Automated Identification of Toxic Code Reviews Using ToxiCR

JAYDEB SARKER, ASIF KAMAL TURZO, MING DONG, and AMIANGSHU BOSU, Wayne State University

Toxic conversations during software development interactions may have serious repercussions on a Free and Open Source Software (FOSS) development project. For example, victims of toxic conversations may become afraid to express themselves, therefore get demotivated, and may eventually leave the project. Automated filtering of toxic conversations may help a FOSS community maintain healthy interactions among its members. However, off-the-shelf toxicity detectors perform poorly on a software engineering dataset, such as one curated from code review comments. To counter this challenge, we present *ToxiCR*, a supervised learning based toxicity identification tool for code review interactions. ToxiCR includes a choice to select one of the 10 supervised learning algorithms, an option to select text vectorization techniques, eight preprocessing steps, and a large-scale labeled dataset of 19,651 code review comments. Two out of those eight preprocessing steps are software engineering domain specific. With our rigorous evaluation of the models with various combinations of preprocessing steps and vectorization techniques, we have identified the best combination for our dataset that boosts 95.8% accuracy and an 88.9% F1-score in identifying toxic texts. ToxiCR significantly outperforms existing toxicity detectors on our dataset. We have released our dataset, pre-trained models, evaluation results, and source code publicly, which is available at https://github.com/WSU-SEAL/ToxiCR.

CCS Concepts: • **Software and its engineering** → **Collaboration in software development**; **Integrated and visual development environments**; • **Computing methodologies** → **Supervised learning;**

Additional Key Words and Phrases: Toxicity, code review, sentiment analysis, Natural Language Processing, tool development

ACM Reference format:
Jaydeb Sarker, Asif Kamal Turzo, Ming Dong, and Amiangshu Bosu. 2023. Automated Identification of Toxic Code Reviews Using ToxiCR. *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 118 (July 2023), 32 pages.
https://doi.org/10.1145/3583562

1 INTRODUCTION

Communications among the members of many **Free and Open Source Software (FOSS)** communities include manifestations of toxic behaviors [12, 32, 46, 62, 66, 81]. These toxic communications may have decreased the productivity of those communities by wasting valuable work hours [16, 73]. FOSS developers who are frustrated over peers with "prickly" personalities [17, 34] may contemplate leaving a community for good [11, 26]. Moreover, as most FOSS communities rely on contributions from volunteers, attracting and retaining prospective joiners is crucial for

Authors' address: J. Sarker, A. K. Turzo, M. Dong, and A. Bosu, Wayne State University, Detroit, MI; emails: jaydebsarker@wayne.edu, asifkamal@wayne.edu, mdong@wayne.edu, amiangshu.bosu@wayne.edu.

1049-331X/2023/07-ART118 $15.00
https://doi.org/10.1145/3583562

the growth and survival of FOSS projects [72]. However, toxic interactions with existing members may pose barriers against successful onboarding of newcomers [48, 83]. Therefore, it is crucial for FOSS communities to proactively identify and regulate toxic communications.

Large-scale FOSS communities, such as Mozilla, OpenStack, Debian, and GNU, manage hundreds of projects and generate large volumes of text-based communications among their contributors. Therefore, it is highly time consuming and infeasible for the project administrators to identify and timely intervene with ongoing toxic communications. Although many FOSS communities have codes of conduct, they are rarely enforced due to time constraints [11]. As a result, toxic interactions can be easily found within the communication archives of many well-known FOSS projects. As an anonymous FOSS developer wrote after leaving a toxic community, "[I]t's time to do a deep dive into the mailing list archives or chat logs . . . Searching for terms that degrade women (chick, babe, girl, bitch, cunt), homophobic slurs used as negative feedback ("that's so gay"), and ableist terms (dumb, retarded, lame), may allow you to get a sense of how aware (or not aware) the community is about the impact of their language choice on minorities" [11]. Therefore, it is crucial to develop an automated tool to identify toxic communication of FOSS communities.

Toxic text classification is a **Natural Language Processing (NLP)** task to automatically classify a text as "toxic" or "non-toxic." There are several state-of-the-art tools to identify toxic contents in blogs and tweets [7, 9, 39, 54]. However, off-the-shelf toxicity detectors do not work well on **Software Engineering (SE)** communications [77], since several characteristics of such communications (e.g., code reviews and bug interactions) are different from those of blogs and tweets. For example, compared to code review comments, tweets are shorter and are limited to a maximum length. Tweets rarely include SE domain specific technical jargon, URLs, or code snippets [6, 77]. Moreover, due to different meanings of some words (e.g, "kill," "dead," and "dumb") in the SE context, SE communications with such words are often incorrectly classified as "toxic" by off-the-shelf toxicity detectors [73, 77].

To encounter this challenge, Raman et al. [73] developed a toxicity detector tool (referred to as the "STRUDEL tool" hereinafter) for the SE domain. However, the STRUDEL tool was trained and evaluated with only 611 SE texts. Recent studies have found that it performed poorly on new samples [59, 70]. To further investigate these concerns, Sarker et al. [77] conducted a benchmark study of the STRUDEL tool and four other off-the-shelf toxicity detectors using two large-scale SE datasets. To develop their datasets, they empirically developed a rubric to determine which SE texts should be placed in the "toxic" group during their manual labeling. Using that rubric, they manually labeled a dataset of 6,533 code review comments and 4,140 Gitter messages [77]. The results of their analyses suggest that none of the existing tools are reliable in identifying toxic texts from SE communications, since the performance of all five tools significantly degraded on their SE datasets. However, they also found noticeable performance boosts (i.e., accuracy improved from 83% to 92% and F-score improved from 40% to 87%) after retraining two of the existing off-the-shelf models (i.e., DPCNN [94] and BERT with FastAI [54]) using their datasets. Being motivated by these results, we hypothesize that an SE domain specific toxicity detector can boost even better performance, since off-the-shelf toxicity detectors do not use SE domain specific preprocessing steps, such as preprocessing of code snippets included within texts. On this hypothesis, this article presents ToxiCR, an SE domain specific toxicity detector. ToxiCR is trained and evaluated using a manually labeled dataset of 19,651 code review comments selected from four popular FOSS communities (i.e., Android, Chromium OS, OpenStack, and LibreOffice). We selected code review comments, since a code review usually represents a direct interaction between two persons (i.e., the author and a reviewer). Therefore, a toxic code review comment has the potential to be taken as a personal attack and may hinder future collaboration between the participants. ToxiCR is written in Python using scikit-learn [67] and TensorFlow [3]. It provides an option to train models using one of the 10 supervised Machine Learning (ML) algorithms, including five Classical and Ensemble (CLE) based, four Deep Neural Network (DNN) based, and one that is Bidirectional Encoder Representations from Transformers (BERT) based. It also includes eight preprocessing steps with two being SE domain specific and an option to choose from five different text vectorization techniques.

We empirically evaluated various optional preprocessing combinations for each of the 10 algorithms to identify the best-performing combination. During our 10-fold cross-validation evaluations, the best-performing model of ToxiCR significantly outperforms existing toxicity detectors on the code review dataset with an accuracy of 95.8% and an F1-score of 88.9% in identifying toxic texts.

The primary contributions of this work are as follows:

- ToxiCR, which is an SE domain specific toxicity detector. ToxiCR is publicly available on GitHub at https://github.com/WSU-SEAL/ToxiCR.
- An empirical evaluation of 10 ML algorithms to identify toxic SE communications.
- Implementations of eight preprocessing steps including two SE domain specific ones that can be added to model training pipelines.
- An empirical evaluation of three optional preprocessing steps in improving the performance of toxicity classification models.
- Empirical identification of the best possible combinations for all 10 algorithms.

Article Organization. The rest of the article is organized as follows. Section 2 provides a brief background and discusses prior related works. Section 3 discusses the concepts utilized in designing ToxiCR. Section 4 details the design of ToxiCR. Section 5 details the results of our empirical evaluation. Section 6 discusses the lessons learned based on this study. Section 7 discusses threats to validity of our findings. Finally, Section 8 provides a future direction based on this work and concludes this article.

2 BACKGROUND

This section defines toxic communications, provides a brief overview of prior works on toxicity in FOSS communities, and describes state-of-the-art toxicity detectors.

2.1 What Constitutes a Toxic Communication?

Toxicity is a complex phenomenon to construct, as it is more subjective than other text classification problems (e.g., online abuse, spam) [53]. Whether a communication should be considered as "toxic" also depends on a multitude of factors, such as communication medium, location, culture, and the relationship between the participants. In this research, we focus specially on written online communications. According to the Google Jigsaw AI team, a text from an online communication can be marked as toxic if it contains disrespectful or rude comments that make a participant leave the discussion forum [7]. However, the Pew Research Center marks a text as toxic if it contains threats, offensive name-calling, or sexually inappropriate content [28]. The definition of toxic communication by Anderson et al. [10] also includes insulting language or mockery. Adinolf and Turkay [5] studied toxic communication in online communities, and their views of toxic communications include harassment, bullying, griefing (i.e, constantly making other players annoyed), and trolling. To understand how persons from various demographics perceive toxicity, Kumar et al. [53] conducted a survey with 17,280 participants inside the United States. To their surprise, their results indicate that the notion toxicity cannot be attributed to any single demographic factor [53]. According to Miller et al. [59], various antisocial behaviors fit inside the toxicity umbrella, such as hate speech, trolling, flaming, and cyberbullying [59]. Some of the SE studies have investigated antisocial behaviors among SE communities using the "toxicity" construct [59, 73, 77];

Table 1. Anti-Social Constructs Investigated in Prior SE Studies

SE Study	Construct	Definition
Sarker et al. [77]	Toxicity	"[I]ncludes any of the following: (i) offensive name calling, (ii) insults, (iii) threats, (iv) personal attacks, (v) flirtations, (vi) reference to sexual activities, and (vii) swearing or cursing."
Miller et al. [59]	Toxicity	"[A]n umbrella term for various antisocial behaviors including trolling, flaming, hate speech, harassment, arrogance, entitlement, and cyberbullying."
Ferreira et al. [33]	Incivility	"[F]eatures of discussion that convey an unnecessarily disrespectful tone toward the discussion forum, its participants, or its topics."
Gunawardena et al. [40]	Destructive criticism	Negative feedback that is non-specific and is delivered in a harsh or sarcastic tone, includes threats, or attributes poor task performance to flaws of the individual.
Egelman et al. [29]	Pushback	"[T]he perception of unnecessary interpersonal conflict in code review while a reviewer is blocking a change request."

however, other studies have used various other lenses such as incivility [33], pushback [29], and destructive criticism [40]. Table 1 provides a brief overview of the studied constructs and their definitions.

2.2 Toxic Communications in FOSS Communities

Several prior studies have identified toxic communications in FOSS communities [21, 66, 73, 77, 81]. Squire and Gazda [81] found occurrences of expletives and insults in publicly available IRC and mailing list archives of top FOSS communities, such as Apache, Debian, Django, Fedora, KDE, and Joomla. More alarmingly, they identified sexist "maternal insults" being used by many developers. Recent studies have also reported toxic communications among issue discussions on GitHub [73] and during code reviews [33, 40, 66, 77].

Although toxic communications are rare in FOSS communities [77], toxic interactions can have severe consequences [21]. Carillo et al. [21] termed *toxic communications* as a "poison" that impacts the mental health of FOSS developers [21] and may contribute to stress and burnout [21, 73]. When the level of toxicity increases in a FOSS community, the community may disintegrate because developers may no longer wish to be associated with that community [21]. Moreover, toxic communications hamper onboarding of prospective joiners, as a newcomer may get turned off by the signs of a toxic culture prevalent in a FOSS community [48, 83]. Miller et al. [59] conducted a qualitative study to better understand toxicity in the context of FOSS development. They created a sample of 100 GitHub issues representing various types of toxic interactions such as insults, arrogance, trolling, entitlement, and unprofessional behavior. Their analyses also suggest that toxicity in FOSS communities differs from that observed on other online platforms such as Reddit or Wikipedia [59].

Ferreira et al. [33] investigated incivility during code review discussions based on a qualitative analysis of 1,545 emails from Linux Kernel Mailing Lists and found that the most common forms of incivility among those forums are frustration, name-calling, and importance. Egelman et al. [29] studied the negative experiences during code review, which they referred to as "pushback," which is a scenario when a reviewer is blocking a change request due to unnecessary conflict. Qiu et al. [70] further investigated such "pushback" phenomena to automatically identify interpersonal conflicts. Gunawardena et al. [40] investigated negative code review feedbacks based on a survey of 93 software developers, and they found that destructive criticism can be a threat to gender diversity in the software industry because women are less motivated to continue when they receive negative comments or destructive criticism.

2.3 State-of-the-Art Toxicity Detectors

To combat abusive online content, Google's Jigsaw AI team developed the Perspective API (PPA), which is publicly available [7]. PPA is one of the general-purpose state-of-the-art toxicity detectors. For a given text, PPA generates the probability (0 to 1) of that text being toxic. As researchers are working to identify adversarial examples to deceive PPA [44], the Jigsaw team periodically updates it to eliminate identified limitations. The Jigsaw team also published a guideline [8] to manually identify toxic contents and used that guideline to curate a crowd-sourced labeled dataset of toxic online contents [2]. This dataset has been used to train several DNN-based toxicity detectors [22, 30, 37, 39, 82, 86]. Recently, Bhat et al. proposed *ToxiScope*, a supervised learning based classifier to identify toxicity in workplace communications [14]. However, ToxiScope's best model achieved a low F1-score (i.e., 0.77) during their evaluation.

