous
trend
trends
vastly
view
viewed
viewing
viewpoint
viewpoints
views
virtual
vision
visions
widely
will
wish
wished
wishes
wishing
wonder
wondered
wondering
wonders
worried
worries
worry
worrying
would

A.0.2 Probables

likely	theoretically	implausible	frequent
sometimes	unlikely	improbable	general
probably	commonly	indicative	normal
frequently	generally	plausible	occasional
could have	normally	possible	pervasive
often	occasionally	potential	popular
apparently	oftentimes	predictive	prevalent
arguably	rarely	probabilistic	rare
conceivably	routinely	probable	regular
ostensibly	seldom	prone to	scarce
perhaps	typically	putative	typical
possibly	usually	speculative	uncommon
potentially	apparent	suggestive	not uncommon
presumably	apt to	theoretical	usual
seemingly	conceivable	characteristic	
supposedly	doubtful	common	
tentatively	hypothetical	commonplace	

A.0.3 Hypotheticals

I cannot imagine
What if
Imagine
I was wondering

A.0.4 I Statements

I wonder	I believe
I struggle	I reckon
I am not sure	I suppose
I am curious	I suspect
I bet	I do not understand
I would guess not	I could not understand
I am confused	I do not know
I guess	I do not get it
I doubt	I am unsure
I think	

A.0.5 Nonverbals

hm
hmm
hmmm
hmmmm
mm
mmm
oh
ooh

A.0.6 Meta

It appears	I did not know
It seems	I did not get it
argues that	It turns out
claims that	wtf
contends that	what the fuck
maintains that	what the heck
There is a probability	what the hell
There is a likelihood	It is not clear
There is a possibility	It was not clear
There is a chance	confusing
I did not understand	weird