Automated Identification of Toxic Code Reviews: How Far Can We Go?

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

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 to maintain healthy interactions among its members. However, off-the-shelf toxicity detectors perform poorly on Software Engineering (SE) dataset, such as one curated from code review comments. To encounter 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 ten supervised learning algorithms, an option to select text vectorization techniques, five mandatory and three optional SE domain specific processing steps, and a large scale labeled dataset of 19,571 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 for our dataset that boosts 95.8% accuracy and 88.9% $F_1$ score. ToxiCR significantly outperforms existing toxicity detectors on our dataset. We have released our dataset, pretrained models, evaluation results, and source code publicly 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, Amiangshu Bosu. 2022. Automated Identification of Toxic Code Reviews: How Far Can We Go?. ACM Trans. Softw. Eng. Methodol. 31, 0, Article 0 (2022), 27 pages. https://doi.org/10.1145/nmmn.nnnn

1 INTRODUCTION

Communications among the members of many Free and Open Source Software (FOSS) communities include manifestations of toxic behaviours [11, 29, 40, 53, 56, 68]. These toxic communications may have decreased the productivity of those communities by wasting valuable work hours [15, 61]. FOSS developers being frustrated over peers with ‘prickly’ personalities [16, 30] may contemplate leaving a community for good [10, 24]. Moreover, as most FOSS communities rely on contributions from volunteers, attracting and retaining prospective joiners is crucial for the growth and survival of FOSS projects [60]. However, toxic
interactions with existing members may pose barriers against successful onboarding of newcomers [41, 70]. 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, those are rarely enforced due to time constraints [10]. 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, "...it’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." [10]. 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 [6, 8, 34, 46]. However, off-the-shelf toxicity detectors do not work well on Software Engineering (SE) communications [65], 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 [5, 65]. 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 [61, 65].

To encounter this challenge, Raman et al. developed a toxicity detector tool (referred as the ‘STRUDEL tool’ hereinafter) for the SE domain [61]. However, as the STRUDEL tool was trained and evaluated with only 611 SE texts, its reliability is questionable. Recently, Sarker et al. 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 [65]. The results of their analyses suggest that none of the existing tools are reliable in identifying toxic texts from SE communications, since all the five tools’ performances 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 supervised models using their datasets. Being motivated by these results, we hypothesize that a SE domain specific toxicity detector can boost even better performances, since off-the-shelf toxicity detectors do not use SE domain specific preprocessing steps, such as URL removal and preprocessing of code snippets. On this hypothesis, this paper presents ToxiCR, a SE domain specific toxicity detector. ToxiCR is trained and evaluated using a manually labeled dataset of 19,571 code review comments selected from four popular FOSS communities (i.e., Android, Chromium OS, OpenStack and LibreOffice). ToxiCR is written in Python using the Scikit-learn [57] and TensorFlow [2]. It provides an option to train models using one of the ten supervised machine learning algorithms including five classical and ensemble-based, four deep neural network-based, and a Bidirectional Encoder Representations from Transformer (BERT) based ones. It also includes five mandatory and three optional preprocessing steps and an option to use three different types of vectorization techniques (i.e., word2vec, GloVe, and fastText) with the deep neural network-based algorithms. We empirically evaluated the eight possible \(2^3 = 8\) optional preprocessing combinations for each of the ten algorithms to identify the best
performing combination. During our 10-fold cross-validations evaluations, the best performing model of ToxiCR significantly outperforms existing toxicity detectors on the code review dataset with an accuracy of 95.8% and F-score of 88.9%.

The primary contributions of this paper are:
- ToxiCR: An SE domain specific toxicity detector. ToxiCR is publicly available on Github at: https://github.com/WSU-SEAL/ToxiCR.
- An empirical evaluation of ten machine learning algorithms to identify toxic SE communications.
- An empirical evaluation of three optional preprocessing steps in improving the performances of toxicity classification models.
- Empirical identification of the best possible combination for each of the ten algorithms.

**Paper organization:** The remainder of this paper is organized as following. 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 paper.