One of the major challenges in developing toxicity detectors is character-level obfuscations, where one or more characters of a toxic word are intentionally misplaced (e.g., fcuk), or repeated (e.g., shiiit), or replaced (e.g., s*ck) to avoid detection. To address this challenge, researchers have used character-level encoders instead of word-level encoders to train neural networks [54, 60, 63]. Although, character-level encoding-based models can handle such character-level obfuscations, they come with significant increments of computation times [54]. Several studies have also found racial and gender bias among contemporary toxicity detectors, as some trigger words (i.e., "gay," "black") are more likely to be associated with false positives (i.e., a non-toxic text marked as toxic) [75, 85, 89].

However, off-the-shelf toxicity detectors suffer significant performance degradation on SE datasets [77]. Such degradation is not surprising, since prior studies found off-the-shelf NLP tools also performing poorly on SE datasets [6, 50, 56, 64]. Raman et al. [73] created the STRUDEL tool, an SE domain specific toxicity detector [73], by leveraging the PPA tool and a customized version of Stanford's Politeness Detector [25]. Sarker et al. [77] investigated the performance of the STRUDEL tool and four other off-the-shelf toxicity detectors on two SE datasets [77]. In their benchmark, none of the tools achieved reliable performance to justify practical applications on SE datasets. However, they also achieved encouraging performance boosts when they retrained two of the tools (i.e., DPCNN [94] and BERT with FastAI [54]) using their SE datasets.

3 RESEARCH CONTEXT

To better understand our tool design, this section provides a brief overview of the ML algorithms integrated in ToxiCR and five word vectorization techniques for NLP tasks.

3.1 Supervised ML Algorithms

For ToxiCR, we selected 10 supervised ML algorithms from the ones that have been commonly used for text classification tasks. Our selection includes algorithms of which five are CLE based, four are DNN based, and one is BERT based. The following sections provide a brief overview of the selected algorithms.

3.1.1 Classical ML Algorithms. We have selected the following three classical algorithms, which have been used previously for classification of SE texts [6, 20, 55, 84, 84].

(1) *Decision Tree (DT):* In this algorithm, the dataset is continuously split according to a certain parameter. DT has two entities, namely decision nodes and leaves. The leaves are the decisions or the final outcomes. The decision nodes are where the data is split into two or more subnodes [71].
(2) *Logistic Regression (LR)*: LR creates a mathematical model to predict the probability for one of the two possible outcomes and is commonly used for binary classification tasks [13].

(3) *Support Vector Machine (SVM)*: After mapping the input vectors into a high-dimensional non-linear feature space, SVM tries to identify the best hyperplane to partition the data into *n*-classes, where *n* is the number of possible outcomes [24].

3.1.2 Ensemble Methods. Ensemble methods create multiple models and then combine them to produce improved results. We have selected the following two ensemble method based algorithms, based on prior SE studies [6, 55, 84]:

(1) *Random Forest (RF)*: RF is an ensemble-based method that combines the results produced by multiple decision trees [42]. RF creates independent decision trees and combines them in parallel using the "bagging" approach [18].
(2) *Gradient-Boosted Decision Trees (GBT)*: Similar to RF, GBT is also an ensemblebased method using decision trees [36]. However, GBT creates decision trees sequentially so that each new tree can correct the errors of the previous one and combines the results using the "boosting" approach [80].

3.1.3 Deep Neural Networks. In recent years, DNN-based models have shown significant performance gains over CLE-based models in text classification tasks [52, 94]. In this research, we have selected four state-of-the-art DNN-based algorithms:

(1) *Long Short-Term Memory (LSTM)*: A **Recurrent Neural Network (RNN)** processes inputs sequentially, remembers the past, and makes decisions based on what it has learned from the past [74]. However, traditional RNNs may perform poorly on a long-sequence of inputs, such as those seen in text classification tasks due to the "vanishing gradient problem." This problem occurs when a RNN's weights are not updated effectively due to exponentially decreasing gradients. To overcome this limitation, Hochreiter and Schmidhuber [43] proposed LSTM, a new type of RNN architecture that overcomes the challenges posed by long-term dependencies using a gradient-based learning algorithm. LSTM consists four units: (i) the *input gate*, which decides what information to add from current step; (ii) the *forget gate*, which decides what is to be kept from prior steps; (iii) the *output gate*, which determines the next hidden state; and (iv) the *memory cell*, which stores information from previous steps.
(2) *Bidirectional LSTM (BiLSTM)*: A BiLSTM is composed of a forward LSTM and a backward LSTM to model the input sequences more accurately than an unidirectional LSTM [23, 38]. In this architecture, the forward LSTM takes input sequences in the forward direction to model information from the past, whereas the backward LSTM takes input sequences in the reverse direction to model information from the future [45]. BiLSTM has been shown to be perform better than unidirectional LSTM in several text classification tasks, as it can identify language contexts better than LSTM [38].
(3) *Gated Recurrent Unit (GRU)*: Similar to LSTM, GRU belongs to the RNN family of algorithms. However, GRU aims to handle the "vanishing gradient problem" using a different approach than LSTM. GRU has a much simpler architecture with only two units: the update gate and reset gate. The reset gate decides what information should be forgotten for the next pass, and the update gate determines which information should pass to the next step. Unlike LSTM, GRU does not require any memory cell and therefore needs shorter training time than LSTM [31].
(4) *Deep Pyramid CNN (DPCNN)*: **Convolutional Neural Networks (CNNs)** are a specialized type of neural network that utilizes a mathematical operation called *convolution* in at least one of their layers. CNNs are most commonly used for image classification tasks. Johnson and Zhang [49] proposed a special type of CNN architecture (DPCNN) for text classification tasks. Although DPCNN achieves faster training time by utilizing word-level CNNs to

represent input texts, it does not sacrifice accuracy over character-level CNNs due to its carefully designed deep but low-complexity network architecture.

3.1.4 Transformer Model. In recent years, Transformer-based models have been used for sequence-to-sequence modeling such as neural machine translations. For sequence-to-sequence modeling, a Transformer architecture includes two primary parts: (i) the *encoder*, which takes the input and generates the higher-dimensional vector representation, and (ii) the *decoder*, which generates the the output sequence from the abstract vector from the encoder. For classification tasks, the output of encoders are used for training. Transformers solve the "vanishing gradient problem" on long text inputs using the "self-attention mechanism," which is a technique to identify the important features from different positions of an input sequence [87].

In this study, we select BERT [27]. BERT-based models have achieved remarkable performance in various NLP tasks, such as question answering, sentiment classification, and text summarization [4, 27]. BERT's transformer layers use multi-headed attention instead of recurrent units (e.g., LSTM, GRU) to model the contextualized representation of each word in an input.

3.2 Word Vectorization

To train an NLP model, input texts need to be converted into a vector of features that ML models can work on. **Bag-of-Words (BOW)** is one of the most basic representation techniques, which turns an arbitrary text into a fixed-length vector by counting how many times each word appears. As BOW representations do not account for grammar and word order, ML models trained using BOW representations fail to identify relationships between words. *Word embedding* techniques convert words to n-dimensional vector forms in such a way that words having similar meanings have vectors close to each other in the n-dimensional space. Word embedding techniques can be further divided into two categories: (i) *context-free embedding*, which creates the same representation of a word regardless of the context where it occurs, and (ii) *contextualized word embedding*, which aims at capturing word semantics in different contexts to address the issue of polysemy (i.e., words with multiple meanings) and the context-dependent nature of words. For this research, we have experimented with five word vectorization techniques: one BOW based, three context free, and one contextualized. The following sections provide a brief overview of those techniques.

3.2.1 Tf-Idf. **Term Frequency–Inverse Document Frequency (Tf-Idf)** is a BOW-based vectorization technique that evaluates how relevant a word is to a document in a collection of documents. The Tf-Idf score for a word is computed by multiplying two metrics: how many times a word appears in a document (Tf) and the inverse document frequency of the word across a set of documents (Idf). The following equations show the computation steps for Tf-Idf scores:

$$Tf(w,d) = f(w,d) / \sum_{t \epsilon d} f(t,d), \tag{1}$$

where $f(t,d)$ is the frequency of the word (w) in the document (d), and $\sum_{t \epsilon d} f(t,d)$ represents the total number of words in d. Inverse document frequency (Idf) measures the importance of a term across all documents.

$$Idf(w) = log_e(N/w_N) \tag{2}$$

Here, N is the total number of documents and w_N represents the number of documents having w. Finally, we computed the Tf-Idf score of a word as

$$TfIdf(w,d) = Tf(w,d) * Idf(w). \tag{3}$$

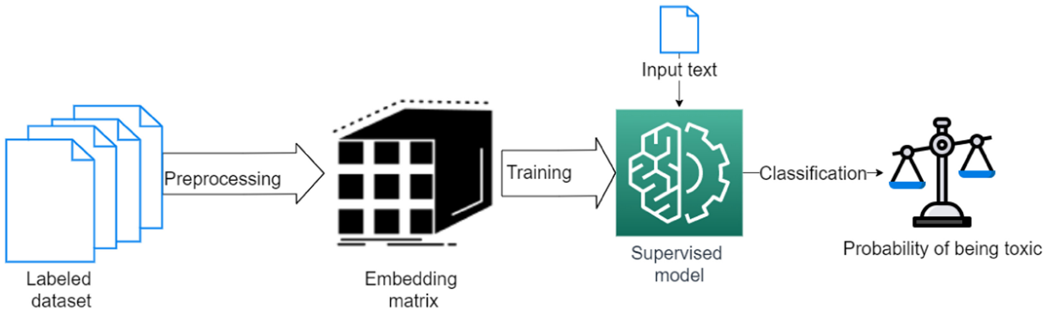


Fig. 1. A simplified overview of ToxiCR showing the key pipeline.

3.2.2 word2vec. In 2013, Mikolov et al. [58] proposed word2vec, a context-free word embedding technique. It is based on two neural network models named *CBOW* (Continuous Bag-of-Words) and *Skip-gram*. CBOW predicts a target word based on its context, whereas Skip-gram uses the current word to predict its surrounding context. During the training, word2vec takes a large corpus of text as input and generates a vector space, where each word in the corpus is assigned a unique vector and words with similar meaning are located close to one another.

3.2.3 GloVe. Proposed by Pennington et al. [68], **Global Vectors for Word Representation (GloVe)** is an unsupervised algorithm to create context-free word embedding. Unlike word2vec, GloVe generates vector space from the global co-occurrence of words.

3.2.4 fastText. Developed by the Facebook AI team, *fastText* is a simple and efficient method to generate context-free word embeddings [15]. Whereas word2vec and GloVe cannot provide embedding for out of vocabulary words, fastText overcomes this limitation by taking into account morphological characteristics of individual words. A word's vector in fastText-based embedding is built from vectors of substrings of characters contained in it. Therefore, fastText performs better than word2vec or GloVe in NLP tasks if a corpus contains unknown or rare words [15].

3.2.5 BERT. Unlike context-free embeddings (e.g., word2vec, GloVe, and fastText), where each word has a fixed representation regardless of the context within which the word appears, a contextualized embedding produces word representations that are dynamically informed by the words around them. In this study, we use BERT [27]. Similar to fastText, BERT can also handle out-of-vocabulary words.

4 TOOL DESIGN

Figure 1 shows the architecture of ToxiCR. It takes a text (i.e., code review comment) as input and applies a series of mandatory preprocessing steps. Then, it applies a series of optional preprocessing based on selected configurations. Preprocessed texts are then fed into one of the selected vectorizers to extract features. Finally, output vectors are used to train and validate our supervised learning based models. The following sections detail the research steps to design ToxiCR.

4.1 Conceptualization of Toxicity

As mentioned in Section 2.1, what constitutes a "toxic communication" depends on various contextual factors. In this study, we specifically focus on popular FOSS projects such as Android, Chromium OS, LibreOffice, and OpenStack, where participants represent diverse culture, education, ethnicity, age, religion, gender, and political views. As participants are expected and even recommended to maintain a high level of professionalism during their interactions with other members of those communities [35, 65, 69], we adopt the following expansive definition of toxic contents for this context[1]:

> *"An SE conversation will be considered as toxic, if it includes any of the following: (i) offensive name calling, (ii) insults, (iii) threats, (iv) personal attacks, (v) flirtations, (vi) reference to sexual activities, and (vii) swearing or cursing."*

Our conceptualization of toxicity closely aligns with another recent work by Bhat et al. [14] that focuses on professional workplace communication. According to their definition, toxic behavior includes any of the following: sarcasm, stereotyping, rude statements, mocking conversations, profanity, bullying, harassment, discrimination, and violence.

