Incivility Detection in Open Source Code Review and Issue Discussions

Isabella Ferreira*a, Ahlaam Rafiqb, Jinghui Chengc

aDepartment of Computer and Software Engineering, Polytechnique Montréal, Montréal, Quebec, Canada
bDepartment of Physics, Indian Institute of Technology, Guwahati, Assam, India
cDepartment of Physics, Indian Institute of Technology, Guwahati, Assam, India

Abstract

Given the democratic nature of open source development, code review and issue discussions may be uncivil. Incivility, defined as features of discussion that convey an unnecessarily disrespectful tone, can have negative consequences to open source communities. To prevent or minimize these negative consequences, open source platforms have included mechanisms for removing uncivil language from the discussions. However, such approaches require manual inspection, which can be overwhelming given the large number of discussions. To help open source communities deal with this problem, in this paper, we aim to compare six classical machine learning models with BERT to detect incivility in open source code review and issue discussions. Furthermore, we assess if adding contextual information in the previous email/comment improves the models’ performance and how well the models perform in a cross-platform setting. We found that BERT performs better than classical machine learning models, with a best F1-score of 0.95. Furthermore, classical machine learning models tend to underperform to detect tone-bearing and civil discussions. Our results show that adding the previous email/comment to BERT did not improve its performance and that none of the analyzed classifiers had an outstanding performance in a cross-platform setting. Finally, we provide insights into the tones that the classifiers misclassify and lessons learned for using automated techniques in incivility detection.

Keywords: incivility, code review, github issues, open source, bert, machine learning

1. Introduction

Open source software (OSS) development provides abundant opportunities for public discussions, which happen within the context of issue tracking, bug report, code review, and user feedback, to just name a few. These opportunities characterize the democratic essence of open source development by allowing anyone who has the relevant knowledge to contribute to the development process and shape the project one way or another. However, as in all types of public discussions, conversations that happen in open source development can become uncivil. Take, for example, this code review comment of a patch submitted to the Linux kernel: “What the F*CK, guys? This piece-of-shit commit is marked for stable, but you clearly never even test-compiled it, did you?” Although the comment expressed opinions on a technical issue, the commenter used an unnecessarily disrespectful tone.

We define incivility, in the context of software engineering (SE) in general and OSS development in specific, as features of discussion that convey an unnecessarily disrespectful tone toward the discussion forum, its participants, or its topics [1]. This concept is related to other constructs used to describe unhealthy discussions in the context of SE, such as hate speech, offensive language, toxicity, and pushback. Although these different concepts might share certain characteristics with incivility, in our definition, the concept of incivility covers a broader spectrum [1]. That is, the construct of toxicity often emphasizes the impact on the target of incivility; e.g., “make someone leave a discussion” [2] and “impact the health of FOSS peer production communities” [3]. Yet, the concept of incivility is not confined to this aspect. While hate speech and offensive language usually focus on a specific emotion/tone (such as entitlement [4], insults [5], or racist terms [6]), incivility does not [1]. Finally, while both incivility and pushback are unnecessary behaviors, pushback focuses on a specific action (i.e., “a reviewer blocking a change request” [7]), while incivility is broader. Based on the aforementioned differences, we argue that compared to the other constructs, the concept of incivility is not confined to a particular impact or specific emotions/tones and thus is more general.

Although incivility can be rare, uncivil expressions often have important impacts on the communication and the discussion participants, resulting in escalated incivility, discontinued conversation, or disengaged contributors [1]. As such, many major SE platforms such as Stack Overflow and GitHub have incorporated mechanisms for labeling and removing offensive and toxic languages [9, 10]. Many of these approaches involve manual inspection, which requires considerable human efforts given the large amount of content generated daily in those platforms. Additionally, they often target one construct and ignores the broader spectrum of uncivil discussion features. Hence,
automated techniques for detecting uncivil communication in software engineering platforms would help open source communities to proactively manage uncivil interactions.

Several tools have been developed by researchers to detect toxicity [4, 14, 15], offensive language [5], and pushback [7]. However, these tools do not perform well when it comes to new samples or SE discussions [2, 4, 14, 46]. Additionally, the concept of incivility may cover a wider spectrum than those previously explored constructs [1]. Based on these results, we were motivated to build an incivility-specific classifier.

Developing such a classifier, however, involves major challenges. First, although with great impacts, uncivil exchanges in open source communities can be infrequent. Previous studies identified that only 7.25% of code review comments [1] and 8.82% of issue comments [8] demonstrate incivility. The lack of uncivil cases poses challenges in creating datasets for training and evaluating the automated techniques. Second, incivility can be manifested in various ways. For example, previous work has identified many characteristics of discussion that can be seen as uncivil. Among them, there are straightforward features such as name calling and vulgarity. But at the same time, incivility can be manifested through discussion characteristics such as irony, mocking, and threat that are difficult to detect automatically [1]. As a result, many existing software engineering sentiment analysis tools do not perform well when detecting incivility [1]. Finally, incivility can be “very much in the eye of the beholder” [11]. Thus, the discussion context can have strong indications on whether a comment is uncivil. So analyzing the text in isolation may lead to inaccurate results.

We aimed to address these challenges in our research by exploring machine learning and deep learning techniques trained with two manually labeled datasets [1, 8]. Particularly, we investigated data augmentation and class balancing techniques to facilitate the detection of infrequent uncivil comments. The datasets we used are also created by considering the complexity of civil and uncivil discussion features, such as humility, confusion, sadness, irony, and mocking. Additionally, by exploring how the previous discussion can affect the model performance, we aimed to incorporate the beholder’s perspective, addressing the problem of classifying the text in isolation. More especially, we pose the following three research questions.

**RQ1. How well can machine learning models (including BERT-based model and classical machine learning models) detect incivility?**

To the best of our knowledge, none of the previous research has built classifiers to detect incivility in open source code review or issue discussions. Hence, it is unknown if incivility can be automatically detected with a good performance in such discussions. Previous studies have identified that BERT-based deep learning models [12] outperformed classical machine learning models when detecting related concepts such as toxicity [15] and offensive language [5]. Based on these results, we hypothesize that BERT can detect incivility in open source code review and issue discussions with satisfactory performance and can outperform the classical machine learning models.

**RQ2. To what extent does adding the previous email/comment help to detect incivility in code review and issue discussions?**

Ferreira et al. [1] have shown that the existing discussion and the context are important factors that should be considered when detecting incivility in code review discussions. However, Murgia et al. [16] found that adding previous discussions did not help human raters reach an agreement on assessing emotion expressed in comments on issue tracking systems. Considering these conflicting conclusions in previous work, we aim to assess if adding the previous discussion helps to improve the performance of automated incivility detection techniques.

**RQ3. How well do the incivility detection techniques work in a cross-platform setting?**

Building a manually annotated gold standard for incivility detection on a particular platform is a time-consuming task. Currently, only two datasets are available on the literature [1, 8], focusing on code review discussions and issue discussions, respectively. However, discussions that happened on different platforms could have characteristics that indicate incivility in different ways. Hence, in RQ3, we aim to assess if it is feasible to use BERT and classical machine learning models to detect incivility in a cross-platform setting. This information will help us assess the performance of incivility detection on a new dataset when a gold standard is not available.

**RQ4: What types of discussion features tend to be misclassified by the incivility detection techniques?**

Incivility is a broad and complex construct that can be manifested by different discussion features such as *irony*, *mocking*, and *bitter frustration* [1, 8]. The performance of the automated detectors may vary when detecting incivility demonstrating different features. Thus in RQ4, we investigate the performance of incivility detection techniques on different discussion features. The goal of this analysis is to identify error patterns and further inform the creation of practical tools that address limitations of automated techniques.

Similar to our previous work of manual incivility labeling [1, 8], our incivility detection approach follows two steps. First, SE discussions are classified into *tone-bearing* and *non-tone-bearing* groups; the former includes discussions that demonstrate at least one tone-bearing discussion feature (such as sadness, appreciation, or frustration) while the latter includes other discussions that are often purely technical. Then, we classify the tone-bearing discussions into *civil* or *uncivil* depending on the specific discussion feature that they demonstrate [1].

Our results show that BERT outperforms classical machine learning techniques, having an outstanding performance when detecting incivility in both code review and issue discussions ($F1 > 0.9$). Additionally, we found that classical machine learning models tend to underperform to classify non-tone-bearing comments as well as civil sentences in both datasets. Our findings also indicated that adding the previous email/comment did not improve the incivility classification. Furthermore, we found that although none of the classifiers have had an outstanding performance to detect incivility in a cross-platform setting, all
classifiers were able to perform well when detecting the non-tone-bearing and uncivil classes. Our analysis of misclassified cases also shed light on the complexity of incivility detection when considering the various features of discussions. Together, our results allow us to summarize lessons learned for developing incivility detection techniques. The dataset, source code, and evaluation results are available at our replication package.1

2. Background & Related Work

In this section, we discuss the background information and related work on (i) machine learning for text classification, (ii) automated detection of unhealthy discussions in online communication platforms, and (iii) automated detection of unhealthy discussions in software engineering.