## 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?

Whether a communication should be considered as ‘toxic’ depends on a multitude of factors, such as communication medium, location, culture, and 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 to leave the discussion forum [6]. On the other hand, the Pew Research Center marks a text as toxic if it contains threat, offensive call, or sexually expletive words [26]. Anderson et al.’s definition of toxic communication also includes insulting language or mockery [9]. Adinolf and Turkay 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[4]. Since FOSS communities are professional workplaces, Sarker et al. adopted an expansive view of toxic communication for the SE domain by including all of the following: offensive name calling, insults, threats, personal attacks, flirtations, any reference to sexual activities, and swearing or cursing [65].

### 2.2 Toxic communications in FOSS communities

Several prior studies have identified toxic communications in FOSS communities [19, 56, 61, 65, 68]. Squire and Gazda 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 [68]. More alarmingly, they identified sexist ‘maternal insults’ being used by many developers. Recent studies have also reported toxic communications among issue discussions on Github [61], and during code reviews [56, 65].

Although toxic communications are rare in FOSS communities [65], toxic interactions can have severe consequences [19]. Carillo et al. termed Toxic communications as a ‘poison’ that impacts mental health of FOSS developers [19], and may contribute to stress and burnouts [19, 61]. When the level of toxicity increases in a FOSS community, the community may disintegrate as developers may no longer wish to be associated with that community [19]. 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 [41, 70]. Miller et al. conducted a qualitative study to better understand toxicity in the context of FOSS development [50]. They created a sample of 100 Github issues representing various types of toxic interactions such as insults, arrogance, trolling, entitlement, and unprofessional behaviour. Their analyses also suggest that toxicity in FOSS communities differ from those observed on other online platforms such as Reddit or Wikipedia [50].

2.3 State of the art toxicity detectors

To combat abusive online contents, Google’s Jigsaw AI team developed the Perspective API (PPA), which is publicly available [6]. 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 the PPA [38], the Jigsaw team periodically updates it to eliminate identified limitations. The Jigsaw team also published a guideline [7] to manually identify toxic contents and used that guideline to curate a crowd-sourced labeled dataset of toxic online contents [1]. This dataset has been used to train several deep neural network based toxicity detectors [20, 27, 32, 34, 69, 73]. Recently, Bhat et al. proposed ToxiScope, a supervised learning-based classifier to identify toxic on workplace communications [13]. 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 [46, 51, 54]. Although, character-level encoding based models can handle such character level obfuscations, they come with significant increments of computation times [46]. 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 nontoxic text marked as toxic) [63, 72, 76].

However, off-the-shelf toxicity detectors suffer significant performance degradation on SE datasets [65]. Such degradation is not surprising, since prior studies found off-the-shelf natural language processing (NLP) tools also performing poorly on SE datasets [5, 43, 55]. Raman et al. created the STRUDEL tool, an SE domain specific toxicity detector [61], by leveraging the PPA tool and a customized version of Stanford’s Politeness Detector [23]. Sarker et al. investigated the performance of the STRUDEL tool and four other off-the-shelf toxicity detectors on two SE datasets [65]. 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 using their SE datasets.

3 RESEARCH CONTEXT

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

3.1 Supervised machine learning algorithms

For ToxiCR we selected ten supervised ML algorithms from the ones that have been commonly used for text classification tasks. Our selection includes three classical, two ensemble methods based, four deep neural network (DNN) based, and a Bidirectional Encoder Representations from Transformer (BERT) based algorithms. Following subsections 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 previously used for classification of SE texts [5, 18, 47, 71, 71].

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. And the decision nodes are where the data is split into two or more sub-nodes [59].
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 [12].
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 [22].

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 methods based algorithms, based on prior SE studies [5, 47, 71].

1. Random Forest (RF): RF is an ensemble based method that combines the results produced by multiple decision trees [36]. RF creates independent decision trees and combines them in parallel using on the ‘bagging’ approach [17].
2. Gradient-Boosted Decision Trees (GBT): Similar to RF, GBT is also an ensemble based method using decision trees [31]. 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 [67].