4.2 Training Dataset Creation

As of May 2021, there are three publicly available labeled datasets of toxic communications from the SE domain. The dataset of Raman et al. [73] created for the STRUDEL tool includes only 611 texts. In our recent benchmark study (referred to as the "benchmark study" hereinafter), we created two datasets: (i) a dataset of 6,533 code review comments selected from three popular FOSS projects (referred to as "code review dataset 1" hereinafter; i.e., Android, Chromium OS, and LibreOffice) and (ii) a dataset of 4,140 Gitter messages selected from the Gitter Ethereum channel (referred to as the "Gitter dataset" hereinafter) [77]. We followed the exact same process used in the benchmark study to select and label an additional 13,038 code review comments selected from OpenStack projects. In the following, we briefly describe our four-step process, which is detailed in our prior publication [77].

4.2.1 Data Mining. In the benchmark, we wrote a Python script to mine the Gerrit [61] managed code review repositories of three popular FOSS projects: Android, Chromium OS, and LibreOffice. Our script leverages Gerrit's REST API to mine and store all publicly available code reviews in a MySQL dataset. We use the same script to mine ≈ 2.1 million code review comments belonging to 670,996 code reviews from the OpenStack projects' code review repository hosted at https://review.opendev.org/. We followed an approach similar to that of Paul et al. [66] to identify potential bot accounts based on keywords (e.g., "bot," "auto," "build," "auto," "travis," "CI," "jenkins," and "clang"). If our manual manual validations of comments authored by a potential bot account confirmed it to be a bot, we excluded all comments posted by that account.

4.2.2 Stratified Sampling of Code Review Comments. Since toxic communications are rare [77] during code reviews, a randomly selected dataset of code review comments will be highly imbalanced with less than 1% toxic instances. To overcome this challenge, we adopted a stratified sampling strategy as suggested by Särndal et al. [79]. We used Google's PPA [7] to compute the toxicity score for each review comment. If the PPA score is more than 0.5, then the review comment is more likely to be toxic. Among the 2.1 million code review comments, we found 4,118 comments with PPA scores greater than 0.5. In addition to those 4,118 review comments, we selected 9,000 code review comments with PPA scores less than 0.5. We selected code review comments with PPA scores less than 0.5 in a well-distributed manner. We split the texts into five categories (i.e., score: 0–0.1, 0.11–0.2, and so on) and took the same amount (1,800 texts) from each category. For example, we took 1,800 samples that have a score between 0.3 and 0.4.

4.2.3 Manual Labeling. During the benchmark study [77], we developed a manual labeling rubric fitting our definition as well as study context. Our initial rubric was based on the guidelines

[1]We introduced this definition in our prior study [77]. We are repeating this definition to assist in better comprehension of this article's context.

Table 2. Rubric to Label the SE Text as Toxic or Non-Toxic, Adjusted from Previous Work [77]

#	Rule	Rationale	Example*
Rule 1:	If a text includes profane or curse words, it would be marked as "toxic."	Profanities are the most common sources of online toxicities.	"fuck! Consider it done!"
Rule 2:	If a text includes an acronym, which generally refers to an expletive or swearing, it would be marked as "toxic."	Sometimes people use acronyms of profanities, which are equally toxic as their expanded form.	"WTF are you doing!"
Rule 3:	Insulting remarks regarding another person or entities would be marked as "toxic."	Insulting another developer may create a toxic environment and should not be encouraged.	"...shut up, smartypants."
Rule 4:	Attacking a person's identity (e.g., race, religion, nationality, gender, or sexual orientation) would be marked as "toxic."	Identity attacks are considered toxic among all categories of online conversations.	"Stupid fucking superstitious Christians."
Rule 5:	Aggressive behavior or threatening another person or a community would be marked as "toxic."	Aggregations or threats may stir hostility between two developers and force the recipients to leave the community.	"yeah, but I'd really give a lot for an opportunity to punch them in the face."
Rule 6:	Both implicit or explicit references to sexual activities would be marked as "toxic."	Implicit or explicit references to sexual activities may make some developers, particularly females, uncomfortable and make them leave a conversation.	"This code makes me so horny. It's beautiful."
Rule 7:	Flirtations would be marked as "toxic."	Flirtations may also make a developer uncomfortable and make a recipient avoid the other person during future collaborations.	"I really miss you my girl."
Rule 8:	If a demeaning word (e.g., "dumb," "stupid," "idiot," "ignorant") refers to either the writer him/herself or his/her work, the sentence would not be marked as "toxic" if it does not fit any of the first seven rules.	It is common in the SE community to use those words for expressing their own mistakes. In those cases, the use of those toxic words with regard to themselves or their work does not make toxic meaning. Although such texts are unprofessional [59], they do not degrade future communication or collaboration.	"I'm a fool and didn't get the point of the deincrement. It makes sense now."
Rule 9:	A sentence that does not fit rules 1 through 8 would be marked as "non-toxic."	General non-toxic comments.	"I think ResourceWithProps should be there instead of GenericResource."

*Examples are provided verbatim from the datasets to accurately represent the context. We did not censor any text, except omitting the reference to a person's name.

published by the **Conversation AI Team (CAT)** [8]. With these guidelines as our starting point, two of the authors independently went through 1,000 texts to adopt the rules to better fit our context. Then, we had a discussion session to merge and create a unified set of rules. Table 2 represents our final rubric that has been used for manual labeling during both the benchmark study and this study.

Although we have used the guideline from the CAT as our starting point, our final rubric differs from the CAT rubric in two key aspects to better fit our target SE context. First, our rubric is targeted toward professional communities, whereas the CAT rubric is targeted toward general online communications. Therefore, profanities and swearing to express a positive attitude may

Table 3. Overview of the Three SE Domain Specific Toxicity Datasets Used in This Study

Dataset	# Total Texts	# Toxic	# Non-Toxic
Code review 1	6,533	1,310	5,223
Code review 2	13,118	2,447	10,591
Gitter dataset	4,140	1,468	2,672
Code review (combined)	19,651	3,757	15,819

not be considered as toxic by the CAT rubric. For example, "That's fucking amazing! thanks for sharing." is given as an example of "Not Toxic, or Hard to Say" by the CAT rubric. On the contrary, any sentence with profanity or swearing is considered "toxic" according to our rubric, since such a sentence does not constitute a healthy interaction. Our characterization of profanities also aligns with the recent SE studies on toxicity [59] and incivility [33]. Second, the CAT rubric is for labeling on a 4-point scale (i.e., "Very Toxic," "Toxic," "Slightly Toxic or Hard to Say," and "Non-Toxic") [1]. On the contrary, our labeling rubric is much simpler on a binary scale ("Toxic" and "Non-Toxic"), since development of a 4-point rubric as well as classifier is significantly more challenging. We consider the development of a 4-point rubric as a potential future direction.

Using this rubric, two of the authors independently labeled the 13,118 texts as either "toxic" or "non-toxic." After the independent manual labeling, we compared the labels from the two raters to identify conflicts. The two raters had agreements on 12,608 (96.1%) texts during this process and achieved a Cohen's kappa (κ) score of 0.92 (i.e., an almost perfect agreement).[2] We had meetings to discuss the conflicting labels and assign agreed-upon labels for those cases. At the end of conflict resolution, we found 2,447 (18.76%) texts labeled as "toxic" among the 13,118 texts. We refer to this dataset as "code review dataset 2" hereinafter. Table 3 provides an overview of the three datasets used in this study.

4.2.4 Dataset Aggregation. Since the reliability of a supervised learning based model increases with the size of its training dataset, we decided to merge the two code review datasets into a single dataset (referred to as the "combined code review dataset" hereinafter). We believe that such merging is not problematic for the following reasons:

(1) Both of the datasets are labeled using the same rubrics and following the same protocol.
(2) We used the same set of raters for manual labeling.
(3) Both of the datasets are picked from the same type of repository (i.e., Gerrit-based code reviews).

The merged code review dataset includes a total of 19,651 code review comments, where 3,757 comments (19.1%) are labeled as "toxic."

4.3 Data Preprocessing

Code review comments are different from news, articles, books, or even spoken language. For example, review comments often contain word contractions, URLs, and code snippets. Therefore, we implemented eight data preprocessing steps. Five of those steps are mandatory, as those aim to remove unnecessary or redundant features. The remaining three steps are optional, and their impacts on toxic code review detection are empirically evaluated in our experiments. Two out of

[2]Kappa (κ) values are commonly interpreted as follows: values ≤ 0 indicating "no agreement," 0.01 to 0.20 as "none to slight," 0.21 to 0.40 as "fair," 0.41 to 0.60 as "moderate," 0.61 to 0.80 as "substantial," and 0.81 to 1.00 as "almost perfect agreement."

Table 4. Examples of Text Preprocessing Steps Implemented in ToxiCR

Step	**Original**	**Post Preprocessing**
URL-rem	ah crap. Not sure how I missed that. http://goo.gl/5NFKcD	ah crap. Not sure how I missed that.
Cntr-exp	this line shouldn't end with a period	this line *should not* end with a period
Sym-rem	Missing: Partial-Bug: #1541928	Missing Partial Bug 1541928
Rep-elm	haha... *looooooooser!*	haha.. loser!
Adv-ptrn	oh right, *sh**t*	oh right, shit
Kwrd-rem†	These *static* values should be put at the top	These values should be put at the top
Id-split†	idp = self._create_dummy_idp (add_clean_up = False)	idp = self. create dummy idp(add clean up= False)

†, an optional SE domain specific preprocessing step.

the three optional preprocessing steps are SE domain specific. Table 4 shows examples of texts before and after preprocessing.

4.3.1 *Mandatory Preprocessing.* ToxiCR implements the following five mandatory preprocessing steps:

- *URL removal (URL-rem)*: A code review comment may include a URL (e.g., reference to documentation or a StackOverflow post). Although URLs are irrelevant for a toxicity classifier, they can increase the number of features for supervised classifiers. We used a regular expression matcher to identify and remove all URLs from our datasets.
- *Contraction expansion (Cntr-exp)*: Contractions, which are the shortened form of one or two words, are common among code review texts. For example, some common ones are doesn't →does not and we're →we are. By creating two different lexicons of the same term, contractions increase the number of unique lexicons and add redundant features. We replaced the commonly used 153 contractions, each with its expanded version.
- *Symbol removal (Sym-rem)*: Since special symbols (e.g., &, #, and ˆ) are irrelevant for toxicity classification tasks, we use a regular expression matcher to identify and remove special symbols.
- *Repetition elimination (Rep-elm)*: A person may repeat some of the characters to misspell a toxic word to evade detection from dictionary-based toxicity detectors. For example, in the sentence "You're duumbbbb!," "dumb" is misspelled through character repetitions. We have created a pattern-based matcher to identify such misspelled cases and replace each with its correctly spelled form.
- *Adversarial pattern identification (Adv-ptrn)*: A person may misspell profane words by replacing some characters with a symbol (e.g., "f*ck" and "b!tch") or use an acronym for a slang (e.g., "stfu"). To identify such cases, we have developed a profanity preprocessor, which includes pattern matchers to identify various forms of the 85 commonly used profane words. Our preprocessor replaces each identified case with its correctly spelled form.

4.3.2 *Optional Preprocessing.* ToxiCR includes options to apply following three optional preprocessing steps:

- *Identifier splitting (Id-split)*: In this preprocessing, we use a regular expression matcher to split identifiers written in both camelCase and under_score forms. For example, this step will replace "isCrap" with "is Crap" and replace "is_shitty" with "is shitty." This preprocessing may help identify example code segments with profane words.

- *Programming keywords removal (Kwrd-rem)*: Code review texts often include programming language specific keywords (e.g., "while," "case," "if," "catch," and "except"). These keywords are SE domain specific jargon and are not useful for toxicity prediction. We have created a list of 90 programming keywords used in the popular programming languages (e.g., C++, Java, Python, C#, PHP, JavaScript, and Go). This step searches and removes occurrences of those programming keywords from a text.
- *Count profane words (profane-count)*: Since the occurrence of profane words is suggestive of a toxic text, we believe that the number of profane words in a text may be an excellent feature for a toxicity classifier. We have created a list of 85 profane words, and this step counts the occurrences of these words in a text. The remaining seven preprocessing steps modify an input text pre-vectorization; however, this step adds an additional dimension to the post-vectored output of a text.

4.4 Word Vectorizers

ToxiCR includes the option to use five different word vectorizers. However, due to the limitations of the algorithms, each of the vectorizers can work with only one group of algorithms. In our implementation, Tf-Idf works only with the CLE methods; word2vec, GloVe, and fastText work with the DNN-based algorithms; and the BERT model includes its pre-trained vectorizer. For vectorizers, we chose the following implementations.

(1) *Tf-Idf*: We select the `TfidfVectorizer` from the scikit-learn library. To prevent overfitting, we discard words not belonging to at least 20 documents in the corpus.
(2) *word2vec*: We select the pre-trained word2vec model available at https://code.google.com/archive/p/word2vec/. This model was trained with a Google News dataset of 100 billion words and contains 300-dimensional vectors for 3 million words and phrases.
(3) *GloVe*: Among the publicly available, pre-trained GloVe models (https://github.com/stanfordnlp/GloVe), we select the common crawl model. This model was trained using web crawl data of 820 billion tokens and contains 300-dimensional vectors for 2.2 million words and phrases.
(4) *fastText*: From the pre-trained fastText models (https://fasttext.cc/docs/en/english-vectors.html), we select the common crawl model. This model was trained using the same dataset as our selected our GloVe model and contains 300-dimensional vectors for 2 million words.
(5) *BERT*: We select a variant of the BERT model published as "BERT_en_uncased." This model was pre-trained on a dataset of 2.5 billion words from Wikipedia and 800 million words from BookCorpus [95].