2.1. Machine Learning for Text Classification

Text classification is a classical natural language processing (NLP) problem that aims at assigning labels to textual documents, such as sentences or paragraphs [17]. Currently, there are two kinds of machine learning approaches for automatic text classification, namely classical machine learning-based models and neural network-based approaches.

Common classical machine learning-based models include classification and regression tree (CART), k-nearest-neighbors (KNN), logistic regression, naïve Bayes, random forest, and support vector machine (SVM), among others. They were applied in various general text classification tasks [18, 19, 20, 21], as well as for software engineering tasks in specific [22, 23, 24, 25]. To use these models, features need to be first defined and extracted from textual documents, then fed into the classifier for prediction. Popular features for textual data include bag of words (BoW) and term frequency–inverse document frequency (tf-idf). Although widely used, classical machine learning classifiers have a major limitation. That is, choosing the proper features for each domain requires extensive domain knowledge; thus it is hard to define cross-domain or cross-task features [17]. In this work, we assess if it is feasible to use the six aforementioned classical machine learning-based models to detect incivility in code review and issue discussions.

To solve the aforementioned challenges, neural network-based approaches have been widely explored in the literature to address text classification tasks [12, 26, 27, 28, 29, 30, 31]. In 2018, Devlin et al. [12] proposed BERT (Bidirectional Encoder Representations from Transformers), which is currently the state of the art embedding model [17]. The BERT base model consists of 110M parameters and has been trained on BookCorpus [32] and English Wikipedia [33], which include a total of 3.3 billion words. BERT is trained with two objectives: masked language modeling (MLM) and next sentence prediction (NSP). MLM allows the model to learn a bidirectional representation of the sentence by randomly masking 15% of the words in the input and then training the model to predict the masked words. For NSP, the model concatenates two masked sentences as inputs during the pretraining phase and then predicts if the two sentences are continuous in the text or not.

Many variants have been made to BERT since it was proposed [34, 35, 36, 37, 38]. RoBERTa [34], for example, is a more robust implementation of BERT, trained with a much larger amount of data. ALBERT [35] optimizes BERT by lowering its memory consumption and increasing its training speed. DistillBERT [36] uses knowledge distillation, i.e., a compression technique in which a compact model is trained to reproduce the behavior of a larger model, to reduce the size of the BERT model. SpanBERT [37] is a pre-trained method to better represent and predict spans of text. CodeBERT [38] is a pre-trained language model for both programming languages and natural languages. In many NLP [13] and software engineering problems [5, 39, 40, 41], BERT has demonstrated to have better performance than classical machine learning models. Thus, we investigate BERT’s ability to detect incivility in code review and issue discussions. To simplify this initial exploration, we used the original BERT model instead of its variants.

2.2. Automated Detection of Incivility in Online Communication Platforms

By using either classical machine learning-based models or neural network-based approaches, many authors have tried to automatically detect incivility on online platforms, such as in news discussions [42, 43] and Twitter [44]. Daxenberge et al. [42], for example, sought to understand incivility (defined as “expressions of disagreement by denying and disrespecting opposing views”) on user comments of nine German media outlets on Facebook. By using a logistic regression classifier, they found that incivility can be identified with an overall F1-score of 0.46. To assess how well machine learning models are able to detect incivility (defined as “the act of sending or posting mean text messages intended to mentally hurt, embarrass or humiliate another person using computers, cell phones, and other electronic devices”) detection with character-level bidirectional long short-term memory (bi-LSTM) and character-level convolutional neural networks (CNNs) with a rectified linear unit (ReLU) outperformed the best baseline model with a F1-score of 0.82 [44].

In our literature review, we were not able to find previous research investigating automated detection of incivility in software engineering settings, although some recent work focused on detecting unhealthy discussions that we review in the next section. In this paper, we thus address this gap by leveraging the concepts and the datasets established in our previous work about incivility in code review [1] and issue discussions [8].

1https://doi.org/10.6084/m9.figshare.24603237
2.3. Automated Detection of Unhealthy Discussions in Software Engineering

Unhealthy interactions are often characterized in software engineering (SE) discussions as toxicity [2, 3, 15, 45, 46], offensive language [5, 14], heated discussions [8, 22], hate speech [4], and pushback [7, 46]. Table 1 presents the studies proposing models to detect different kinds of unhealthy interactions. We compare our study with the literature with respect to the models implemented, the used dataset, and the techniques to improve the models’ performance, such as cross-validation (i.e., iteratively training and testing a model using different portions of the data), data augmentation (i.e., artificially increasing data points in the training set by generating new data points from existing ones), class balancing (i.e., adjusting the class sizes with unbalanced samples), and hyperparameter optimization (i.e., choosing the hyperparameter combination for optimum performance).

Previous studies identified that open source contributors might have different, sometimes negative, communication styles. For example, a Naive Bayes classifier identified that the leaders of the Linux Kernel Mailing List (LKML) have different communication styles (F1-score = 0.96) [50]; some used more impolite, rude, aggressive, or offensive words. Offensive language (defined as “communication that contains gutter language, swearing, racist, or offensive content”) can also be identified in other platforms such as GitHub, Gitter, Slack, and Stack Overflow, with more than 97% of accuracy using BERT [5].

In addition to having different communication styles, contributors might also demonstrate negative emotions when expressing themselves in open source discussions. Anger, for example, can be accurately identified in Jira discussions with SVM (F1-score = 0.81), J48 decision tree (F1-score = 0.77), and Naive Bayes (F1-score = 0.72) [51]. Another negative emotion that might emerge in code review discussions, more specifically, is the feeling of pushback, which is characterized by interpersonal conflicts, impatience, disappointment, and frustration [7]. In Google’s code review discussions, a logistic regression model found that code review authors are between 3.0 and 4.1 times more likely to experience the feeling of pushback for at least once and between 7.0 and 13.7 times more likely to experience it multiple times when compared to code reviews that were not flagged with a potential feeling of pushback.

Finally, toxic language, i.e., hate speech and microagressions, can be identified with automated methods. For example, using the SVM model Raman et al. detected toxicity in GitHub issues with a precision of 0.75, but a low recall of 0.35 [4]. Sarker et al. [14] also tested the SVM classifier on other 100k randomly sampled GitHub issues; they found that the precision decreased to 0.50, demonstrating that the model might be overfitting to the training set. Similarly, toxicity can be identified in Gerrit code review and Gitter discussions, with the STRUDEL toxicity detector having an F1-score of 0.49 and 0.73, respectively [14]. Interestingly, Sarker et al. [14] found that toxicity detectors tend to perform worse on more formal SE discussions, such as code reviews, than on informal conversations such as Gitter messages. Finally, a recent study [15] proposes ToxiCR, ten supervised machine learning algorithms with a combination of text vectorization and processing steps on 19,651 code review comments. ToxiCR has an F1-score of 0.89.

Our work differs from the previous works in several ways. First, our study proposes an incivility-specific classifier for SE discussions. This study builds upon our previous work on characterizing incivility in code review discussions of rejected patches from the LKML and GitHub issue discussions locked as too heated. We chose to compare six classical machine learning models (Classification and Regression Tree (CART), k-Nearest Neighbors (KNN), Logistic Regression, Naive Bayes, Random Forest, and SVM) with BERT. Additionally, we use four strategies to augment our data (i.e., synonym replacement, random insertion, random swap, and random deletion) and compare three class balancing techniques, i.e., random undersampling, random oversampling, and SMOTE. We also perform hyperparameter optimization with Grid Search on the classical machine learning models’ hyperparameters and Bayesian Optimization on BERT’s hyperparameters to improve the models’ performance. Finally, the performance of our models is evaluated in a 5-fold cross-validation. On top of evaluating the performance of the machine learning models in detecting incivility, we also analyze the impact of the previous discussion and the feasibility of detecting incivility in a cross-platform setting. Finally, we assess what are the different kinds of uncivil discussion features that the models are able to (mis)classify.

3. Study Design

Figure 1 depicts the key components in the pipeline of the classifiers explored in this study. After preprocessing the datasets (Sections 3.1 and 3.2) and extracting the features for the classical machine learning models (Section 3.3), we stratify our dataset into train, test, and validation sets. Then, we augment our training set (Section 3.4.1) and balance our classes (Section 3.4.2). We investigate six classical machine learning models (Section 3.5.1) and one deep learning model (Section 3.5.2). During training, to obtain the optimal models, we perform hyperparameter tuning to find the best set of hyperparameters. We then test the trained classifiers on the test set and assess the performance of each classifier according to four performance metrics (Section 3.5.3). Finally, we perform experiments and additional analysis to answer the four RQs (Section 3.6).

3.1. Datasets

The general goal of this study is to assess the extent to which incivility can be detected in code reviews and issue discussions. To the best of our knowledge, only two incivility datasets are available in the literature, i.e., a code review dataset comprising code review emails of rejected patches that were sent to Linux Kernel Mailing List (LKML) [1] and an issues dataset comprising GitHub issues locked as too heated [8]. We used both datasets to train our classifiers.