3.1.3 **Deep neural networks:** In recent years, DNN based models have shown significant performance gains over both classical and ensemble based models in text classification tasks [45, 79]. 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 learnt from the past [62]. However, traditional RNNs may perform poorly on 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 proposed LSTM, a new type of RNN architecture, that overcomes the challenges posed by long term dependencies using a gradient-based learning algorithm [37]. LSTM consists four units: i) input gate, which decides what information to add from current step, ii) forget gate, which decides what is to keep from prior steps, iii) output gate, which determines the next hidden state, and iv) memory cell, 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 [21, 33]. In this architecture, the forward LSTM takes input sequences in the forward direction to model information from the past, while the backward LSTM takes input sequences in the reverse direction to model information from the future [39]. BiLSTM has been shown to be performing better than the unidirectional LSTM in several text classification tasks, as it can identify language contexts better than LSTM [33].
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, update gate and reset gate. The reset gate decides what information should be forgot for next pass and update gate determines...
which information should pass to next step. Unlike LSTM, GRU does not require any memory cell, and therefore needs shorter training time than LSTM [28].

(4) Deep Pyramid CNN (DPCNN): Convolutional neural networks (CNN) are a specialized type of neural networks 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 proposed a special type of CNN architecture, named deep pyramid convolutional neural network (DPCNN) for text classification tasks [42]. 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 a 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, 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’, a technique to identify the important features from different positions of an input sequence [74].

In this study, we select Bidirectional Encoder Representations from Transformers, which is commonly known as BERT [25]. BERT based models have achieved remarkable performances in various NLP tasks, such as question answering, sentiment classification, and text summarization [3, 25]. 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 machine learning models can work on. Bag-of-Words (BOW) is one of the most basic representation techniques, that turns an arbitrary text into 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. On the other hand, 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; ii) contextualized word embeddings aim at capturing word semantics in different contexts to address the issue of polysemous (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. Following subsections provide a brief overview of those techniques.

3.2.1 Tf-Idf: TF-IDF is a BOW based vectorization technique that evaluates how relevant a word is to a document in a collection of documents. TF-IDF score for a word is computed by multiplying two metrics: how many times a word appears in a document \((T_f)\), and the inverse document frequency of the word across a set of documents \((I_df)\). Following equations show the computation steps for Tf-Idf scores.

\[
T_f(w, d) = f(w, d) \frac{1}{f(t, d)}
\]
Fig. 1. A simplified overview of ToxiCR showing key pipeline

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

$$Idf(w) = \log_e \left( \frac{N}{w_N} \right)$$

Here, $N$ is the total number of documents and $w_N$ represents the number of documents having $w$. Finally, we computed TfIdf score of a word as:

$$TfIdf(w, d) = Tf(w, d) \times Idf(w)$$

3.2.2 Word2vec: In 2013, Mikolaev et al. [49] proposed Word2vec, a context free word embedding technique. It is based on two neural network models named Continuous Bag-of-Words (CBOW) and Skip-gram. CBOW predicts a target word based on its context, while 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. [58] Global Vectors for Word Representation (GloVe) is an unsupervised algorithm to create context free word embedding. Unlike word2vec, GloVe generates vector space from 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 [14]. While 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 [14].

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 [25]. 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
Table 1. An 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         | 13,038        | 2,447   | 10,591      |
| Gitter dataset        | 4,140         | 1,468   | 2,672       |
| Code review (combined)| 19,571        | 3,757   | 15,819      |

features. Finally, output vectors are used to train and validate our supervised learning-based models. The following subsections detail the research steps to design ToxiCR.

4.1 Training Dataset Creation

As of May 2021, there are three publicly available labeled datasets of toxic communications from the SE domain. Raman et al.’s dataset created for the STRUDEL tool [61] includes only 611 texts. In our recent benchmark study (referred 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 as ‘code review dataset 1’ hereinafter), i.e., Android, Chromium OS, and LibreOffice; ii) a dataset of 4,140 Gitter messages selected from the Gitter Ethereum channel (referred as ‘gitter dataset’ hereinafter) [65]. We followed the exact same process used in the benchmark study to select and label additional 13,038 code review comments selected from the OpenStack projects. In the following, we briefly describe our four-step process, which is detailed in our prior publication [65].

4.1.1 Data Mining. In the benchmark, we wrote a Python script to mine the Gerrit [52] managed code review repositories of three popular FOSS projects, i.e., 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 Paul et al. [56] to identify the bot accounts and exclude code review comments authored by the bots.