4.5 Architecture of the ML Models

This section discusses the architecture of the ML models implemented in ToxiCR.

4.5.1 CLE Methods.

We used the scikit-learn [67] implementations of the CLE classifiers:

- *Decision Tree (DT)*: We used the `DecisionTreeClassifier` class with default parameters.
- *Logistic Regression (LR)*: We used the `LogisticRegression` class with default parameters.
- *Support Vector Machine (SVM)*: Among the various SVM implementations offered by scikit-learn, we selected the `LinearSVC` class with default parameters.
- *Random Forest (RF)*: We used the `RandomForestClassifier` class from scikit-learn ensembles. To prevent overfitting, we set the minimum number of samples to split at to 5. For the other parameters, we accepted the default values.

- *Gradient-Boosted Decision Trees (GBT)*: We used the `GradientBoostingClassifier` class from the scikit-learn library. We set n_iter_no_change =5, which stops the training early if the last five iterations did not achieve any improvement in accuracy. We accepted the default values for the other parameters.

4.5.2 DNN Model. We used version 2.5.0 of the TensorFlow library [3] for training the four DNN models (i.e., LSTM, BiLSTM, GRU, and DPCNN).

Common parameters of the four models are the following:

- We set `max_features = 5000` (i.e., maximum number of features to use) to reduce the memory overhead as well as to prevent model overfitting.
- Maximum length of input is set to 500, which means our models can take texts with at most 500 words as inputs. Any input over this length would be truncated to 500 words.
- As all three pre-trained word embedding models use 300-dimensional vectors to represent words and phrases, we set the embedding size to 300.
- The embedding layer takes input embedding matrix as inputs. Each of the words (w_i) from a text is mapped (embedded) to a vector (v_i) using one of the three context-free vectorizers (i.e., fastText, GloVe, and word2vec). For a text T, its embedding matrix will have a dimension of ($300Xn$), where n is the total number of words in that text.
- Since we are developing binary classifiers, we selected `binary_crossentropy` loss function for model training.
- We selected the `Adam optimizer` (Adaptive Moment Estimation) [51] to update the weights of the network during the training time. The initial `learning_rate` is set to 0.001.
- During the training, we set *accuracy* (A) as the evaluation metric.

The four deep neural models of ToxiCR are primarily based on three layers, as described briefly in the following. Architecture diagrams of the models are included in our replication package [76].

- *Input embedding layer*: After preprocessing of code review texts, they are converted to an input matrix. The embedded layer maps the input matrix to a fixed-dimension input embedding matrix. We used three pre-trained embeddings, which help the model capture the low-level semantics using position-based texts.
- *Hidden state layer*: This layer takes the position-wise embedding matrix and helps capture the high-level semantics of words in code review texts. The configuration of this layer depends on the choice of the algorithm. ToxiCR includes one CNN (i.e., DPCNN) and three RNN (i.e., LSTM, BiLSTM, GRU) based hidden layers. In the following, we describe the key properties of these four types of layers:
  - *DPCNN blocks*: Following the implementation of DPCNN [49], we set seven convolution blocks with the `Conv1D` layer after the input embedding layer. We also set the other parameters of the DPCNN model following Johnson and Zhang [49]. Outputs from each of the CNN blocks are passed to a `GlobalMaxPooling1D` layer to capture the most important features from the inputs. A dense layer is set with 256 units, which is activated with a linear activation function.
  - *LSTM blocks*: From the Keras library, we use the LSTM unit to capture the hidden sequence from the input embedding vector. The LSTM unit generates the high-dimensional semantic representation vector. To reshape the output dimension, we use the flatten and dense layers after the LSTM unit.
  - *BiLSTM blocks*: For text classification tasks, BiLSTM works better than LSTM for capturing the semantics of a long sequence of text. Our model uses 50 units of BiLSTM units from the Keras library to generate the hidden sequence of input embedding matrix. To

downsample the high-dimension hidden vector from BiLSTM units, we set a `GlobalMaxPool1D` layer. This layer downsamples the hidden vector from the BiLSTM layer by taking the maximum value of each dimension and thus captures the most important features for each vector.
  - *GRU blocks*: We use bidirectional GRUs with 80 units to generate the hidden sequence of the input embedding vector. To keep the most important features from the GRU units, we set a concatenation of `GlobalAveragePooling1D` and `GlobalMaxPooling1D` layers. `GlobalAveragePooling1D` calculates the average of the entire sequence of each vector, and `GlobalMaxPooling1D` finds the maximum value of the entire sequence.
- *Classifier layer*: The output vector of the hidden state layer projects to the output layer with a dense layer and a `sigmoid` activation function. This layer generates the probability of the input vector from the range 0 to 1. We chose a `sigmoid` activation function because it provides the probability of a vector within a 0 to 1 range.

4.5.3 Transformer Models. Among the several pre-trained BERT models,[3] we used `bert_en_uncased`, which is also known as the *BERT_base* model. We downloaded the models from `tensorflow_hub`, which consists of trained ML models ready for fine-tuning.

Our BERT model architecture is as follows:

- *Input layer*: This layer takes the preprocessed input text from our SE dataset. To fit into the BERT pre-trained encoder, we preprocess each text using a matching preprocessing model (i.e., `bert_en_uncased_preprocess`[4]).
- *BERT encoder*: From each preprocessed text, this layer produces BERT embedding vectors with higher-level semantic representations.
- *Dropout layer*: To prevent overfitting as well as eliminate unnecessary features, outputs from the BERT encoder layer are passed to a dropout layer with a probability of 0.1 to drop an input.
- *Classifier layer*: Outputs from the dropout layer are passed to a two-unit dense layer, which transforms the outputs into two-dimensional vectors. From these vectors, a one-unit dense layer with a linear activation function generates the probabilities of each text being toxic. Unlike the DNN's output layer, we found that linear activation function provides better accuracy than non-linear ones (e.g., *relu*, *sigmoid*) for BERT-based models.
- *Parameters*: Similar to the DNN models, we use `binary_crossentropy` as the loss function and *Binary Accuracy* as the evaluation metric during training.
- *Optimizer*: We set the optimizer as `Adamw` [57], which improved the generalization performance of the Adam optimizer. `Adamw` minimizes the prediction loss and does regularization by decaying weight. Following the recommendation of Devlin et al. [27], we set the initial learning rate to $3e-5$.

4.6 Model Training and Validation

The following sections detail our model training and validation approaches.

4.6.1 Classical and Ensemble. We evaluated all the models using 10-fold cross validations, where the dataset was randomly split into 10 groups and each of the 10 groups was used as test dataset once, whereas the remaining 9 groups were used to train the model. We used a stratified split to ensure similar ratios of the classes between the test and training sets.

[3]https://github.com/google-research/bert.
[4]https://tfhub.dev/tensorflow/bert_en_uncased_preprocess/3.

Table 5. Overview of the Hyper-Parameters for Our DNNs and Transformer

Hyper-Parameter	**Deep Neural Network (i.e., DPCNN, LSTM, BiLSTM, and GRU)**	**Transformer (BERT)**
Activation	sigmoid	linear
Loss function	binary crossentropy	binary crossentropy
Optimizer	adam	Adamw
Learning rate	0.001	3e-5
Early stopping monitor	val_loss	val_loss
Epochs	40	15
Batch size	128	256

4.6.2 DNN and Transformers. We customized several hyper-parameters of the DNN models to train our models. Table 5 provides an overview of those customized hyper-parameters. A DNN model can be overfitted due to overtraining. To encounter that, we configured our training parameters to find the best fit model that is not overfitted. During training, we split our dataset into three sets according to an 8:1:1 ratio. These three sets are used for training, validation, and testing, respectively, during our 10-fold cross validations to evaluate our DNN and transformer models. For training, we set maximum 40 epochs[5] for the DNN models and a maximum of 15 epochs for the BERT model. During each epoch, a model is trained using 80% samples, is validated using 10% samples, and the remaining 10% is used to measure the performance of the trained model. To prevent overfitting, we used an *EarlyStopping* function from the Keras library, which monitors *minimum val loss*. If the performance of a model on the validation dataset starts to degrade (e.g., loss begins to increase or accuracy begins to drop), then the training process is stopped.

4.7 Tool Interface

We designed ToxiCR to support stand-alone evaluation and to be used as a library for toxic text identification. We also included pre-trained models to save model training time. Listing 1 shows a sample code to predict the toxicity of texts using our pre-trained BERT model.

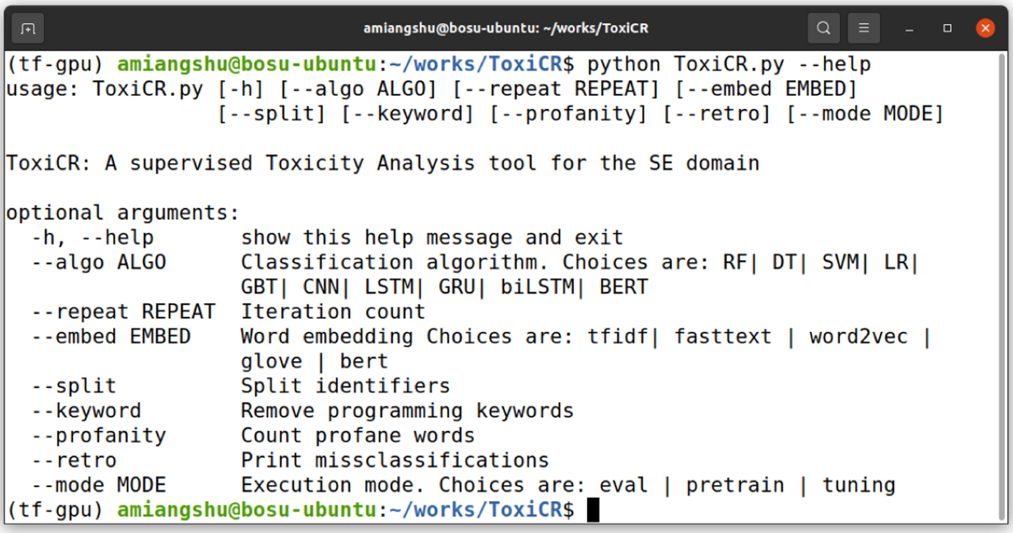
We also included a command line based interface for model evaluation, retraining, and fine-tuning hyper-parameters. Figure 2 shows the usage help message of ToxiCR. Users can customize execution with eight optional parameters, which are as follows:

```
from ToxiCR import ToxiCR

clf=ToxiCR(ALGO="BERT", count_profanity=False, remove_keywords=True,
            split_identifier=False,
            embedding="bert", load_pretrained=True)

clf.init_predictor()
sentences=["this is crap", "thank you for the information",
            "shi*tty code" ]

results=clf.get_toxicity_class(sentences)
```

Listing 1. Example usage of ToxiCR to classify toxic texts.

- *Algorithm selection*: Users can select 1 of the 10 included algorithms by using the `--algo ALGO` option.

[5]The number of times a learning algorithm will work through the entire training dataset.



```
(tf-gpu) amiangshu@bosu-ubuntu:~/works/ToxiCR$ python ToxiCR.py --help
usage: ToxiCR.py [-h] [--algo ALGO] [--repeat REPEAT] [--embed EMBED]
                 [--split] [--keyword] [--profanity] [--retro] [--mode MODE]

ToxiCR: A supervised Toxicity Analysis tool for the SE domain

optional arguments:
  -h, --help       show this help message and exit
  --algo ALGO      Classification algorithm. Choices are: RF| DT| SVM| LR|
                   GBT| CNN| LSTM| GRU| biLSTM| BERT
  --repeat REPEAT  Iteration count
  --embed EMBED    Word embedding Choices are: tfidf| fasttext | word2vec |
                   glove | bert
  --split          Split identifiers
  --keyword        Remove programming keywords
  --profanity      Count profane words
  --retro          Print missclassifications
  --mode MODE      Execution mode. Choices are: eval | pretrain | tuning
(tf-gpu) amiangshu@bosu-ubuntu:~/works/ToxiCR$ 
```

Fig. 2. The command line interface of ToxiCR showing various customization options.

- *Number of repetitions*: Users can specify the number of times to repeat the 10-fold cross validations in evaluation mode using the `--repeat n` option. The default value is 5.
- *Embedding*: ToxiCR includes five different vectorization techniques: `tfidf`, `word2vec`, `glove`, `fasttext`, and `bert`. `tfidf` is configured to be used only with the CLE models. `word2vec`, `glove`, and `fastext` can be used only with the DNN models. Finally, `bert` can be used only with the Transformer model. Users can customize this selection using the `--embed EMBED` option.
- *Identifier splitting*: Using the `--split` option, users can select to apply the optional preprocessing step to split identifiers written in camelCases or under_scores.
- *Programming keywords*: Using the `--keyword` option, users can select to apply the optional preprocessing step to remove programming keywords.
- *Profanity*: The `--profanity` optional preprocessing step allows to add the number of profane words in a text as an additional feature.
- *Missclassification diagnosis*: The `--retro` option is useful for error diagnosis. If this option is selected, ToxiCR will write all misclassified texts in a spreadsheet to enable manual analyses.
- *Execution mode*: ToxiCR can be executed in three different modes. The `eval` mode will run 10-fold cross validations to evaluate the performance of an algorithm with the selected options. In the `eval` mode, ToxiCR writes the results of each run and model training time in a spreadsheet. The `retrain` mode will train a classifier with the full dataset. This option is useful for saving models in a file to be used in the future. Finally, the `tuning` mode allows to explore various algorithm hyper-parameters to identify the optimum set.