For each dataset, the natural language emails (in the case of the code review dataset) and comments (in the case of the GitHub issues dataset) were first labeled as non-tone-bearing or tone-bearing. Following the definition used by the datasets, the
| Authors          | Goal                                                                 | Model                               | Dependent variables                                                                 | Dataset       | Techniques                                      |
|------------------|----------------------------------------------------------------------|-------------------------------------|-------------------------------------------------------------------------------------|---------------|------------------------------------------------|
| Schneider et al. [50] | Identify the discourse patterns of the leaders of the LKML.         | Naive Bayes                         | Email sent by Linus Torvalds Email sent by Greg Kroah-Hartman                      | Code reviews  | Cross-validation: ✓ Data augmentation: ✗ Class balancing: ✗ Hyperparameter optimization: ✗ |
| Gachechiladze et al. [51] | Detect anger towards self, others, and objects.                  | Weka implementation: SVM, J48, Naive Bayes | Anger towards self Anger towards others Anger towards objects                      | Jira issues   | Cross-validation: ✓ Data augmentation: ✗ Class balancing: ✗ Hyperparameter optimization: ✓ |
| Egelman et al. [7]       | Detect the feelings pushback in code review, i.e., the perception of unnecessary interpersonal conflicts in code review while a reviewer is blocking a change request. | Logistic Regression Model           | Interpersonal conflict Feeling that acceptance was withheld for too long Reviewer asked for excessive changes Feeling negative about future code reviews Frustration | Code reviews  | Cross-validation: ✗ Data augmentation: ✗ Class balancing: ✗ Hyperparameter optimization: ✗ |
| Raman et al. [4]        | Detect toxic language, i.e., hate speech and microaggressions.     | SVM                                 | Toxic Non-toxic                                                                    | GitHub issues | Cross-validation: ✓ Data augmentation: ✗ Class balancing: ✓ Hyperparameter optimization: ✓ |
| Sarker et al. [14]      | Evaluate different tools to detect toxicity.                      | Perspective API, STRUDEL Toxicty Detector, Deep Pyramid Convolutional Neural Networks, BERT with fast.ai, Hate Speech Detection | Toxic Non-toxic                                                                    | Code reviews, Gitter messages | Cross-validation: ✓ Data augmentation: ✗ Class balancing: ✓ Hyperparameter optimization: ✓ |
| Cheriyan et al. [5]     | Detect and classify offensive language, i.e., communication that contains gutter language, swearing, racist, or offensive content. | Random Forest, SVM, BERT            | Offensive Non-offensive                                                              | GitHub, Gitter, Slack, Stack Overflow | Cross-validation: ✗ Data augmentation: ✗ Class balancing: ✓ Hyperparameter optimization: ✓ |
| Sarker et al. [15]      | Detect toxicity, i.e., SE conversations that include offensive, name calling, insults, threats, personal attacks, flirtations, reference to sexual activities, and swearing or cursing. | Decision tree, Logistic Regression, SVM, Random Forest, Gradient-Boosted Decision Trees, LST, BiLSTM, GRU, DPCNN, BERT | Toxic Non-toxic                                                                    | Code reviews Gitter Stack Overflow | Cross-validation: ✓ Data augmentation: ✗ Class balancing: ✓ Hyperparameter optimization: ✓ |
| Our work              | Detect incivility, i.e., features of discussion that convey an unnecessarily disrespectful tone toward the discussion forum, its participants, or its topics in code review and issue discussions. | CART, KNN, Logistic Regression, Naive Bayes, Random Forest, SVM, BERT | Tone-bearing Non-tone-bearing Civil Uncivil                                         | Code reviews GitHub issues       | Cross-validation: ✓ Data augmentation: ✓ Class balancing: ✓ Hyperparameter optimization: ✓ |
non-tone-bearing class (formerly called technical by Ferreira et al. [1]) comprises emails and comments in which none of their sentences convey a mood or style of expression [1]. On the contrary, tone-bearing (previously called non-technical by Ferreira et al. [1]) code review emails or issue comments are those in which at least one sentence expresses a tone-bearing discussion feature (TBDF). The datasets use the concept of TBDF to indicate "conversational characteristics demonstrated in a written sentence that convey a mood or style of expression" [1]. These TBDFs are further divided into four categories: (1) civil positive (e.g., humility and excitement), (2) civil neutral (e.g., apologies and friendly joke), (3) civil negative (e.g., sadness and oppression), and (4) uncivil (e.g., bitter frustration, impatience, mocking, or vulgarity) [1, 8]. In total, there are 1,365 non-tone-bearing emails and 168 tone-bearing emails in the code review dataset; there are 4,793 non-tone-bearing comments and 718 tone-bearing comments in the issues dataset.

Next, sentences in tone-bearing emails and comments were then further categorized as civil or uncivil. The uncivil class contains sentences that demonstrate at least one uncivil TBDF as defined in the dataset [1, 8]. Conversely, civil sentences are those that only contain civil TBDFs (either civil positive, civil neutral, or civil negative, as described above). There are 117 civil sentences and 276 uncivil sentences in the tone-bearing emails of the code review dataset and there are 353 civil sentences and 896 uncivil sentences in the tone-bearing comments of the issues dataset.

Our classification tasks are thus two-layered to mimic the human classification process in our previous work [1]: first, we aim to classify code review emails and issue comments into tone-bearing or non-tone-bearing (CT1); then, for tone-bearing contents, we aim to classify sentences into civil or uncivil (CT2). The goal to separate these two classification tasks is because in a concrete scenario in which open source contributors would use our classifiers to assess whether their comments are uncivil or not, first, we would detect if it is a tone-bearing text or not. If it is tone-bearing, then we would detect (in) civility.

3.2. Data Preprocessing

We followed a series of steps to reduce noise on the datasets described above. First, we considered sentences coded with both civil and uncivil TBDFs as uncivil data points. This is because the appearance of any uncivil expression can already signal an unhealthy discussion, regardless of the appearance of civil tones [1]; for example, “I don’t think ‘not fixing it because it’s not fixed yet’ is a good reason to keep things the way they are. But maybe that’s just me.” should be considered as an uncivil sentence for demonstrating impatience although a civil tone of humility is present. We then performed the following preprocessing steps:

1. We manually remove the source code (including variable names, function names, stack traces, etc.), words other than English, emojis, and GitHub username mentions (such as @username) from the text;
2. We automatically remove the header of code review emails, including the first line that follows the regex pattern “On (\.*\?) wrote:\n”;
3. We automatically remove greetings such as “Hi [person_name]” and statements such as “Reviewed by [person_name]” or “Tested by [person_name]”;
4. We automatically remove any signature statement that is in the following list of words: “warm regards”, “kind regards”, “regards”, “cheers”, “many thanks”, “thanks”, “sincerely”, “best”, “thank you”, “talk soon”, “cordially”, “yours truly”, “all the best”, “best regards”, “best wishes”, “looking forward to hearing from you”, “sincerely yours”, “thanks again”, “with appreciation”, “with gratitude”, and “yours sincerely”;
5. We automatically remove all reply quotes, usually represented by “<”;

Our classification tasks are thus two-layered to mimic the human classification process in our previous work [1]: first, we aim to classify code review emails and issue comments into tone-bearing or non-tone-bearing (CT1); then, for tone-bearing contents, we aim to classify sentences into civil or uncivil (CT2). The goal to separate these two classification tasks is because in a concrete scenario in which open source contributors would use our classifiers to assess whether their comments are uncivil or not, first, we would detect if it is a tone-bearing text or not. If it is tone-bearing, then we would detect (in) civility.

3.2. Data Preprocessing

We followed a series of steps to reduce noise on the datasets described above. First, we considered sentences coded with both civil and uncivil TBDFs as uncivil data points. This is because the appearance of any uncivil expression can already signal an unhealthy discussion, regardless of the appearance of civil tones [1]; for example, “I don’t think ‘not fixing it because it’s not fixed yet’ is a good reason to keep things the way they are. But maybe that’s just me.” should be considered as an uncivil sentence for demonstrating impatience although a civil tone of humility is present. We then performed the following preprocessing steps:

1. We manually remove the source code (including variable names, function names, stack traces, etc.), words other than English, emojis, and GitHub username mentions (such as @username) from the text;
2. We automatically remove the header of code review emails, including the first line that follows the regex pattern “On (\.*\?) wrote:\n”;
3. We automatically remove greetings such as “Hi [person_name]” and statements such as “Reviewed by [person_name]” or “Tested by [person_name]”;
4. We automatically remove any signature statement that is in the following list of words: “warm regards”, “kind regards”, “regards”, “cheers”, “many thanks”, “thanks”, “sincerely”, “best”, “thank you”, “talk soon”, “cordially”, “yours truly”, “all the best”, “best regards”, “best wishes”, “looking forward to hearing from you”, “sincerely yours”, “thanks again”, “with appreciation”, “with gratitude”, and “yours sincerely”;
5. We automatically remove all reply quotes, usually represented by “<”;

Our classification tasks are thus two-layered to mimic the human classification process in our previous work [1]: first, we aim to classify code review emails and issue comments into tone-bearing or non-tone-bearing (CT1); then, for tone-bearing contents, we aim to classify sentences into civil or uncivil (CT2). The goal to separate these two classification tasks is because in a concrete scenario in which open source contributors would use our classifiers to assess whether their comments are uncivil or not, first, we would detect if it is a tone-bearing text or not. If it is tone-bearing, then we would detect (in) civility.