4.1.2 Stratified sampling of code review comments. Since toxic communications are rare [5, 65] 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. [66]. We used Google’s Perspective API (PPA) [6] 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 4,038 comments with PPA scores greater than 0.5. In addition to those 4,038 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 to 0.4.

4.1.3 Manual Labeling. Using the rubric developed in the benchmark study [65], two of the authors independently labeled the 13,038 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 12,528 (96.1%) texts during this process and achieved a Cohen’s Kappa (κ) score of 0.92 (i.e., an
Table 2. Examples of text preprocessing steps implemented in ToxiCR

| Step      | Original                                      | Post Preprocessing          |
|-----------|-----------------------------------------------|------------------------------|
| URL-rem   | ah crap. Not sure how I missed that. [link](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-clm   | hahaha... loooooooser!                      | hahaha... 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 pre-processing step

almost perfect agreement)\(^1\). 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 13,038 texts. We refer to this dataset as ‘code review dataset 2’ hereinafter. Table 1 provides an overview of the three dataset used in this study.

### 4.1.4 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 dataset into a single dataset (referred as ‘combined code review dataset’ hereinafter). We believe such merging is not problematic, due to 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 dataset are picked from the same type of repository (i.e., Gerrit based code reviews).

The merged code review dataset includes total 19,571 code review comments, where 3,757 comments (19.2%) are labeled as ‘toxic’.

### 4.2 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, since 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. Table 2 shows examples of texts before and after preprocessing.

#### 4.2.1 Mandatory preprocessing

ToxiCR implements the following five mandatory pre-processing steps.

- **URL removal (URL-rem):** A code review comment may include an 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 shortened form of one or two words, are common among code review texts. For example, some common words are: doesn’t → does not, we’re

\(^1\)Kappa (\(\kappa\)) values are commonly interpreted as follows: values ≤ 0 as indicating ‘no agreement’ and 0.01 – 0.20 as ‘none to slight’, 0.21 – 0.40 as ‘fair’, 0.41 – 0.60 as ‘moderate’, 0.61–0.80 as ‘substantial’, and 0.81–1.00 as ‘almost perfect agreement’.  

ACM Trans. Softw. Eng. Methodol., Vol. 31, No. 0, Article 0. Publication date: 2022.
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 `) 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 a 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 36 commonly used profane words. Our preprocessor replaces each identified case with its correctly spelled form.

### 4.2.2 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 to 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 think 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. While the remaining seven pre-processing steps modify an input text pre-vectorization, this step adds an additional dimension to the post vectored output of a text.

### 4.3 Word Vectorizers

ToxiCR includes 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, Tfidf works only with the classical and ensemble (CLE) methods, Word2vec, GloVe, and fastText work with the deep neural network based algorithms, and BERT model includes its pre-trained vectorizer. For vectorizers, we chose the following implementations.

1. **Tfidf:** 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, pretrained 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 pretrained 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 GloVe model and contains 300 dimensional vectors for 2 million words.

(5) **BERT**: We select a variant of BERT model published as `BERT_en_uncased`. This model was pre-trained on a dataset of 2.5 billion words from the Wikipedia and 800 million words from the Bookcorpus [80].

### 4.4 Architecture of the ML Models

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

#### 4.4.1 Classical and ensemble methods

We have used the scikit-learn [57] implementations of the CLE classifiers.

- **Decision Tree (DT)**: We have used the `DecisionTreeClassifier` class with default parameters.
- **Logistic Regression (LR)**: We have used the `LogisticRegression` class with default parameters.
- **Support-Vector Machine (SVM)**: Among the various SVM implementations offered by scikit-learn, we have selected the `LinearSVC` class with default parameters.
- **Random Forest (RF)**: We have 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 have used the `GradientBoostingClassifier` class from the scikit-learn library. We set `n_iter_no_change = 5`, which stops the training early if the last 5 iterations did not achieve any improvement in accuracy. We accepted the default values for the other parameters.