5 EVALUATION

We empirically evaluated the 10 algorithms included in ToxiCR to identify the best possible configuration to identify toxic texts from our datasets. The following sections detail our experimental configurations and the results of our evaluations.

Table 6. Performance of the Four Contemporary Toxic Detectors to Establish a Baseline Performance

Models	**Non-Toxic**			**Toxic**			**Accuracy**
	P_0	R_0	$F1_0$	P_1	R_1	$F1_1$	
Perspective API [7] (off-the-shelf)	0.92	0.79	0.85	0.45	0.70	0.55	0.78
Strudel tool (off-the-shelf) [73]	0.93	0.76	0.83	0.43	0.77	0.55	0.76
Strudel (retrain) [78]	**0.97**	**0.96**	**0.97**	**0.85**	**0.86**	**0.85**	**0.94**
DPCNN (retrain) [77]	0.94	0.95	0.94	0.81	0.76	0.78	0.91

For our classifications, we consider toxic texts as "class 1" and non-toxic texts as "class 0."

5.1 Experimental Configuration

To evaluate the performance of our models, we use precision, recall, F-score, and accuracy for both toxic (class 1) and non-toxic (class 0) classes. We computed the following evaluation metrics:

- *Precision (P)*: For a class, precision is the percentage of identified cases that truly belongs to that class.
- *Recall (R)*: For a class, recall is the ratio of correctly predicted cases and total number of cases.
- *F1-score (F1)*: F1-score is the harmonic mean of precision and recall.
- *Accuracy (A)*: Accuracy is the percentage of cases that a model predicted correctly.

In our evaluations, we consider the F1-score for the toxic class (i.e., $F1_1$) as the most important metric to evaluate these models, since (i) identification of toxic texts is our primary objective, and (ii) our datasets are imbalanced with more than 80% non-toxic texts.

To estimate the performance of the models more accurately, we repeated 10-fold cross validations five times and computed the means of all metrics over those 5 *10 = 50 runs. We use Python's `Random` module, which is a pseudo-random number generator, to create stratified 10-fold partitions, preserving the ratio between the two classes across all partitions. If initialized with the same seed number, `Random` would generate the exact same sequence of pseudo-random numbers. At the start of each algorithm's evaluation, we initialized the `Random` generator using the same seed to ensure the exact same sequence of training/testing partitions for all algorithms. As the model performance is normally distributed, we use paired sample t-tests to check if observed performance differences between two algorithms are statistically significant ($p < 0.05$). We use the "paired sample t-test" since our experimental setup guarantees that cross-validation runs of two different algorithms would get the same sequences of train/test partitions. We have included the results of the statistical tests in the replication package [76].

We conducted all evaluations on an Ubuntu 20.04 LTS workstation with an Intel i7-9700 CPU, 32 GB of RAM, and an NVIDIA Titan RTX GPU with 24 GB of memory. For the Python configuration, we created an Anaconda environment with Python 3.8.0, and `tensorflow` / `tensorflow-gpu` 2.5.0.

5.2 Baseline Algorithms

To establish baseline performance, we computed the performance of four existing toxicity detectors (Table 6) on our dataset. We briefly describe the four tools next:

(1) *PPA* [7] *(off-the-shelf)*: To prevent the online community from abusive content, Jigsaw and Google's Counter Abuse Technology team developed PPA [7]. Algorithms and datasets to train these models are not publicly available. PPA can generate the probability score of a text being toxic, servere_toxic, insult, profanity, threat, identity_attack, and sexually explicit. The score for each category is from 0 to 1, where the probability of a text belonging to that

category increases with the score. For our two class classification, we considered a text as toxic if its PPA score for the toxicity category is higher than 0.5.

(2) *STRUDEL tool* [73] *(off-the-shelf)*: The STRUDEL tool is an ensemble based on two existing tools: PPA and the Stanford politeness detector and BoW vector obtained from preprocessed text. Its classification pipeline obtains a toxicity score of a text using the PPA, computes the politeness score using the Stanford politeness detector tool [25], and computes BoW vector using Tf-Idf. For SE specificity, its Tf-Idf vectorizer excludes words that occur more frequently in the SE domain than in a non-SE domain. Although the STRUDEL tool also computes several other features such as sentiment score, subjectivity score, polarity score, number of LIWC anger words, and the number of emoticons in a text, none of these features contributed to improved performance during its evaluation [73]. Hence, the best-performing ensemble from STRUDEL uses only the PPA score, Stanford politeness score, and Tf-Idf vector. The off-the-shelf version is trained on a manually labeled dataset of 654 GitHub issues.

(3) *STRUDEL tool* [78] *(retrain)*: Due to several technical challenges, we were unable to retrain the STRUDEL tool using the source code provided in its repository [73]. Therefore, we wrote a simplified re-implementation based on the description included in the work and our understanding of the current source code. The primary author of the tool acknowledged our implementation as correct. Our implementation is publicly available inside the WSU-SEAL directory of the publicly available repository: https://github.com/WSU-SEAL/toxicity-detector. Our pull request with this implementation has been also merged to the original repository. For computing baseline performance, we conducted a stratified 10-fold cross validation using our code review dataset.

(4) *DPCNN* [77] *(retrain)*: We cross validated a DPCNN model [49] using our code review dataset. We include this model in our baseline since it provided the best retrained performance during our benchmark study [77].

Table 6 shows the performance of the four baseline models. Unsurprisingly, the two retrained models provide better performance than the off-the-shelf ones. Overall, the retrained Strudel tool provides the best scores among the four tools on all seven metrics. Therefore, we consider this model as the key baseline to improve on. The best toxicity detector among the ones participating in the 2020 SemEval challenge achieved a 0.92 F1-score on the Jigsaw dataset [91]. As the baseline models listed in Table 6 are evaluated on a different dataset, it may not be fair to compare these models against the ones trained on the Jigsaw dataset. However, the best baseline model's F1-score is 7 (i.e., 0.92−0.85) points lower than the ones from a non-SE domain. This result suggests that with existing technology, it may be possible to train SE domain specific toxicity detectors with better performance than the best baseline listed in Table 6.

Finding 1: *Retrained models provide considerably better performance than the off-the-shelf ones, with the retrained STRUDEL tool providing the best performance. Still, the $F1_1$-score from the best baseline model lags 7 points behind the $F1_1$-score of state-of-the-art models trained and evaluated on the Jigsaw dataset during the 2020 SemEval challenge [91].*

5.3 How Do the Algorithms Perform Without Optional Preprocessing?

The following sections detail the performance of the three groups of algorithms described in Section 4.5.

5.3.1 CLE Algorithms. The top five rows of Table 7 (i.e., the CLE group) show the performance of the five CLE models. Among those five algorithms, RF achieves significantly higher P_0 (0.956),

Table 7. Mean Performance of the 10 Selected Algorithms Based on 10-Fold Cross Validations

Group	Model	Vectorizer	Non-Toxic			Toxic			Accuracy (A)
			P_0	R_0	$F1_0$	P_1	R_1	$F1_1$	
CLE	DT	Tf-Idf	0.954	0.963	0.959	0.841	0.806	0.823	0.933
	GBT	Tf-Idf	0.926	0.985	0.955	0.916	0.672	0.775	0.925
	LR	Tf-Idf	0.918	0.983	0.949	0.900	0.633	0.743	0.916
	RF	Tf-Idf	0.956	0.982	0.969	0.916	0.810	0.859	0.949
	SVM	Tf-Idf	0.929	0.979	0.954	0.888	0.688	0.775	0.923
DNN1	DPCNN	word2vec	0.962	0.966	0.964	0.870	0.841	0.849	0.942
	DPCNN	GloVe	0.963	0.966	0.964	0.871	0.842	0.851	0.943
	DPCNN	fastText	0.964	0.967	0.965	0.870	0.845	0.852	0.944
DNN2	LSTM	word2vec	0.929	0.978	0.953	0.866	0.698	0.778	0.922
	LSTM	GloVe	0.944	0.971	0.957	0.864	0.758	0.806	0.930
	LSTM	fastText	0.936	0.974	0.954	0.853	0.718	0.778	0.925
DNN3	BiLSTM	word2vec	0.965	0.974	0.969	0.887	0.851	0.868	0.950
	BiLSTM	GloVe	0.965	0.976	0.968	0.895	0.828	0.859	0.948
	BiLSTM	fastText	0.966	0.974	0.970	0.888	0.854	0.871	0.951
DNN4	GRU	word2vec	0.964	0.976	0.970	0.894	0.847	0.870	0.951
	GRU	GloVe	0.965	0.977	0.971	0.901	0.851	0.875	0.953
	GRU	fastText	0.965	0.974	0.969	0.888	0.852	0.869	0.951
Transformer	BERT	bert	0.971	0.976	0.973	0.901	0.876	0.887	0.957

For each group, a shaded background indicates significant improvements over the others from the same group.

$F1_0$ (0.969), R_1 (0.81), $F1_1$ (0.859), and accuracy (0.949) than the four other algorithms from this group. The RF model also significantly outperforms (one sample t-test) the key baseline (i.e., retrained STRUDEL) in terms of the two key metrics: accuracy (A) and $F1_1$. Although STRUDEL retrain achieves better recall (R_1), our RF model achieves better precision (P_1).

5.3.2 *Deep Neural Networks.* We evaluated each of the four DNN algorithms using three different pre-trained word embedding techniques (i.e., word2vec, GloVe, and fastText) to identify the best-performing embedding combinations. Rows 6 through 17 (i.e., groups DNN1, DNN2, DNN3, and DNN4) of Table 7 show the performance of the four DNN algorithms using three different embeddings. For each group, statistically significant improvements (paired sample t-tests) over the other two configurations are highlighted using a shaded background. Our results suggest that the choice of embedding does influence the performance of the DNN algorithms. However, such variations are minor.

For DPCNN, only the R_1-score is significantly better with fastText than it is with GloVe or word2vec. The other scores do not vary significantly among the three embeddings. Based on these results, we recommend fastText for DPCNN in ToxiCR. For LSTM and GRU, GloVe boosts significantly better $F1_1$-scores than those based on fastText or word2vec. Since $F1_1$ is one of the key measures to evaluate our models, we recommend GloVe for both LSTM and GRU in ToxiCR. GloVe also boosts the highest accuracy for both LSTM (although not statistically significant) and GRU. For BiLSTM, since fastText provides significantly higher P_0, R_1, and $F1_1$-scores than those based on GloVe or word2vec, we recommend fastText for BiLSTM in ToxiCR. These results also suggest that three out of the four selected DNN algorithms (i.e., except LSTM) significantly outperform (one sample t-test) the key baseline (i.e., retrained STRUDEL) in terms of both accuracy and $F1_1$-score.

5.3.3 Transformer. The bottom row of Table 7 shows the performance of our BERT-based model. This model achieves the highest mean accuracy (0.957) and $F1_1$ (0.887) among all 18 models listed in Table 7. This model also outperforms the baseline STRUDEL retrain on all seven metrics.

Finding 2: *From the CLE group, RF provides the best performance. From the DNN group, GRU with GloVe provides the best performance. Among the 18 models from the six groups, BERT achieves the best performance. Overall, 10 out of the 18 models also outperform the baseline STRUDEL retrain model.*

5.4 Do Optional Preprocessing Steps Improve Performance?

For each of the 10 selected algorithms, we evaluated whether the optional preprocessing steps (especially SE domain specific ones) improve performance. Since ToxiCR includes three optional preprocessings (i.e., identifier splitting (*id-split*), keyword removal (*kwrd-remove*), and counting profane words (*profane-count*), we ran each algorithm with $2^3 = 8$ different combinations. For the DNN models, we did not evaluate all three embeddings in this step, as that would require evaluating 3*8 = 24 possible combinations for each one. Rather, we used only the best-performing embedding identified in the previous step (i.e., Section 5.3.2).

To select the best optional preprocessing configuration from the eight possible configurations, we use mean accuracy and mean $F1_1$-scores based on five-time 10-fold cross validations. We also used pair sampled t-tests to check whether any improvement over that of its base configuration, as listed in the Table 7 (i.e., no optional preprocessing selected), is statistically significant (paired sample t-test, $p < 0.05$).

Table 8 shows the best-performing configurations for all algorithms and the mean scores for those configurations. Checkmarks (✓) in the preprocessing columns for an algorithm indicate that the best configuration for that algorithm does use that preprocessing. To save space, we report the performance of only the best combination for each algorithm. Detailed results are available in our replication package [76].

These results suggest that optional preprocessing steps do improve the performance of the models. Notably, CLE models gained higher improvements than the other two groups. RF's accuracy improved from 0.949 to 0.955 and $F1_1$ improved from 0.859 to 0.879 with the *profane-count* preprocessing.