3.2. Data Preprocessing

We followed a series of steps to reduce noise on the datasets described above. First, we considered sentences coded with both civil and uncivil TBDFs as uncivil data points. This is because the appearance of any uncivil expression can already signal an unhealthy discussion, regardless of the appearance of civil tones [1]; for example, “I don’t think ‘not fixing it because it’s not fixed yet’ is a good reason to keep things the way they are. But maybe that’s just me.” should be considered as an uncivil sentence for demonstrating impatience although a civil tone of humility is present. We then performed the following preprocessing steps:

1. We manually remove the source code (including variable names, function names, stack traces, etc.), words other than English, emojis, and GitHub username mentions (such as @username) from the text;
2. We automatically remove the header of code review emails, including the first line that follows the regex pattern “On (\.*\?) wrote:\n”;
3. We automatically remove greetings such as “Hi [person_name]” and statements such as “Reviewed by [person_name]” or “Tested by [person_name]”;
4. We automatically remove any signature statement that is in the following list of words: “warm regards”, “kind regards”, “regards”, “cheers”, “many thanks”, “thanks”, “sincerely”, “best”, “thank you”, “talk soon”, “cordially”, “yours truly”, “all the best”, “best regards”, “best wishes”, “looking forward to hearing from you”, “sincerely yours”, “thanks again”, “with appreciation”, “with gratitude”, and “yours sincerely”;
5. We automatically remove all reply quotes, usually represented by “<”;

Our classification tasks are thus two-layered to mimic the human classification process in our previous work [1]: first, we aim to classify code review emails and issue comments into tone-bearing or non-tone-bearing (CT1); then, for tone-bearing contents, we aim to classify sentences into civil or uncivil (CT2). The goal to separate these two classification tasks is because in a concrete scenario in which open source contributors would use our classifiers to assess whether their comments are uncivil or not, first, we would detect if it is a tone-bearing text or not. If it is tone-bearing, then we would detect (in) civility.
6. We automatically remove stop words and punctuation; we perform stemming on each remaining word. This step is only performed for the classical machine learning classifiers.

3.3. Feature Extraction for Classical ML Classifiers

This step consists in extracting features to detect incivility in code review emails/issues comments and sentences using supervised techniques. We created two sets of features for both classification tasks, namely textual features and conversational features, which were inspired by the work of Arya et al. [23] and adapted to our context.

Textual features: We consider n-grams as textual features. First, we perform text vectorization by transforming each word of the text into a feature, based on the absolute term frequencies. Second, we use 2-grams that represent the appearance of tokens sequences. Then, we use weighted TF-IDF to transform and adapted to our context.

c. i.e., the frequency of words and 2-grams in the text are multiplied by their inverse document frequency [23]. We tune each model by using only the n-gram configuration that yields the best result.

Conversational features describe the participants, length, structural, and temporal attributes of code review and issue discussions. Each one of these features is described in Table 2, along with the classification tasks in which they are used. The conversational features include the following categories:

- Participant features include features describing the discussion participants, i.e., authors who wrote the code review email or issue comment as well as if the author is a reviewer or a developer. Participant features are relevant in our context because our previous work identified that emails/comments sent by maintainers exhibit different causes of incivility than those sent by other developers [1]. Furthermore, maintainers send more uncivil code review emails than developers [1].

- Length features concern the length of emails, comments, or sentences. These specific features indicate length in terms of the number of characters in the email/comment or the sentence, as well as the relative number of words with respect to other emails, comments, or sentences. We included these length features because our previous work found that code review discussions with tone-bearing discussion features tend to be longer and have more uncivil than civil emails [1].

- Structural features describe the location of an email, comment, or sentence in relation to the entire email thread, issue thread, or the current email/comment itself. As shown by Ferreira et al. [8], uncivil comments tend to emerge during the discussion in issue discussions, at various locations in the discussion thread. Hence, we hypothesize that structural features are pertinent to incivility classification.

- Temporal features concern the time that the email/comment was sent with respect to the immediately previous and next email/comment as well as the beginning and the end of the email/issue thread. Egelman et al. [7] has shown that long reviewing time and long time working towards the solution of the problem are effective for flagging that code review authors have a feeling of pushback. Hence, we hypothesize that these features are suitable in our context because a long reply time might lead to incivility.

3.4. Data Augmentation and Class Balancing

Our datasets, especially the ones for civil and uncivil classification, are relatively small. To increase the training set and to boost performance for both classification tasks, we used the Easy Data Augmentation (EDA) [53] techniques to augment the current datasets; the EDA techniques are known to contribute to performance gains of classifications when the dataset is small [53]. Additionally, the datasets we use are highly imbalanced, skewing toward tone-bearing emails/comments and uncivil sentences. Machine learning classifiers are well known for underperforming when the data is skewed toward one class [54, 55]. To address this issue, we explored and evaluated three class balancing techniques that we describe in this section.

3.4.1. Easy Data Augmentation Techniques (EDA)

In this study, we use the Easy Data Augmentation (EDA) Techniques [53] to increase the size of our datasets. EDA is composed of four operations:

- Synonym Replacement (SR) consists of randomly choosing n words (excluding stop words) from the text and replacing them with a random synonym.
- Random Insertion (RI) consists of finding a synonym of a random word in the text (excluding stop words) and inserting the synonym in a random position in the text. This process is repeated n times.
- Random Swap (RS) is when two words are randomly chosen and their positions are swapped. This is repeated n times.
- Random Deletion (RD) consists of randomly removing words in a sentence with probability p.

To find a synonym to perform the SR and RI operations, we use the NLTK WordNet corpus and the function synsets(word) to lookup for the word’s synonym. Furthermore, we use the NLTK’s list of English stopwords [56] to exclude stopwords from the text. To mitigate the threat of long texts having more words than short texts, Wei and Zou [53] suggest varying the number of words n for SR, RI, and RS based on the text length l with the formula n = αl, where α is a hyperparameter that indicates the percentage of words in a text to be changed. Furthermore, for each original email/comment/sentence, it is possible to generate n_aug augmented emails/comments/sentences. We evaluated different combinations of hyperparameters to augment the training set (see our replication package for details). The hyperparameter search space was chosen based on the training set size and thresholds that have resulted in high performance in previous work by Wei and Zou [53]. The hyperparameter tuning process is described in detail in Section 3.5.

3.4.2. Class Balancing Techniques

To address the class imbalance problem of our dataset [54], we explored three class balancing techniques: random oversampling, random undersampling, and Synthetic Minority Over-sampling Technique (SMOTE). We implemented these techniques using the Python library imblearn to compare their results when answering RQ1. The class balancing techniques are applied after the datasets are augmented by EDA.
• **Random oversampling** aims at taking random samples for the minority class and duplicating them until it reaches a size comparable to the majority class [57].

• **Random undersampling** selects random samples from the majority class and removes them from the dataset until it reaches a size comparable to the minority class [55].

• **SMOTE** is a method in which the minority class is over-sampled by creating new samples and the majority class is undersampled [58].

3.5. Training and Evaluating the Classifiers

3.5.1. Classical Machine Learning Models

We consider six classical classifiers to detect incivility in code review and issues discussions (RQ1). The classifiers were implemented using the scikit-learn Python library.

- **Classification and Regression Tree (CART)** is a binary tree that aims at producing rules that predict the value of an outcome variable [21].

- **k-nearest neighbors (KNN)** assumes that similar data points are close to each other. Hence, the algorithm relies on distance metrics for classification. The resulting class is the one that has the nearest neighbors [18].

- **Logistic Regression (LR)** uses a logistic function to model the dependent variable. The goal of the algorithm is to find the best fitting model to describe the relationship between the dependent and independent variable [18].

- **Naive Bayes (NB)** is a probabilistic classifier based on the Bayes theorem. It assumes that the presence (or absence) of a particular feature of a class is unrelated to the presence (or absence) of any other feature [20]. In this work, our text classification is performed using the Multinomial Naive Bayes model that has improved performance over the Bernoulli model for text classification [59].

- **Random Forest (RF)** is a group of decision trees whose nodes are defined based on the training data [18]. The most frequent label found by the trees from the forest is the resulting class [23].

- **Support Vector Machine (SVM)** is a linear model that creates a hyperplane separating the data into two classes [60]. During the training process, we performed nested cross-validation with grid search [61] to test a combination of hyperparameters and evaluate the models’ performance. Specifically, we first split the dataset into train and test sets in the outer stratified 5-fold cross-validation for model evaluation. The training set obtained from the outer cross-validation is then further split into training (for training the models) and validation (for selecting the best hyperparameters) sets in the inner stratified 5-fold cross-validation. The search space for the hyperparameters of each model, which were defined according to the literature, can be found in our replication package. In each outer cross-validation fold, we used the performance metrics presented in Section 3.5.3 to evaluate the performance of the classifier.