#### 4.4.2 Deep Neural Networks Model

We used the version 2.5.0 of the TensorFlow library [2] for training the four deep neural network models (i.e., LSTM, BiLSTM, GRU, and DPCNN). Common parameters of the four models are:

- 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 the three pre-trained word embedding models use 300 dimensional vectors to represent words and phrases, we have set embedding size to 300.
- The embedding layer takes input embedding matrix as inputs. Each of word ($w_i$) 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 $300 \times n$, where $n$ is the total number of words in that text.
- Since we are developing binary classifiers, we have selected `binary_crossentropy` loss function for model training.
- We have selected the `Adam` optimizer (Adaptive Moment Estimation) [44] 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 [64].

- **Input Embedding Layer**: After preprocessing of code review texts, those are converted to input matrix. Embedded layer maps input matrix to a fixed dimension input embedding matrix. We used three pre-trained embeddings which help the model to capture the low level semantics using position based texts.

- **Hidden State Layer**: This layer takes the position wise embedding matrix and helps to 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 [42], we set 7 convolution blocks with Conv1D layer after the input embedding layer. We also set the other parameters of DPCNN model following [42]. Outputs from each of the CNN blocks is 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 LSTM unit to capture the hidden sequence from input embedding vector. LSTM unit generates the high dimensional semantic representation vector. To reshape the output dimension, we use flatten and dense layer after LSTM unit.
  - **BiLSTM blocks**: For text classification tasks, BiLSTM works better than LSTM for capturing the semantics of long sequence of text. Our model uses 50 units of Bidirectional LSTM 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 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 input embedding vector. To keep the most important features from GRU units, we set a concatenation of GlobalAveragePooling1D and GlobalMaxPooling1D layers. GlobalAveragePooling1D calculates the average of entire sequence of each vector and GlobalMaxPooling1D finds the maximum value of entire sequence.

- **Classifier Layer**: The output vector of hidden state layer project 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 0 to 1 range.

### 4.4.3 Transformer models

Among the several pre-trained BERT models\(^2\) we have used bert\(_{en\_uncased}\), which is also known as the BERT\(_{base}\) model. We downloaded the models from the tensorflow\_hub\(^3\), which consists trained machine learning models ready for fine tuning. Our BERT model architecture is as following:

- **Input layer**: takes the preprocessed input text from our SE dataset. To fit into BERT pretrained encoder, we preprocess each text using matching preprocessing model (i.e. bert\(_{en\_uncased\_preprocess}\)).

\(^2\)https://github.com/google-research/bert
\(^3\)https://tfhub.dev/tensorflow/bert\(_{en\_uncased\_preprocess}\)/3
Table 3. An overview of the hyper parameters for our deep neural networks and transformers

| Hyper-Parameters | Deep neural networks (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               |

- **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 is passed to a dropout layer with a probability of 0.1 to drop an input.
- **Classifier Layer**: Outputs from the dropout layer is passed to a two-unit dense layer, which transforms the outputs into two-dimensional vectors. From these vectors, a one-unit dense layer with linear activation function generates the probabilities of each text being toxic. Unlike deep neural network’s output layer, we have found that linear activation function provides better accuracy than non-linear ones (e.g, $\text{relu}$, $\text{sigmoid}$) for the BERT-based models.
- **Parameters**: Similar to the deep neural network models, We use $\text{binary} \ _\text{crossentropy}$ as the loss function and $\text{Binary Accuracy}$ as the evaluation metric during training.
- **Optimizer**: We set the optimizer as $\text{Adamw}$ [48] which improved the generalization performance of ‘adam’ optimizer. $\text{Adamw}$ minimizes the prediction loss and does regularization by decaying weight. Following the recommendation of Devlin et al. [25], we set the initial learning rate to $3e-5$.

### 4.5 Model Training and Validation

Following subsections detail our model training and validation approaches.

#### 4.5.1 Classical and ensembles

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

#### 4.5.2 DNN and Transformers

We have customized several hyper-parameters of the DNN models to train our models. Table 3 provides an overview of those customized hyper-parameters. A DNN model can be overfitted due to over training. To encounter that, we have 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 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 have set maximum 40 epochs\(^4\) for the DNN models and maximum 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 have used an `EarlyStopping` function from the Keras library, which monitors `minimum val loss`. If the performance of a model on the validation

\(^4\) the number times that a learning algorithm will work through the entire training dataset
dataset starts to degrade (e.g. loss begins to increase or accuracy begins to drop), then the training process is stopped.