During these evaluations, other CLE models also achieved between 0.02 and 0.04 performance boosts in our key measures (i.e., A and $F1_1$). Improvements from optional preprocessing also depend on algorithm choices. The *profane-count* preprocessing improved performance of all the CLE models, whereas *kwrd-remove* improved all except RF. However, *id-split* improved none of the CLE models.

All DNN models also improved performance with the *profane-count* preprocessing. Contrasting with the CLE models, *id-split* was useful for three out of the four DNNs. *kwrd-remove* preprocessing improved only LSTM models. Noticeably, gains from optional preprocessing for the DNN models were less than 0.01 over that of the base configurations and statistically insignificant (paired sample t-test, $p > 0.05$) for most of the cases. Finally, although we noticed slight performance improvement (i.e., in A and $F1_1$) of the BERT model with *kwrd-remove*, the differences are not statistically significant. Overall, at the end of our extensive evaluation, we found the best-performing combination was a BERT model with *kwrd-remove* optional preprocessing. The best combination provides an 0.889 $F1_1$-score and 0.958 accuracy. The best-performing model also significantly outperforms (one sample t-test, $p < 0.05$) the baseline model (i.e, STRUDEL retrain in Table 6) in all seven performance measures.

Table 8. Best-Performing Configurations of Each Model with Optional Preprocessing Steps

Group	Algo	Vectorizer	Preprocessing			Non-Toxic			Toxic			A
			profane-count	**kwrd-remove**	**id-split**	P_0	R_0	$F1_0$	P_1	R_1	$F1_1$	
CLE	DT	Tf-Idf	✓	✓	–	0.960	0.968	0.964	0.862	0.830	0.845	0.942
	GBT	Tf-Idf	✓	✓	–	0.938	0.981	0.959	0.901	0.729	0.806	0.932
	LR	Tf-Idf	✓	✓	–	0.932	0.981	0.956	0.898	0.698	0.785	0.927
	RF	Tf-Idf	✓	–	–	0.964	**0.981**	0.972	**0.917**	0.845	0.879	0.955
	SVM	Tf-Idf	✓	✓	–	0.939	0.977	0.958	0.886	0.736	0.804	0.931
DNN	DPCNN	fasttext	✓	–	–	0.964	0.973	0.968	0.889	0.846	0.863	0.948
	LSTM	glove	✓	✓	✓	0.944	0.974	0.959	0.878	0.756	0.810	0.932
	BiLSTM	fasttext	✓	–	✓	0.966	0.975	0.971	0.892	0.858	0.875	0.953
	BiGRU	glove	✓	–	✓	0.966	0.976	0.971	0.897	0.856	0.876	0.954
Transformer	BERT	bert	–	✓	–	**0.970**	0.978	**0.974**	0.907	**0.874**	**0.889**	**0.958**

The shaded background indicates significant improvements over its base configuration (i.e., no optional preprocessing). For each column, a bold font indicates the highest value for that measure. A dagger (†) indicates an optional SE domain specific preprocessing step.

Table 9. Performance of ToxiCR on the Gitter Dataset

Mode	Models	Vectorizer	Non-Toxic			Toxic			Accuracy
			P	*R*	*F*1	*P*	*R*	*F*1	
Cross validation (retrain)	RF	Tf-Idf	0.851	0.945	0.897	0.879	0.699	0.779	0.859
	BERT	bert	0.931	0.909	0.919	0.843	0.877	0.856	0.898
Cross prediction (off-the-shelf)	RF	Tf-Idf	0.857	0.977	0.914	0.945	0.704	0.807	0.881
	BERT	bert	0.897	0.949	0.923	0.897	0.802	0.847	0.897

Table 10. Confusion Matrix for Our Best-Performing Model (i.e., BERT) for the Combined Code Review Dataset

		Predicted	
		Toxic	Non-Toxic
Actual	Toxic	3,259	483
	Non-Toxic	373	15,446

Finding 3: *Eight out of the 10 models (i.e., except SVM and DPCNN) achieved significant performance gains through SE domain preprocessing such as programming keyword removal and identifier splitting. Although keyword removal may be useful for all four classes of algorithms, identifier splitting is useful only for three DNN models. Our best model is based on BERT, which significantly outperforms the STRUDEL retrain model on all seven measures.*

5.5 How Do the Models Perform on Another Dataset?

To evaluate the generality of our models, we used the Gitter dataset of 4,140 messages from our benchmark study [77]. In this step, we conducted two types of evaluations. First, we ran 10-fold cross validations of the top CLE model (i.e., RF) and the BERT model using the Gitter dataset. Second, we evaluated cross-dataset prediction performance (i.e., off-the-shelf) by using the code review dataset for training and the Gitter dataset for testing.

The top two rows of Table 9 show the results of 10-fold cross validations for the two models. We found that the BERT model provides the best accuracy (0.898) and the best $F1_1$ (0.856). On the Gitter dataset, all seven performance measures achieved by the BERT model are lower than those on the code review dataset. This may be due to the smaller size of the Gitter dataset (4,140 texts) than the code review dataset (19,651 texts). The bottom two rows of Table 9 show the results of our cross predictions (i.e., off-the-shelf). Our BERT model achieved similar performance in terms of A and $F1_1$ in both modes. However, the RF model performed better on the Gitter dataset in cross-prediction mode (i.e., off-the-shelf) than in cross-validation mode. This result further supports our hypothesis that the performance drops of our models on the Gitter dataset may be due to smaller-size training data.

Finding 4: *Although our best-performing model provides higher precision off-the-shelf on the Gitter dataset than that from the retrained model, the latter achieves better recall. Regardless, our BERT model achieves similar accuracy and $F1_1$ during both off-the-shelf usage and retraining.*

5.6 What Are the Distributions of Misclassifications from the Best-Performing Model?

The best-performing model (i.e., BERT) misclassified only 856 texts out of the 19,651 texts from our dataset. There are 373 false positives and 483 false negatives. Table 10 shows the confusion matrix of the BERT model. To understand the reasons behind misclassifications, we adopted an open coding approach where two of the authors independently inspected each misclassified text

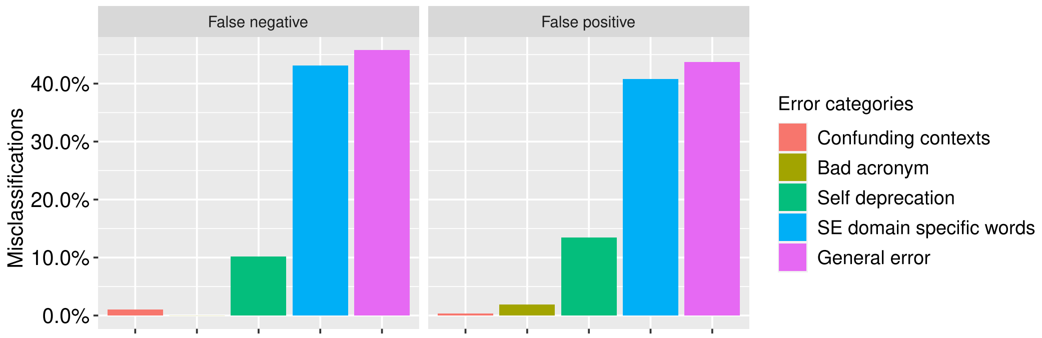


Fig. 3. Distribution of the misclassifications from the BERT model.

to identify general scenarios. Next, they had a discussion session, where they developed an agreed upon higher-level categorization scheme of five groups. With this scheme, those two authors independently labeled each misclassified text into one of those five groups. Finally, they compared their labels and resolved conflicts through mutual discussions.

Figure 3 shows distributions of the five categories of misclassifications from ToxiCR grouped by false positives and false negatives. The following sections detail those error categories.

5.6.1 General Errors. General Errors (GEs) are due to failures of the classifier to identify the pragmatic meaning of various texts. These errors represent 45% of the false positives and 46% of the false negatives. Many GE false positives are due to words or phrases that more frequently occur in toxic contexts and vice versa. For example, "If we do, should we just get rid of the HBoundType?" and "Done. I think they came from a messed up rebase." are two false positive cases, due to the phrases "get rid of" and "messed up" that have occurred more frequently in toxic contexts.

GE errors also occurred due to infrequent words. For example, "Oh, look. The stupidity that makes me rant so has already taken root. I suspect it's not too late to fix this, and fixing this rates as a mitzvah in my book" is incorrectly predicted as non-toxic as very few texts in our dataset include the word "stupidity." Another such instance was "this is another instance of uneducated programmers calling any kind of polymorphism overloading, please translate it to override" due to the word "uneducated." As we did not have many instances of identify attacks in our dataset, most of those were also incorrectly classified. For example, "most australian dummy var name ever!" was predicted as non-toxic by our classifier.

5.6.2 SE Domain Specific Words (SE). Words that have different meanings in the SE domain than its meaning in the general domain (die, dead, kill, junk, and bug) [77] were responsible for 40% false positives and 43% false negatives. For example, the text "you probably wanted 'die'here. eerror is not fatal." is incorrectly predicted as toxic due to the presence of the words "die" and "fatal." However, although the word "junk" is used to harshly criticize a code in the sentence "I don't actually need all this junk...," this sentence was predicted as non-toxic, as most of the code review comments from our dataset do not use "junk" in such a way.

5.6.3 Self-Deprecation (SD). Usage of self-deprecating texts to express humility is common during code reviews [59, 77]. We found that 13% of 373 false positives and 11% of 493 false negatives were due to the presence of self-deprecating phrases. For example, "Missing entry in kerneldoc above... (stupid me)" is labeled as "non-toxic" in our dataset but is predicted as "toxic" by our model. Although our model did classify many of the self-deprecating texts expressing humbleness correctly, those texts also led to some false negatives. For example, although "Huh? Am I stupid?

How's that equivalent?" was misclassified as non-toxic, it fits "toxic" according to our rubric due to its aggressive tone.

5.6.4 *Bad Acronym (BA).*

In a few cases, developers have used acronyms with alternate toxic expansion. For example, the webkit framework used the acronym "WTF" for "Web Template Framework"[6] for a namespace. Around 2% of our false-positive cases were comments referring to the "WTF" namespace from Webkit.

5.6.5 *Confounding Contexts (CC).*

Some of the texts in our dataset represent confounding contexts and were challenging even for the human raters to make a decision. Such cases represent 0.26% false positives and 1.04% false negatives. For example, "This is a bit ugly, but this is what was asked so I added a null ptr check for |inspector_agent_|. Let me know what you think." is a false-positive case from our dataset. We had labeled it as non-toxic, since the word "ugly" is applied to critique code written by the author of this text. However, "I just know the network stack is full of _bh poop. Do you ever get called from irq context? Sorry, I didn't mean to make you thrash." is labeled as toxic due to thrashing another person's code with the word "poop." However, the reviewer also said sorry in the next sentence. During labeling, we considered it as toxic since the reviewer could have critiqued the code in a nicer way. Probably due to the presence of mixed contexts, our classifier incorrectly predicted it as "non-toxic."

Finding 5: *Almost 85% of the misclassifications are due to either our model's failure to accurately comprehend the pragmatic meaning of a text (i.e., GE) or words having SE domain specific synonyms.*

6 IMPLICATIONS

Based on our design and evaluation of ToxiCR, we have identified following lessons.

Lesson 1: Development of a reliable toxicity detector for the SE domain is feasible. Despite creating an ensemble of multiple NLP models (i.e., PPA, sentiment score, politeness score, subjectivity, and polarity) and various categories of features (i.e., BoW, number of anger words, and emoticons), the STRUDEL tool achieved only a 0.57 F-score during their evaluation. Moreover, a recent study by Miller et al. [59] found false-positive rates as high as 98% [59]. However, the best model from the 2020 Semeval Multilingual Offensive Language Identification in Social Media task achieved an $F1_1$-score of 92.04% [92]. Therefore, the question remains as to whether we can build a SE domain specific toxicity detector that achieves similar performance (i.e., F1 = 0.92) as the ones from non-SE domains.

In designing ToxiCR, we adopted a different approach—that is, focusing on text preprocessing and leveraging state-of-the-art NLP algorithms rather than creating ensembles to improve performance. Our extensive evaluation with a large-scale SE dataset has identified a model that boosts 95.8% accuracy and 88.9% $F1_1$-score in identifying toxic texts. This model's performance is within 3% of the best one from a non-SE domain. This result also suggests that with a carefully labeled large-scale dataset, we can train an SE domain specific toxicity detector that achieves performance close to those of toxicity detectors from non-SE domains.

Lesson 2: Performance from RF's optimum configuration may be adequate if GPU is not available. Although a deep learning based model (i.e., BERT) achieved the best performance during our evaluations, that model is computationally expensive. Even with a high-end GPU such as Titan RTX, our BERT model required an average of 1,614 seconds for training. We found that RF-based models trained on a Core-i7 CPU took only 64 seconds on average.

[6]https://stackoverflow.com/questions/834179/wtf-does-wtf-represent-in-the-webkit-code-base.