3.5.2. BERT Base Model

We use the uncased BERT base model, pretrained on the English language [62], to detect incivility. We chose to use the uncased model (i.e., the model does not make a difference between “english” and “English”) because the case information is not relevant to our classification tasks. Furthermore, due to the large number of parameters in this model (approximately 110 million parameters), we did not pretrain it from scratch to reduce the risk of overfitting [63]. The classification task is done using the Transformers PyTorch library with AutoModelForSequenceClassification [64], which has a classification head.

We split the input dataset into train, test, and evaluation datasets in a 70-15-15 ratio stratified along the labels. To optimize BERT’s hyperparameters, we run Bayesian optimization [65] with 50 trials for each one of the EDA parameter settings (see Section 3.4.1), i.e., eight times. BERT’s hyperparameter optimization was done using the hyperparameter_search() function [66] from the Trainer class with optuna as the backend. The Bayesian optimization takes in the training and evaluation sets as inputs; the former is used to train the model with different hyperparameters and the latter is used to select the best hyperparameters. The search space for the hyperparameters can be found in our replication package.

After obtaining the best set of hyperparameters, we perform a 5-fold cross-validation to train and test BERT. For that, we use the Trainer [67] class from the Transformers library. The training adopts an epoch evaluating strategy, i.e., evaluating BERT’s performance at the end of each epoch using the

| Feature type | Classification task | Feature name | Description | Values |
|--------------|---------------------|--------------|-------------|--------|
| Participant  | CT1, CT2            | AUTHOR_ROLE  | Email author’s role in the Linux kernel. We first group identities that have the same names or the same email addresses. The author is considered a maintainer if one of those identities appears in the MAINTAINERS file [52], and a developer otherwise. | Maintainer, Developer |
| Length       | CT1                 | CHAR_TEXT    | Number of characters in the email/comment | [0, 1] |
|              | CT1                 | LEN_TEXT     | Number of words in the email/comment divided by that of the longest email/comment in the thread. | [0, 1] |
|              | CT1                 | CHAR_SENT    | Number of characters in the sentence. | [0, 1] |
|              | CT1                 | LEN_SENT_T   | Number of words in sentence divided by that of longest sentence in the thread. | [0, 1] |
|              | CT1                 | LEN_SENT_C   | Number of words in sentence divided by that of longest sentence in the email/comment. | [0, 1] |
| Structural   | CT1                 | POS_TEXT_T   | Position of sentence in email/comment divided by the number of sentences in email/comment. Sentences are identified based on the following regular expression: (\?|\!|\?\?). | [0, 1] |
|              | CT1                 | POS_SENT_T   | Position of sentence in thread divided by the number of sentences in thread. Sentences are identified based on the following regular expression: (\?|\!|\?\?). | [0, 1] |
| Temporal     | CT1, CT2            | TIME_FIRST_COMMENT | Time from first email/comment to current email/comment divided by the total time of the thread. | [0, 1] |
|              | CT1, CT2            | TIME_TEXT_LAST | Time from current email/comment to last email/comment divided by the total time of the thread. | [0, 1] |
|              | CT1, CT2            | TIME_PREVIOUS_COMMENT | Time from previous email/comment to current email/comment divided by the total time of the thread. | [0, 1] |
|              | CT1, CT2            | TIME_TEXT_NEXT | Time from current email/comment to next email/comment divided by total time of the thread. | [0, 1] |

Note: CT1 = classification task 1 on non-tone-bearing and tone-bearing emails, CT2 = classification task 2 on civil and uncivil sentences.
performance metrics described in Section 3.5.3.

3.5.3. Performance metrics

To compare the performance of our classifiers, we evaluate their performances using the confusion matrix: TP is the number of true positives, FN is the number of false negatives, FP is the number of positives, and TN is the number of true negatives.

Based on this matrix, we first computed two well-known metrics, namely precision and recall [71]. The precision of a given target class (i.e., non-tone-bearing, tone-bearing, civil, or uncivil) is defined by the ratio of data points (emails, comments, or sentences) for which a given classifier correctly predicted the target class; i.e., precision = TP/(TP + FP). The precision value is always between 0 (worst possible score) and 1 (perfect score). In each classification task of our experiments, we first calculated the precision in each class, then the macro-average metric across both classes to represent the overall precision.

The recall of a given target class is the ratio of all data points with the target class that a given classifier was able to correctly find; i.e., recall = TP/(TP + FN). The recall value is also between 0 (worst possible score) and 1 (perfect score). Similar to precision, we calculated per-class and macro-averaged recall metrics for each classification task.

Then, to have a single value representing the goodness of the models, we computed the F1-score, which is the harmonic mean of precision and recall, i.e., \( F1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \). The F1-score is independent of the number of true negatives and it is highly influenced by classes labeled as positive. The F1-score is always between 0 (lowest precision and lowest recall) and 1 (highest precision and highest recall). In our experiments, we calculated per-class and macro-averaged F1 metrics.

Finally, we computed the Matthews Correlation Coefficient (MCC) [72], which is a single-value classification metric that is more interpretable and robust to changes in the prediction goal [73], because it summarizes the results of all four quadrants of a confusion matrix [74]. The MCC metric is calculated as \( \text{MCC} = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \) and its value is always between -1 (worst possible score) and 1 (perfect score), with 0 suggesting that the model’s performance is equal to random prediction. To calculate TP, TN, FP, and FN for MCC, we considered civil and non-tone-bearing as positive classes and uncivil and tone-bearing as negative classes, although this selection does not influence the end results. To be able to compare the MCC scores with the other performance metrics (i.e., macro-averaged precision, recall, and F1-score), we normalized the MCC values to the \([0, 1]\) interval. Therefore, the normalized MCC (nMCC) is defined by \( nMCC = \frac{MCC + 1}{2} \) [73, 75]. We also use nMCC as the primary metric during hyperparameter evaluation (Section 3.5.1).

3.6. Experimental Design to Answer the RQs

3.6.1. Detecting Incivility (RQ1)

In RQ1, we have two classification tasks for each dataset, i.e., (1) classification of code review emails and issue comments into non-tone-bearing and tone-bearing and (2) classification of code review sentences and issue sentences into civil and uncivil.

For each classification task and for each dataset, we compare BERT with six classical machine learning models. We assess BERT with three class balancing conditions: no class balancing, random oversampling, and random undersampling. It was not possible to run SMOTE with BERT because the current SMOTE implementations need to convert textual features to numerical vectors (via tokenization to a form suitable for SMOTE) [76] and cannot be applied to textual features that are used for BERT. The classical machine learning models are assessed with three class balancing techniques: random oversampling, random undersampling, and SMOTE. Thus, for each classification task and each dataset, we have 21 experimental conditions (7 classifiers * 3 balancing techniques). The hyperparameters are tuned separately for each combination of classification task, dataset, classifier, and class balancing technique. In this paper, we report the results related to hyperparameters that had the best-averaged nMCC score across all outer folds.

3.6.2. Adding the Previous Email or Comment When Detecting Incivility (RQ2)

To answer RQ2, we add the previous email/comment from the same email or issue thread and concatenate it with the original email/comment to create a new dataset. For this RQ, we only focus on BERT since it provided the best performance in the initial detection. We use the hyperparameters, the EDA parameter configuration, and the imbalance handling technique that obtained the best nMCC score in RQ1 for BERT in each classification task for each dataset. To analyze if adding the previous email/comment helps to detect incivility, we compare the difference between the performance scores of RQ1 and RQ2. Hence, for each performance metric (PM), \( \Delta PM = PM_{R2} - PM_{R1} \). We consider that the previous email/comment does not help if \( \Delta PM < 0 \) and that it helps if \( \Delta PM > 0 \).

3.6.3. Detecting Incivility in a Cross-Platform Setting (RQ3)

To answer RQ3, we train our models and test them in the other dataset for each classification task; i.e., we train our classifiers on the code review dataset and test them on the GitHub issues dataset, and vice versa. For that, we use the hyperparameters, the EDA parameters configuration, and the imbalance handling techniques that obtained the best nMCC score in RQ1 for the dataset used to train the classifiers. Because the best hyperparameters can differ among the five folds for the classical machine learning models, we pick the hyperparameters that were chosen most frequently across all five folds. If there is a tie between two sets of hyperparameters, we then choose the hyperparameter from the fold that had the highest nMCC score.

3.6.4. Analyzing misclassified cases (RQ4)

To answer RQ4, we assess the tone-bearing discussion features (TBDFs) [1] that the analyzed models misclassified. The misclassified sentences were extracted from the test sets in the outer fold cross-validation (see Section 3.5.1). Since the non-tone-bearing emails/comments are split into sentences and the sentence classification (CT2) depends on the tone that they demonstrate [1], we calculate the percentage of sentences that were misclassified by each classifier per TBDF in CT2.
4. Results

In this section, we present the results for each research question for the non-tone-bearing/tone-bearing and civil/uncivil classification tasks. We only report the results of the hyperparameters that yielded the best results.

4.1. Models’ Performance on Incivility Detection (RQ1)

4.1.1. Classification into Non-tone-bearing and Tone-bearing

Figure 2 presents the average performance scores for each experiment condition for the code reviews (left) and issues (right) datasets and Figure 3 shows the performance per target class.

For the code reviews dataset, BERT without class balancing and with random undersampling has the best performance compared to the classical classifiers, with $F_1 = 0.92$ and 0.94, and $nMCC = 0.91$ and 0.94, respectively. Classical machine learning models underperform to classify tone-bearing code review emails. The nMCC scores for the classical classifiers and for BERT with random oversampling (ranging from 0.50 to 0.64) are very low compared to BERT’s nMCC scores with random undersampling and without class balancing (0.94 and 0.91, respectively), showing that such classifiers are not effective to detect tone-bearing code review emails. Figure 3 (left) confirms this result, indicating that the tone-bearing class (red color) having overall lower precision and recall than the non-tone-bearing class (green color) for the underperforming classifiers. We also observe that among the classical classifiers, Random Forest (RF) achieved the highest precision ($≈0.7$) regardless of the class balancing technique, but with a low recall ($≈0.5$); on the contrary, Logistic Regression (LR) achieved the highest recall ($≈0.7$), but a relatively low precision ($≈0.6$).

For the issues dataset, BERT also performs better than the classical classifiers in all class balancing conditions. Similar to the code reviews dataset, BERT is able to precisely classify non-tone-bearing and tone-bearing issue comments ($precision ≈ 0.9$), finding a substantial number of issue comments ($recall ≈ 0.9$), and effectively classifying the non-tone-bearing and tone-bearing issue comments ($nMCC ≈ 0.9$), in all class balancing conditions. Furthermore, classical classifiers also underperform to classify tone-bearing issue comments. The nMCC scores range from 0.57 to 0.69 demonstrating that tone-bearing issue comments are not effectively detected with classical machine learning models (see Figure 3 (right)), except for RF with random oversampling and SVM with random undersampling, in which their precision metrics are better for the tone-bearing class ($precision_{RF} = 0.91$ and $precision_{SVM} = 1.0$) than the non-tone-bearing class ($precision_{RF} = 0.87$ and $precision_{SVM} = 0.87$); yet their recalls are very low for the tone-bearing class ($recall_{RF} = 0.07$, $recall_{SVM} = 0.03$).

It is surprising that BERT has a good performance overall even without any class balancing technique. This result is confirmed by Figure 3, which shows that even without a class balancing technique both non-tone-bearing and tone-bearing classes have a good F1-score for both the code reviews dataset ($tone−bearing = 0.84$ and $non−tone−bearing = 0.98$) and the issues dataset ($tone−bearing = 0.86$ and $non−tone−bearing = 0.98$).

4.1.2. Classification into Civil and Uncivil

Figure 4 illustrates the performance metrics for each experiment setting and for both the code reviews (left) and issues (right) datasets. Similarly, Figure 5 presents the performance metrics per target class for both datasets.

BERT is the best performing classifier regardless of the class balancing technique for the code reviews dataset. We observe that, similar to the non-tone-bearing/tone-bearing classification task, BERT has the best performance ($precision ≈ 0.9$, $recall ≈ 0.9$, $F_1 ≈ 0.9$, $nMCC ≈ 0.9$) to classify civil and uncivil code review sentences. However, classical machine learning techniques tend to perform better in the classification of civil/uncivil sentences than in non-tone-bearing/tone-bearing emails, with nMCC scores ranging from 0.58 to 0.77 (compared to between 0.50 and 0.64 for the non-tone-bearing/tone-bearing classification). Additionally, the classical models underperform when classifying civil code review sentences (see Figure 5 (left)). The Logistic Regression, Naive Bayes, and Random Forest classifiers have overall promising results though, with $precision ≈ 0.8$ and $recall ≈ 0.7$.

For the issue comments dataset, BERT is also the best classifier for detecting incivility regardless of which class balancing technique is used ($precision ≈ 0.9$, $recall ≈ 0.9$). Although classical techniques also tend to underperform when classifying civil issue sentences (see Figure 5 (right)), Logistic Regression and Random Forest have good precision ($≈ 0.71$ for LR and $≈ 0.75$ for RF) and recall ($≈ 0.72$ for LR and $≈ 0.67$ for RF) overall.

Summary RQ1: BERT performs better than the classical machine learning classifiers regardless of the class balancing technique for non-tone-bearing/tone-bearing and civil/uncivil classification in both datasets. Classical machine learning techniques tend to underperform when classifying the tone-bearing and civil classes.

4.2. Adding the Previous Email or Comment in Incivility Detection (RQ2)

Figure 6 (a) and (b) present the difference in performance metrics between the BERT results considering the previous email or comment (RQ2) and without (RQ1), for the two classifications respectively.

We found that adding the previous email/comment does not help to classify non-tone-bearing and tone-bearing code review emails and issue comments. We observe that, for both datasets when detecting Non-tone-bearing and tone-bearing contents, $APM$ is negative overall, with the tone-bearing class results having most drastically decreased when the previous email or comment is added (Figure 6 (a)). BERT’s performance on the code reviews dataset, more specifically, gets worse by $≈ −0.2$ for the tone-bearing class, having its precision decreased from 0.88 to 0.67 and its recall from 0.92 to 0.71. Similarly, on the issues dataset, BERT’s precision for tone-bearing comments decreased by 0.14, going from 0.92 to 0.78; and the recall decreased by 0.04, from 0.83 to 0.79.

Overall, adding the previous email/comment also does not help to classify civil and uncivil code review and issue
sentences. Our results show that the civil class results have significantly decreased, especially for the issues dataset (Figure 6 (b)). Although the precision did not change for the code reviews dataset, its recall decreased by 0.04, going from 0.92 to 0.88). For the issues dataset, the precision and recall decreased by 0.09 and 0.03, respectively.

**Summary RQ2:** Adding the previous code review email and issue comment makes the prediction worse for both non-tone-bearing/tone-bearing and civil/uncivil classification. The effect is stronger for the tone-bearing class.

### 4.3. Incivility Detection in a Cross-Platform Setting (RQ3)

#### 4.3.1. Classification into Non-tone-bearing and Tone-bearing

Figure 7 (a) presents the performance for non-tone-bearing and tone-bearing classification in the cross-platform setting. The classifiers’ performances degraded to classify tone-bearing
discussions in a cross-platform setting. When training our classifiers on the code reviews dataset and testing them on the issues dataset, we observe that BERT is the best classifier, with a nMCC score of 0.62. Our results show that the classifiers’ performances are not satisfactory in precisely classifying tone-bearing discussions (red color), with precision scores ranging from 0.10 (SVM) to 0.39 (BERT and RF). Interestingly, the Logistic Regression (LR) and Naïve Bayes (NB) classifiers can retrieve a significant percentage of tone-bearing discussions ($\text{recall}_{\text{LR}} = 0.88$, $\text{recall}_{\text{NB}} = 0.85$); even though they fail to precisely classify such cases ($\text{precision}_{\text{LR}} = 0.16$, $\text{precision}_{\text{NB}} = 0.17$). We also observe a similar pattern when training on the issues dataset and testing on the code reviews dataset. The MCC scores ranged from 0.50 (KNN and SVM) to 0.62 (BERT). In this setting, BERT is more precise ($\text{precision} = 0.51$) than in the previous setting ($\text{precision} = 0.39$) at classifying tone-bearing discussions, yet the coverage is lower ($\text{recall} = 0.10$ in this setting, versus original $\text{recall} = 0.30$). Surprisingly, the Logistic Regression classifier has a similar recall for both target classes ($\text{recall}_{\text{non–tone–bearing}} = 0.61$, $\text{recall}_{\text{tone–bearing}} = 0.64$).
4.3.2. Classification into Civil and Uncivil

Figure 7 (b) presents the results for the civil and uncivil classification in the cross-platform setting. The classifiers’ performances are also degraded to classify civil sentences in a cross-platform setting. When training on code reviews and testing on issues, we observe that all classifiers are able to precisely classify uncivil discussions with precision \( \approx 0.7 \) with good coverage (recall \( \approx 0.8 \)). However, all classifiers have low precision (ranging from 0.35 to 0.53) and low recall (ranging from 0.29 to 0.56). When training on issues and testing on code reviews, we observe the same pattern as in the aforementioned configuration, i.e., all classifiers can precisely classify the uncivil class (precision \( \approx 0.8 \)) with a recall \( \approx 0.7 \). Interestingly, the Logistic Regression classifier has a recall higher for the civil class (recall \( \approx 0.82 \)) than the uncivil class (recall \( \approx 0.42 \)).