During a classification task, RF generates the decision using majority voting from all subtrees. RF is suitable for high-dimensional noisy data like the ones found in text classification tasks [47]. With carefully selected preprocessing steps to better understand contexts (e.g., profanity count), RF may perform well for binary toxicity classification tasks. In our model, after adding profane count features, RF achieved an average accuracy of 95.5% and $F1_1$-score of 87.9%, which are within 1% of those achieved by BERT. Therefore, if computation cost is an issue, an RF-based model may be adequate for many practical applications. However, as our RF model uses a context-free vectorizer, it may perform poorly on texts, where prediction depends on surrounding contexts. Therefore, for a practical application, a user must take that limitation into account.

Lesson 3: Preprocessing steps do improve performance. We implemented five mandatory and three optional preprocessing steps in ToxiCR. The mandatory preprocessing steps do improve the performance of our models. For example, a DPCNN model without these preprocessing achieved 91% accuracy and 78% $F1_1$ (see Table 6). However, a model based on the same algorithm achieved 94.4% accuracy and 84.5% $F1_1$ with these preprocessing steps. Therefore, we recommend using both domain-specific and general preprocessing steps.

Two of our preprocessing steps are SE domain specific (i.e., identifier splitting, programming keywords removal). Our empirical evaluation of those steps (Section 5.4) suggest that 9 out of the 10 models (i.e., except SVM and DPCNN) achieved significant performance improvements through these steps. Although none of the models showed significant degradation through these steps, significant gains were dependent on algorithm selection, with CLE algorithms gaining only from keyword removal and identifier splitting improving only the DNN ones.

Lesson 4: Performance boosts from the optional preprocessing steps are algorithm dependent. The three optional preprocessing steps also improved performance of the classifiers. However, performance gains through the these steps were algorithm dependent. The `profane-count` preprocessing had the highest influence as 9 out of the 10 models gained performance with this step. On the other, `id-split` was the least useful one with only three DNN models gaining minor gains with this step. CLE algorithms gained the most with $\approx$ 1% boost in terms of accuracies and 1% to 3% in terms of $F1_1$-scores. However, DNN algorithms had relatively minor gains (i.e., less than 1%) in both accuracies and $F1_1$-scores. Since DNN models utilize embedding vectors to identify semantic representation of texts, those are less dependent on these optional preprocessing steps.

Lesson 5: Accurate identification of self-deprecating texts remains a challenge. Almost 11% (out of 856 misclassified texts) of the errors from our best-performing model were due to self-deprecating texts. Challenges in identifying such texts have been also acknowledged by prior toxicity detectors [41, 88, 93]. Due to the abundance of self-deprecating texts among code review interactions [59, 77], we believe that this can be an area to improve on for future SE domain specific toxicity detectors.

Lesson 6: Achieving even higher performance is feasible. Since 85% of errors are due to failures of our models to accurately comprehend the contexts of words, we believe that achieving further improved performance is feasible. Since supervised models learn better from larger training datasets, a larger dataset (e.g., Jigsaw dataset includes 160K samples) may enable even higher performance. Additionally, NLP is a rapidly progressing area with state-of-the-art techniques changing almost every year. Although we have not evaluated the most recent generation of models, such as GPT-3 [19] and XLNet [90], in this study, those may help achieve better performance, as they are better at identifying contexts.

7 THREATS TO VALIDITY

In the following, we discuss the four common types of threats to validity for this study.

7.1 Internal Validity

The first threat to validity for this study is our selection of data sources that come from four FOSS projects. Although these projects represent four different domains, many domains are not represented in our dataset. Moreover, our projects represent some of the top FOSS projects with organized governance. Therefore, several categories of highly offensive texts may be underrepresented in our datasets.

The notion of toxicity also depends on multitude of different factors such as culture, ethnicity, country of origin, language, and relationship between the participants. We did not account for any such factors during our dataset labeling.

7.2 Construct Validity

Our stratified sampling strategy was based on toxicity scores obtained from the PPA. Although we manually verified all the texts classified as "toxic" by the PPA, we randomly selected only (5,510[7] + 9,000 =14,510) texts that had PPA scores less than 0.5. Among those 14,510 texts, we identified only 638 toxic ones (4.4%). If both the PPA and our random selections missed some categories of toxic comments, instances of such texts may be missing in our datasets. Since our dataset is relatively large (i.e., 19,651), we believe that this threat is negligible.

According to our definition, *toxicity* is a large umbrella that includes various anti-social behaviors such as the use of offensive names, profanity, insults, threats, personal attacks, flirtations, and sexual references. Although our rubric is based on the Conversational AI team, we have modified it to fit a diverse and multicultural professional workplace such as an OSS project. As the notion of toxicity is a context-dependent complex phenomena, our definition may not fit many organizations, especially the homogeneous ones.

Researcher bias during our manual labeling process could also cause mislabeled instances. To eliminate such biases, we focused on developing a rubric first. With the agreed-upon rubric, two of the authors independently labeled each text and achieved "almost perfect" ($\kappa = 0.92$) inter-rater agreement. Therefore, we do not anticipate any significant threat arising from our manual labeling.

We did not change most of the hyper-parameters for the CLE algorithms and accepted the default parameters. Therefore, some of the CLE models may have achieved better performance on our datasets through parameter tuning. To address this threat, we used the `GridSearchCV` function from the scikit-learn library with the top two CLE models (i.e., RF and DT) to identify the best parameter combinations. Our implementation explored six parameters with a total of 5,040 combinations for RF and five parameters with 360 combinations for DT. Our results suggest that most of the default values are identical to those from the best-performing combinations identified through `GridSearchCV`. We also reevaluated RF and DT with the `GridSearchCV` suggested values but did not find any statistically significant (paired sample t-tests, $p > 0.05$) improvements over our already trained models.

For the DNN algorithms, we did not conduct extensive hyper-parameter search due to computational costs. However, parameter values were selected based on the best practice reported in the deep learning literature. Moreover, to identify the best DNN models, we used validation sets and `EarlyStopping`. Still we may not have been able to achieve the best possible performance from the DNN models during our evaluations.

7.3 External Validity

Although we have not used any project or code review specific preprocessing, our dataset may not adequately represent texts from other projects or other software development interactions such as

[7]Code review 1 dataset.

issue discussions, commit messages, or question/answers on StackExchange. Therefore, our pre-trained models may have degraded performance on other contexts. However, our models can be easily retrained using different labeled datasets from other projects or other types of interactions. To facilitate such retraining, we have made both the source code and instructions to retrain the models publicly available [76].

7.4 Conclusion Validity

To evaluate the performance our models, we have standard metrics such as accuracy, precision, recall, and F-scores. For the algorithm implementations, we extensively used state-of-the-art libraries such as scikit-learn [67] and TensorFlow [3]. We also used 10-fold cross validations to evaluate the performance of each model. Therefore, we do not anticipate any threats to validity arising from the set of metrics, supporting library selection, and evaluation of the algorithms.

8 CONCLUSION AND FUTURE DIRECTIONS

This article presented the design and evaluation of ToxiCR, a supervised learning-based classifier to identify toxic code review comments. ToxiCR includes a choice to select one of the ten supervised learning algorithms, an option to select text vectorization techniques, five mandatory and three optional processing steps, and a large-scale labeled dataset of 19,651 code review comments. With our rigorous evaluation of the models with various combinations of preprocessing steps and vectorization techniques, we have identified the best combination that boosts 95.8% accuracy and an 88.9% F1-score. We have released our dataset, pre-trained models, and source code publicly available on GitHub [76]. We anticipate this tool being helpful in combating toxicity among FOSS communities. As a future direction, we aim to conduct empirical studies to investigate how toxic interactions impact code review processes and their outcomes among various FOSS projects.

REFERENCES

[1] GitHub. 2018. Annotation Instructions for Toxicity with Sub-Attributes. Retrieved February 18, 2023 from https://github.com/conversationai/conversationai.github.io/blob/main/crowdsourcing_annotation_schemes/toxicity_with_subattributes.md.

[2] Kaggle. 2018. Toxic Comment Classification Challenge. Retrieved February 18, 2023 from https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283.

[4] Ashutosh Adhikari, Achyudh Ram, Raphael Tang, and Jimmy Lin. 2019. DocBERT: BERT for document classification. *arXiv preprint arXiv:1904.08398* (2019).

[5] Sonam Adinolf and Selen Turkay. 2018. Toxic behaviors in Esports games: Player perceptions and coping strategies. In *Proceedings of the 2018 Annual Symposium on Computer-Human Interaction in Play Companion Extended Abstracts*. 365–372.

[6] Toufique Ahmed, Amiangshu Bosu, Anindya Iqbal, and Shahram Rahimi. 2017. SentiCR: A customized sentiment analysis tool for code review interactions. In *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. IEEE, Los Alamitos, CA, 106–111.

[7] Perspective. n.d. Using machine learning to reduce toxicity online. Retrieved February 18, 2023 from https://www.perspectiveapi.com/.

[8] GitHub. 2018. Annotation Instructions for Toxicity with Sub-Attributes. Retrieved February 18, 2023 from https://github.com/conversationai/conversationai.github.io/blob/master/crowdsourcing_annotation_schemes/toxicity_with_subattributes.md.

[9] Basemah Alshemali and Jugal Kalita. 2020. Improving the reliability of deep neural networks in NLP: A review. *Knowledge-Based Systems* 191 (2020), 105210.

[10] Ashley A. Anderson, Sara K. Yeo, Dominique Brossard, Dietram A. Scheufele, and Michael A. Xenos. 2018. Toxic talk: How online incivility can undermine perceptions of media. *International Journal of Public Opinion Research* 30, 1 (2018), 156–168.

[11] Anonymous. 2014. Leaving Toxic Open Source Communities. Retrieved February 18, 2023 from https://modelviewculture.com/pieces/leaving-toxic-open-source-communities.

[12] Hayden Barnes. 2020. Toxicity in Open Source. Retrieved February 18, 2023 from https://boxofcables.dev/toxicity-in-linux-and-open-source/.

[13] Joseph Berkson. 1944. Application of the logistic function to bio-assay. *Journal of the American Statistical Association* 39, 227 (1944), 357–365.

[14] Meghana Moorthy Bhat, Saghar Hosseini, Ahmed Hassan, Paul Bennett, and Weisheng Li. 2021. Say "YES" to positivity: Detecting toxic language in workplace communications. In *Findings of the Association for Computational Linguistics: EMNLP 2021*. 2017–2029.

[15] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics* 5 (2017), 135–146.

[16] Amiangshu Bosu and Jeffrey C. Carver. 2013. Impact of peer code review on peer impression formation: A survey. In *Proceedings of the 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.* IEEE, Los Alamitos, CA, 133–142.

[17] Amiangshu Bosu, Anindya Iqbal, Rifat Shahriyar, and Partha Chakroborty. 2019. Understanding the motivations, Challenges and needs of blockchain software developers: A survey. *Empirical Software Engineering* 24, 4 (2019), 2636–2673.

[18] Leo Breiman. 1996. Bagging predictors. *Machine Learning* 24, 2 (1996), 123–140.

[19] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33*. 1877–1901.

[20] Fabio Calefato, Filippo Lanubile, and Nicole Novielli. 2017. EmoTxt: A toolkit for emotion recognition from text. In *Proceedings of the 2017 7th International Conference on Affective Computing and Intelligent Interaction Workshops and Demos (ACIIW'17)*. IEEE, Los Alamitos, CA, 79–80.

[21] Kevin Daniel André Carillo, Josianne Marsan, and Bogdan Negoita. 2016. Towards developing a theory of toxicity in the context of free/open source software & peer production communities. In *Proceedings of the SIGOPEN 2016 Developmental Workshop for Openness Research (ICIS'16)*.

[22] Hao Chen, Susan McKeever, and Sarah Jane Delany. 2019. The use of deep learning distributed representations in the identification of abusive text. In *Proceedings of the International AAAI Conference on Web and Social Media*, Vol. 13. 125–133.

[23] Savelie Cornegruta, Robert Bakewell, Samuel Withey, and Giovanni Montana. 2016. Modelling radiological language with bidirectional long short-term memory networks. In *Proceedings of the 7th International Workshop on Health Text Mining and Information Analysis*. 17–27.

[24] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine Learning* 20, 3 (1995), 273–297.

[25] Cristian Danescu-Niculescu-Mizil, Moritz Sudhof, Dan Jurafsky, Jure Leskovec, and Christopher Potts. 2013. A computational approach to politeness with application to social factors. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*. 250–259.

[26] R. Van Wendel De Joode. 2004. Managing conflicts in open source communities. *Electronic Markets* 14, 2 (2004), 104–113.

[27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 4171–4186. https://doi.org/10.18653/v1/N19-1423

[28] Maeve Duggan. 2017. Online Harassment 2017. Retrieved February 18, 2023 from https://www.pewresearch.org/internet/2017/07/11/online-harassment-2017/.

[29] Carolyn D. Egelman, Emerson Murphy-Hill, Elizabeth Kammer, Margaret Morrow Hodges, Collin Green, Ciera Jaspan, and James Lin. 2020. Predicting developers' negative feelings about code review. In *Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE'20)*. IEEE, Los Alamitos, CA, 174–185.

[30] Ahmed Elnaggar, Bernhard Waltl, Ingo Glaser, Jörg Landthaler, Elena Scepankova, and Florian Matthes. 2018. Stop illegal comments: A multi-task deep learning approach. In *Proceedings of the 2018 Artificial Intelligence and Cloud Computing Conference*. 41–47.