Summary RQ3: None of the classifiers are effective to classify tone-bearing and civil discussions in a cross-platform setting. However, all classifiers were able to perform well when classifying the non-tone-bearing and uncivil classes.
4.4. Analysis of misclassified cases (RQ4)

4.4.1. Misclassified TBDFs per incivility classifier

Contrary to the classical machine learning models, BERT can correctly classify more than 70% of the sentences for all civil and uncivil TBDFs for the code reviews and issues datasets. As Figure 8 shows, in the code reviews dataset BERT mostly misclassifies sentences demonstrating the civil TBDFs friendly joke (28.57%), commanding (22.22%), and sadness (14.29%) and the uncivil TBDFs irony (16.67%), threat (15.28%), and vulgarity (15.28%). Although BERT misses 22.22% of sentences expressing commanding, the classical machine learning models are worst in classifying this TBDF (varying from 44.44% for LR and SVM to 88.89% for KNN). Concerning the friendly joke TBDF, although SVM is as good as BERT (both models miss 28.57% of sentences with this TBDF), the other machine learning models misclassify from 42.86% (for CART, LR, and NB) to 71.43% (for KNN AND RF) sentences. The same happens for commanding and sadness. Interestingly, the classical machine learning models perform better than BERT to identify the uncivil TBDFs that BERT misses in the code reviews dataset. That is, KNN and SVM misclassify 14.29% of sentences (instead of 16.67% for BERT) demonstrating irony; CART and KNN misclassify 12.50% of sentences (instead of 15.38% for BERT) showing threat; CART, LR, and SVM do not miss any sentence demonstrating vulgarity, while BERT misses 15.38%.

Note that, as demonstrated by Ferreira et al., the TBDFs named confusion, criticizing oppression, dissatisfaction, and expectation were only encountered in issue discussions [8] and not in code review discussions [1]. For the issues dataset, BERT mainly misclassifies the following civil TBDFs: friendly joke (26.09%), criticizing oppression (17.65%), considerateness (13.56%), and dissatisfaction (12.90%). None of the uncivil TBDFs had more than 10% of misclassified sentences. Furthermore, none of the classical machine learning models can classify the aforementioned TBDFs better than BERT.

Summary RQ4 (part 1): Irony and vulgarity are are among the most difficult uncivil TBDFs to classify by BERT, while friendly joke and commanding are the most difficult civil TBDFs.

4.4.2. BERT’s misclassified TBDFs considering the previous email or comment

Figure 9 presents the percentage of misclassified sentences per TBDF for BERT considering the previous email or comment. Surprisingly, for the code reviews dataset, commanding, friendly joke, sadness, and threat that were most frequently misclassified by BERT without the previous email or comment (see Section 4.4.1) now have 100% of the sentences correctly classified. Irony has a slightly decreased number of misclassified sentences, going from 16.67% to 14.29%, and vulgarity has an increased number of misclassified sentences, by 9.62% (from 15.38% to 25%). Additionally, appreciation and excitement, sincere apologies, impatience, and mocking were more frequently misclassified considering the previous email or comment than without, increasing the number of misclassified sentences by 9.82%, 11.11%, 7.14%, and 7.98%, respectively.

For the issues dataset, the number of misclassified sentences was decreased by 16.09% for the friendly joke TBDF and by 1.67% for the considerateness TBDF. Criticizing oppression has an increased number of misclassified sentences by 11.76%, and dissatisfaction by 0.29%. Furthermore, expectation, sadness, and sincere apologies were the most impacted TBDFs, increasing the number of misclassified sentences by 8.38%, 22.62%, and 14.16%, respectively.

Summary RQ4 (part 2): When the previous email or comment is added, even though the accuracy of detecting some TBDFs (e.g., friendly joke) by BERT has improved, the accuracy of many other TBDFs has deteriorated.

4.4.3. Misclassified TBDFs in cross-platform settings

When training on code reviews and testing on issues, BERT misclassifies more than 50% of the sentences demonstrating confusion, dissatisfaction, oppression, criticizing oppression, commanding, considerateness, and sadness (see Figure 10). It is expected that BERT’s performance is decreased for confusion, criticizing oppression, dissatisfaction, and expectation, since those TBDFs are not present in the code reviews dataset; hence, BERT never saw examples of these TBDFs in the training set. Interestingly, in this setting, BERT classifies all instances of hope to get feedback correctly, and it misses up to 16%; name calling (7.14%), annoyance and bitter frustration (11.09%), impatience (13.58%), and sincere apologies (16.13%). The classical machine learning models tend to misclassify more sentences than BERT, except for the vulgarity with KNN, and irony and mocking with LR.
5. Discussion

5.1. Synthesizing the key results

Our results show that BERT performs better than classical machine learning models in both non-tone-bearing/tone-bearing and civil/uncivil classification on code review emails and issue comments, with a F1-score higher than 0.9. This result is similar to the ones found in the literature for the classification of sentiments [39, 40, 41] and offensive language [5] in different software engineering artifacts (such as Stack Overflow posts, GitHub issues, API reviews, Jira issues, Gerrit code reviews, Gitter, and Slack). Hence, this paper contributes to the literature by demonstrating that BERT can also be used to classify incivility in code review emails and issue discussions. Furthermore, our results demonstrate that classical machine learning techniques tend to underperform when classifying tone-bearing code review emails and issue comments and civil sentences in both datasets. Since BERT has a F1-score greater than 0.90 when identifying both of these target classes, it is unclear what are the cases that classical machine learning models miss and that BERT does not.

However, we found that adding the previous code review email and issue comment makes the prediction of non-tone-bearing/tone-bearing code review emails and issue comments and civil/uncivil sentences worse, if not unchanged. This result echoes Murgia et al. [16], which demonstrated that the previous discussion does not help when classifying emotions in issue comments. But at the same time, this result is counterintuitive to us, since based on our experience of manually classifying incivility in code reviews and issue discussions [1, 8], we would expect that adding some context would improve the classifiers’ performance. One explanation for this is that such conversations are not “flat” or “linear”; i.e., the context is more complex than the immediate previous email. We plan to examine ways to capture this complexity in future work.

Finally, we found that the classifiers’ performance degraded in a cross-platform setting, with BERT still being the best-performing model with F1 and nMCC scores below 0.7. Similarly, Qiu et al. [46] found that classifiers’ performance degraded when training on toxic issues and code review comments and testing on pushback in code reviews and vice versa. While performance degradation is expected, BERT’s classification is still way better than random, especially for the non-tone-bearing/tone-bearing and civil/uncivil classification.
bearing and uncivil classes. However, whether this performance is satisfactory in practice needs future investigation.

5.2. Comparison with toxicity detectors

While civility and toxicity are different concepts, we examined two state-of-the-art toxicity detectors on our datasets to see their performance: (1) an SVM-based model STRUDEL [4] and (2) a BERT-based model ToxiCR [15]. Particularly, we retrained and evaluated the two pipelines with our dataset for both non-tone-bearing/tone-bearing and civil/uncivil classification tasks (CT1 and CT2, respectively). We found that STRUDEL achieved a macro-average F1 of 0.62 for CT1 and 0.72 for CT2 for the code review dataset, and 0.67 for CT1 and 0.68 for CT2 for the issue discussion dataset. This performance is better than our SVM model but is inferior to our BERT pipeline. Moreover, the BERT-based ToxiCR resulted in a macro-average F1 of 0.57 (CT1) and 0.68 (CT2) for the code review dataset, and 0.64 (CT1) and 0.73 (CT2) for the issue discussion dataset. This performance is inferior than our BERT pipeline. These results indicate that civility detection is a unique problem and our pipeline that incorporate data augmentation and class balancing can help achieve better results.

5.3. Lessons learned

Based on our results, we provide insights into the lessons learned for using automated incivility detection techniques.

**Lesson 1:** Adding the previous email/comment helps to accurately classify challenging TBDFs. However, this approach should not be used to classify more straightforward TBDFs.

Our results suggest that some TBDFs are more challenging to be classified than others. For such cases, adding the previous email/comment is a way to mitigate the problem. For example, in the code review dataset, friendly joke, commanding, sadness, and threat can only be accurately classified with the previous email/comment. Other TBDFs seem to be more straightforward to be classified, such as appreciation and excitement, sincere apologies, impatience, and mocking. Irony and vulgarity are still challenging to be classified and may require TBDF-specific classifiers.

**Lesson 2:** If a GPU is not available to train BERT, Random Forest can be used to precisely classify the code review dataset and SVM can be used for the issues dataset.

We found that BERT is the best model to identify incivility in open source discussions with high precision and high recall. However, BERT is a computationally expensive algorithm that requires a GPU. If a GPU is not available, for the code review dataset, the Random Forest model is the best option being able to precisely classify 75% of the data (although it will miss 48% of the cases). For the issues dataset, SVM is a good compromise with a precision of 0.93. Depending on the scenario, recall might be more important than precision. In that case, the Logistic Regression can be used for the code review dataset (recall = 0.68) and the issues dataset (recall = 0.75).

**Lesson 3:** It is feasible to use BERT to classify non-tone-bearing and uncivil code review emails and issue comments in a cross-platform setting when a manually annotated gold standard is not available.