[31] Nelly Elsayed, Anthony S. Maida, and Magdy Bayoumi. 2019. Deep gated recurrent and convolutional network hybrid model for univariate time series classification. *International Journal of Advanced Computer Science and Applications* 10, 5 (2019), 654–664.

[32] Samir Faci. 2020. The Toxicity of Open Source. Retrieved February 18, 2023 from https://www.esamir.com/20/12/23/the-toxicity-of-open-source/.

[33] Isabella Ferreira, Jinghui Cheng, and Bram Adams. 2021. 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.

[34] Anna Filippova and Hichang Cho. 2016. The effects and antecedents of conflict in free and open source software development. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work and Social Computing*. 705–716.

[35] LibreOffice. n.d. The Document Foundation Code of Conduct. Retrieved February 18, 2023 from https://www.documentfoundation.org/foundation/code-of-conduct/.

[36] Jerome H. Friedman. 2001. Greedy function approximation: A gradient boosting machine. *Annals of Statistics* 29, 5 (2001), 1189–1232.

[37] Spiros V. Georgakopoulos, Sotiris K. Tasoulis, Aristidis G. Vrahatis, and Vassilis P. Plagianakos. 2018. Convolutional neural networks for toxic comment classification. In *Proceedings of the 10th Hellenic Conference on Artificial Intelligence*. 1–6.

[38] Alex Graves and Jürgen Schmidhuber. 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks* 18, 5-6 (2005), 602–610.

[39] Isuru Gunasekara and Isar Nejadgholi. 2018. A review of standard text classification practices for multi-label toxicity identification of online content. In *Proceedings of the 2nd Workshop on Abusive Language Online (ALW2'18)*. 21–25.

[40] Sanuri Dananja Gunawardena, Peter Devine, Isabelle Beaumont, Lola Garden, Emerson Rex Murphy-Hill, and Kelly Blincoe. 2022. Destructive criticism in software code review impacts inclusion. *Proceedings of the ACM on Human-Computer Interaction* 6, CSCW2 (2022), Article 292, 29 pages.

[41] Laura Hanu and Unitary Team. 2020. Detoxify. Retrieved February 18, 2023 from https://github.com/unitaryai/detoxify.

[42] Tin Kam Ho. 1995. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, Vol. 1. IEEE, Los Alamitos, CA, 278–282.

[43] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[44] Hossein Hosseini, Sreeram Kannan, Baosen Zhang, and Radha Poovendran. 2017. Deceiving Google's perspective API built for detecting toxic comments. *arXiv preprint arXiv:1702.08138* (2017).

[45] Zhiheng Huang, Wei Xu, and Kai Yu. 2015. Bidirectional LSTM-CRF models for sequence tagging. *arXiv preprint arXiv:1508.01991* (2015).

[46] N. Imtiaz, J. Middleton, J. Chakraborty, N. Robson, G. Bai, and E. Murphy-Hill. 2019. Investigating the effects of gender bias on GitHub. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. 700–711.

[47] Md. Zahidul Islam, Jixue Liu, Jiuyong Li, Lin Liu, and Wei Kang. 2019. A semantics aware random forest for text classification. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 1061–1070.

[48] Carlos Jensen, Scott King, and Victor Kuechler. 2011. Joining free/open source software communities: An analysis of newbies' first interactions on project mailing lists. In *Proceedings of the 2011 44th Hawaii International Conference on System Sciences*. IEEE, Los Alamitos, CA, 1–10.

[49] Rie Johnson and Tong Zhang. 2017. Deep pyramid convolutional neural networks for text categorization. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, Volume 1 (Long Papers)*. 562–570.

[50] Robbert Jongeling, Proshanta Sarkar, Subhajit Datta, and Alexander Serebrenik. 2017. On negative results when using sentiment analysis tools for software engineering research. *Empirical Software Engineering* 22, 5 (2017), 2543–2584.

[51] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. Retrieved February 18, 2023 from https://dblp.org/rec/journals/corr/KingmaB14.html.

[52] Kamran Kowsari, Donald E. Brown, Mojtaba Heidarysafa, Kiana Jafari Meimandi, Matthew S. Gerber, and Laura E. Barnes. 2017. HDLTex: Hierarchical deep learning for text classification. In *Proceedings of the 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA'17)*. IEEE, Los Alamitos, CA, 364–371.

[53] Deepak Kumar, Patrick Gage Kelley, Sunny Consolvo, Joshua Mason, Elie Bursztein, Zakir Durumeric, Kurt Thomas, and Michael Bailey. 2021. Designing toxic content classification for a diversity of perspectives. In *Proceedings of the 17th Symposium on Usable Privacy and Security (SOUPS'21)*. 299–318.

[54] Keita Kurita, Anna Belova, and Antonios Anastasopoulos. 2019. Towards robust toxic content classification. *arXiv preprint arXiv:1912.06872* (2019).

[55] Zijad Kurtanović and Walid Maalej. 2018. On user rationale in software engineering. *Requirements Engineering* 23, 3 (2018), 357–379.

[56] Bin Lin, Fiorella Zampetti, Gabriele Bavota, Massimiliano Di Penta, Michele Lanza, and Rocco Oliveto. 2018. Sentiment analysis for software engineering: How far can we go? In *Proceedings of the 40th International Conference on Software Engineering*. 94–104.

[57] Ilya Loshchilov and Frank Hutter. 2018. Decoupled weight decay regularization. In *Proceedings of the International Conference on Learning Representations.*

[58] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems 26.* 3111–3119.

[59] Courtney Miller, Sophie Cohen, Daniel Klug, Bodgan Vasilescu, and Christian Kästner. 2022. "Did you miss my comment or what?" Understanding toxicity in open source discussions. In *Proceedings of the International Conference on Software Engineering (ICSE'22).* IEEE, Los Alamitos, CA.

[60] Pushkar Mishra, Helen Yannakoudakis, and Ekaterina Shutova. 2018. Neural character-based composition models for abuse detection. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP'18).* 1.

[61] Murtuza Mukadam, Christian Bird, and Peter C. Rigby. 2013. Gerrit software code review data from Android. In *Proceedings of the 2013 10th Working Conference on Mining Software Repositories (MSR'13).* IEEE, Los Alamitos, CA, 45–48.

[62] Dawn Nafus, James Leach, and Bernhard Krieger. 2006. *FLOSSPOLS Deliverable D 16 Gender: Integrated Report of Findings.* FLOSSPOLS.

[63] Chikashi Nobata, Joel Tetreault, Achint Thomas, Yashar Mehdad, and Yi Chang. 2016. Abusive language detection in online user content. In *Proceedings of the 25th International Conference on World Wide Web.* 145–153.

[64] Nicole Novielli, Daniela Girardi, and Filippo Lanubile. 2018. A benchmark study on sentiment analysis for software engineering research. In *Proceedings of the 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR'18).* IEEE, Los Alamitos, CA, 364–375.

[65] OpenStack. n.d. OpenStack Code of Conduct. Retrieved February 18, 2023 from https://wiki.openstack.org/wiki/Conduct.

[66] Rajshakhar Paul, Amiangshu Bosu, and Kazi Zakia Sultana. 2019. Expressions of sentiments during code reviews: Male vs. female. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'19).* IEEE, Los Alamitos, CA.

[67] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[68] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP'14).* 1532–1543.

[69] Android Open Source Project. n.d. Code of Conduct. Retrieved February 18, 2023 from https://source.android.com/setup/cofc.

[70] Huilian Sophie Qiu, Bogdan Vasilescu, Christian Kästner, Carolyn Denomme Egelman, Ciera Nicole Christopher Jaspan, and Emerson Rex Murphy-Hill. 2022. Detecting interpersonal conflict in issues and code review: Cross pollinating open-and closed-source approaches. In *Proceedings of the 2022 ACM/IEEE 44th International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS'22).* 41–55.

[71] J. Ross Quinlan. 1986. Induction of decision trees. *Machine Learning* 1, 1 (1986), 81–106.

[72] Israr Qureshi and Yulin Fang. 2011. Socialization in open source software projects: A growth mixture modeling approach. *Organizational Research Methods* 14, 1 (2011), 208–238.

[73] Naveen Raman, Minxuan Cao, Yulia Tsvetkov, Christian Kästner, and Bogdan Vasilescu. 2020. Stress and burnout in open source: Toward finding, understanding, and mitigating unhealthy interactions. In *Proceedings of the International Conference on Software Engineering, New Ideas, and Emerging Results (ICSE'20).* ACM, New York, NY.

[74] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning representations by back-propagating errors. *Nature* 323, 6088 (1986), 533–536.

[75] Maarten Sap, Dallas Card, Saadia Gabriel, Yejin Choi, and Noah A. Smith. 2019. The risk of racial bias in hate speech detection. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics.* 1668–1678.

[76] Jaydeb Sarkar, Asif Turzo, Ming Dong, and Amiangshu Bosu. 2022. ToxiCR: Replication Rackage. Retrieved February 18, 2023 from https://github.com/WSU-SEAL/ToxiCR.

[77] Jaydeb Sarker, Asif Kamal Turzo, and Amiangshu Bosu. 2020. A benchmark study of the contemporary toxicity detectors on software engineering interactions. In *Proceedings of the 2020 27th Asia-Pacific Software Engineering Conference (APSEC'20).* 218–227. https://doi.org/10.1109/APSEC51365.2020.00030

[78] Jaydeb Sarker, Asif Kamal Turzo, Ming Dong, and Amiangshu Bosu. 2022. WSU SEAL implementation of the STRUDEL toxicity detector. Retrieved February 18, 2023 from https://github.com/WSU-SEAL/toxicity-detector/tree/master/WSU_SEAL.

[79] Carl-Erik Särndal, Bengt Swensson, and Jan Wretman. 2003. *Model Assisted Survey Sampling*. Springer Science & Business Media.

[80] Robert E. Schapire. 2003. The boosting approach to machine learning: An overview. In *Nonlinear Estimation and Classification*. Lecture Notes in Computer Science, Vol. 171. Springer, 149–171.

[81] Megan Squire and Rebecca Gazda. 2015. FLOSS as a source for profanity and insults: Collecting the data. In *Proceedings of the 2015 48th Hawaii International Conference on System Sciences*. IEEE, Los Alamitos, CA, 5290–5298.

[82] Saurabh Srivastava, Prerna Khurana, and Vartika Tewari. 2018. Identifying aggression and toxicity in comments using capsule network. In *Proceedings of the 1st Workshop on Trolling, Aggression, and Cyberbullying (TRAC'18)*. 98–105.

[83] Igor Steinmacher and Marco Aurélio Gerosa. 2014. How to support newcomers onboarding to open source software projects. In *Proceedings of the IFIP International Conference on Open Source Systems*. 199–201.

[84] Pannavat Terdchanakul, Hideaki Hata, Passakorn Phannachitta, and Kenichi Matsumoto. 2017. Bug or not? Bug report classification using n-gram IDF. In *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*. IEEE, Los Alamitos, CA, 534–538.

[85] Ameya Vaidya, Feng Mai, and Yue Ning. 2020. Empirical analysis of multi-task learning for reducing identity bias in toxic comment detection. In *Proceedings of the International AAAI Conference on Web and Social Media (ICWSM'20)*. 683–693.

[86] Betty van Aken, Julian Risch, Ralf Krestel, and Alexander Löser. 2018. Challenges for toxic comment classification: An in-depth error analysis. In *Proceedings of the 2nd Workshop on Abusive Language Online (ALW2'18)*. 33–42.

[87] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30*.

[88] Susan Wang and Zita Marinho. 2020. Nova-Wang at SemEval-2020 Task 12: OffensEmblert: An ensemble ofoffensive language classifiers. In *Proceedings of the 14th Workshop on Semantic Evaluation*. 1587–1597.

[89] Mengzhou Xia, Anjalie Field, and Yulia Tsvetkov. 2020. Demoting racial bias in hate speech detection. In *Proceedings of the 8th International Workshop on Natural Language Processing for Social Media*. 7–14.

[90] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R. Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized autoregressive pretraining for language understanding. In *Advances in Neural Information Processing Systems 32*.

[91] Sara Zaheri, Jeff Leath, and David Stroud. 2020. Toxic comment classification. *SMU Data Science Review* 3, 1 (2020), 13.

[92] Marcos Zampieri, Preslav Nakov, Sara Rosenthal, Pepa Atanasova, Georgi Karadzhov, Hamdy Mubarak, Leon Derczynski, Zeses Pitenis, and Çağrı Çöltekin. 2020. SemEval-2020 Task 12: Multilingual offensive language identification in social media (OffensEval 2020). In *Proceedings of the 14th Workshop on Semantic Evaluation*. 1425–1447.

[93] Justine Zhang, Jonathan P. Chang, Cristian Danescu-Niculescu-Mizil, Lucas Dixon, Yiqing Hua, Nithum Tahin, and Dario Taraborelli. 2018. Conversations gone awry: Detecting early signs of conversational failure. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, Vol. 1.

[94] Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *Advances in Neural Information Processing Systems 28*. 649–657.

[95] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV'15)*.

Received 24 February 2022; revised 5 December 2022; accepted 17 January 2023