Our results show that, in a cross-platform setting, BERT can be used to classify non-tone-bearing discussions with a precision and recall greater than 0.90. For the uncivil class, BERT’s precision and recall are greater than 0.75. Thus, it can be used in certain cross-platform detection use cases.

6. Threats to Validity

We discuss threats to the study validity [77] as follows.

**Construct Validity.** The incivility dataset from Ferreira et al. [1, 8] might contain noise (such as the source code, words other than English, special characters, etc.) that can affect the models’ performance. To mitigate this threat, we followed strict steps to preprocess the text (Section 3.2). Hence, we expect to have removed the noise in the data. Furthermore, the set of features used for the classical models might not represent all confounding factors in incivility. We minimize this threat by adopting the features from a previous work focused on characterizing sentences in issue discussions [23]. Finally, when assessing if the contextual information helps to detect incivility, the presence of civil or uncivil words and non-tone-bearing or tone-bearing words in the previous code review email/issue comment can affect the models’ performance. To mitigate this threat, we computed the number of previous emails and comments that are non-tone-bearing → tone-bearing and civil → uncivil and vice versa. For CT1, we found up to 7.76% code review emails and up to 8.05% issues comments in this situation. For CT2, we found up to 5.36% code review sentences and up to 12.68% of issue sentences in this situation. Given the relatively low number of datapoints in such a situation, we think that our results will not be highly affected by that.

**Internal Validity.** The models might overfit due to the small number of labeled datapoints. To address this problem, we implemented four data augmentation techniques with eight combinations of hyperparameters to ensure optimal results. Additionally, the imbalance in the datasets may lead to poor performance. To minimize this threat, we compared three class balancing techniques and assessed their performance. Finally, the choice of hyperparameters might affect the results. For that, we did hyperparameter optimization on all seven models using search spaces defined in the literature. Additionally, our model evaluation is based on the average metric values of a 5-fold cross-validation.

**Conclusion Validity.** All our validations (either to find the best hyperparameter or to compare the models) are based on the nMCC metric, which is known to be more interpretable and to have more robust results than other performance metrics.
External Validity. Our incivility classifiers are limited to code reviews of rejected patches of the Linux Kernel Mailing List and GitHub issues locked as too heated. Hence, our results may not be generalizable to other software engineering communication artifacts; this includes the results of cross-platform performance. Concerning the features used by the classical techniques, we have experience with incivility studies and we manually coded the data in our previous work [1, 8]. Hence, we were able to assess if the features are accurate to the incivility domain. Finally, our results are confined to the models implemented in this study. It is unknown if other models that would perform better for incivility classification.

7. Conclusion

Open source communities have developed mechanisms for handling uncivil discourse. However, the current mechanisms require considerable manual effort and human intervention. Automated techniques to detect uncivil conversations in open source discussions can help alleviate such challenges. In this paper, we compared six classical machine learning techniques with BERT when detecting incivility in open source code review and issue discussions. Furthermore, we assessed if adding the previous email/comment improves the classifiers’ performance and if the seven classifiers can be used in a cross-platform setting to detect incivility. In our analysis, we identified BERT as the best model to detect incivility in both code review and issue discussions. Furthermore, our results show that classical machine learning models tend to underperform when classifying tone-bearing and civil conversations. We also found that adding the previous email/comment does not improve BERT’s performance and that the classifiers’ performance degraded in a cross-platform setting. Finally, we provide three insights on the discussion features that the classifiers misclassify when detecting incivility. These insights will help future work that aims at leveraging discussion features in automated incivility detection applications, as well as improving cross-platform incivility detection performance.

Acknowledgements

The authors would like to thank Calcul Québec for the computing hardware that enabled them to run the experiments of this study. The authors also thank the Natural Sciences and Engineering Research Council of Canada for funding this research through the Discovery Grants Program [RGPIN-2018-04470].

References

[1] I. Ferreira, J. Cheng, B. Adams. The “shut the f** k up” phenomenon: Characterizing incivility in open source code review discussions, Proceedings of the ACM on Human-Computer Interaction 5 (CSCW2) (2021) 1–35.
[2] C. Miller, S. Cohen, D. Klug, B. Vasilescu, C. Kästner, "did you miss my comment or what?" understanding toxicity in open source discussions (2022).
[3] K. D. A. Carillo, J. Marsan, "the dose makes the poison"-exploring the toxicity phenomenon in online communities (2016).
[4] N. Raman, M. Cao, Y. Tsvetkov, C. Kästner, B. Vasilescu, Stress and burnout in open source: Toward finding, understanding, and mitigating unhealthy interactions, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER ’20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 57–60.
[5] J. Cheriyann, B. T. R. Savarimuthu, S. Cranefeld, Towards offensive language detection and reduction in four software engineering communities, in: Evaluation and Assessment in Software Engineering, 2021, pp. 254–259.
[6] T. Davidson, D. Warmsley, M. Macy, I. Weber, Automated hate speech detection and the problem of offensive language, in: Proceedings of the international AAAI conference on web and social media, Vol. 11, 2017, pp. 512–515.
[7] C. D. Egelman, E. Murphy-Hill, E. Kammer, M. M. Hodges, C. Green, C. Jaspan, J. Lin, Predicting developers’ negative feelings about code review, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE ’20, Association for Computing Machinery, New York, NY, USA, 2020, p. 174–185.
[8] I. Ferreira, B. Adams, J. Cheng, How heated is it? understanding github locked issues, in: Proceedings of the 19th International Conference on Mining Software Repositories, 2022.
[9] Heat detector, https://github.com/SOBotics/HeatDetector, last access: 2022-02-13.
[10] Locking conversations, https://github.blog/2014-06-09-locking-conversations/, last access: 2022-02-13.
[11] K. Cee, K. Kenski, S. A. Rains, Online and uncivil? patterns and determinants of incivility in newspaper website comments, Journal of Communication 64 (4) (2014) 658–679.
[12] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, arXiv preprint arXiv:1810.04805 (2018).
[13] S. González-Carvajal, E. C. Garrido-Merchán, Comparing bert against traditional machine learning text classification, arXiv preprint arXiv:2005.13012 (2020).
[14] J. Sarker, A. K. Turzo, A. Bosu, A benchmark study of the contemporary toxicity detectors on software engineering interactions, in: 2020’27th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2020, pp. 218–227.
[15] J. Sarker, A. K. Turzo, M. Dong, A. Bosu, Automated identification of toxic code reviews using toxirc, ACM Transactions on Software Engineering and Methodology (2023).
[16] A. Murgia, P. Tourani, B. Adams, M. Ortu, Do developers feel emotions? an exploratory analysis of emotions in software artifacts, in: Proceedings of the 11th working conference on mining software repositories, 2014, pp. 262–271.
[17] S. Minaei, N. Kalchbrenner, E. Cambria, N. Nikzad, M. Chenaghlou, J. Gao, Deep learning–based text classification: a comprehensive review, ACM Computing Surveys (CSUR) 54 (3) (2021) 1–40.
[18] K. Shah, H. Patel, D. Sanghvi, M. Shah, A comparative analysis of logistic regression, random forest and km models for the text classification, Augmented Human Research 5 (1) (2020) 1–16.
[19] T. Pranckevicius, V. Marcinkevičius, Comparison of naive bayers, random forest, decision tree, support vector machines, and logistic regression classifiers for text reviews classification, Baltic Journal of Modern Computing 5 (2) (2017) 221.
[20] W. Dai, G.-R. Xue, Q. Yang, Y. Yu, Transferring naive bayers classifiers for text classification, in: AAAI, Vol. 7, 2007, pp. 540–545.
[21] R. J. Lewis, An introduction to classification and regression tree (cart) analysis, in: Annual meeting of the society for academic emergency medicine in San Francisco, California, Vol. 14, Citesee, 2000.
[22] A. Rahman, L. Williams, Source code properties of defective infrastructure as code scripts, Information and Software Technology 112 (2019) 148–163.
[23] D. Arya, W. Wang, J. L. Guo, J. Cheng, Analysis and detection of information types of open source software issue discussions, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 454–464.
[24] M. Chouchen, J. Olongo, A. Ouni, M. W. Mkaouer, Predicting code review completion time in modern code review, arXiv preprint
structure of t4 phage lysozyme, Biochimica et Biophysica Acta (BBA)-Protein Structure 405 (2) (1975) 442–451.

[73] D. Chicco, G. Jurman, The advantages of the matthews correlation coefficient (mcc) over F1 score and accuracy in binary classification evaluation, BMC genomics 21 (1) (2020) 1–13.

[74] R. Croft, D. Newlands, Z. Chen, M. A. Babar. An empirical study of rule-based and learning-based approaches for static application security testing, in: Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2021, pp. 1–12.

[75] D. Chicco, V. Starovoitov, G. Jurman, The benefits of the matthews correlation coefficient (mcc) over the diagnostic odds ratio (dor) in binary classification assessment, IEEE Access 9 (2021) 47112–47124.

[76] H. He, Y. Ma, Imbalanced learning foundations, algorithms, and applications, IEEE Press, Wiley, 2013.

[77] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, Springer Science & Business Media, 2012.