4.6 Tool interface

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

```python
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" ,
          "shitty code"]

results=clf.get_toxicity_class(sentences)
```

Listing 1. Example usage of ToxiCR to classify toxic texts

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

- Algorithm Selection: Users can select one of the ten included algorithms by using the `--algo ALGO` option.
Automated Identification of Toxic Code Reviews: How Far Can We Go?

- 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. Default value is 5.
- Embedding: ToxiCR includes five different vectorization techniques: \texttt{tfdif}, \texttt{word2vec}, \texttt{glove}, \texttt{fasttext}, and \texttt{bert}. \texttt{tfdif} is configured to be used only with the CLE models. \texttt{word2vec}, \texttt{glove}, and \texttt{fasttext} can be used only with the DNN models. Finally, \texttt{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 underscore.
- 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 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 hyperparameters to identify the optimum set.

5 EVALUATION

We empirically evaluated the ten algorithms included in ToxiCR to identify the best possible configuration to identify toxic texts from our datasets. Following subsections detail our experimental configurations and the results of our evaluations.

5.1 Experimental Configuration

To evaluate the performance of our models, we use precision, recall, f-score, 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 F1-score for the toxic class (i.e., $F_1$) 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 $5 \times 10 = 50$ runs. We use Python’s `random` module, which is a pseudo-random number generator, to create 10-fold 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 performances are 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
Table 4. Performances of the four contemporary toxic detectors to establish a baseline performance. For our classifications, we consider toxic texts as the 'class 1' and non-toxic texts as the 'class 0'.

| Models            | Non-toxic | Toxic | Accuracy |
|-------------------|-----------|-------|----------|
|                   | $P_0$     | $R_0$ | $F_{10}$ | $P_1$  | $R_1$ | $F_{11}$ |
| Perspective API   | 0.92      | 0.79  | 0.85     | 0.45   | 0.70  | 0.55     | 0.776   |
| Strudel Tool      | 0.93      | 0.76  | 0.83     | 0.43   | 0.77  | 0.55     | 0.757   |
| Detoxify [35]     | 0.86      | 0.96  | 0.91     | 0.66   | 0.35  | 0.46     | 0.776   |
| DPCNN retrain [65]| **0.94**  | 0.95  | **0.94** | **0.81** | 0.76  | **0.78** | **0.91** |

our experimental setup guarantees 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 [64].

We conducted all evaluations on an Ubuntu 20.04 LTS workstation with Intel i7-9700 CPU, 32GB RAM, and an NVIDIA Titan RTX GPU with 24 GB memory. For 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 performances, we computed the performances of four existing toxicity detectors (Table 4) on our dataset. Among those tools, Perspective API is proprietary and is trained with a large labeled dataset by the Jigsaw AI team [6]. Strudel tool [61] uses output from the Perspective API for classification. We selected Detoxify [35] by the Unitary AI, since it uses a fine tuned BERT model and achieved top-tier performances in a recent Kaggle toxic comment classification challenge [1]. We use those three tools off-the-shelf. Since a DPCNN based model achieved the best performance in our previous benchmark study [65], we also selected that model. We used 10-fold cross-validation to retrain and evaluate DPCNN using our combined code review dataset.

Table 4 shows the performances of the existing toxicity detectors on our combined code review dataset. Among the four tools, the three off-the-shelf ones have lower $P_1$ and $F_{11}$ than the retrained model. The retrained models’ $R_1$ and $F_{11}$ scores are considerably lower than that for the non-toxic class. Since, the retrained DPCNN model performed the best in terms of both accuracy and $F_{11}$, we consider this model as the key baseline to improve on.

5.3 How do the algorithms perform without optional preprocessing?

Following subsections detail the performances of the three groups of algorithm described in the Section 4.4.

5.3.1 Classical and Ensemble (CLE) algorithms. The top five rows of the Table 5 (i.e., CLE group) show the performances of the five CLE models. Among, those five algorithms, RF achieves significantly higher $P_0$ (0.956), $F_{10}$ (0.969), $R_1$ (0.81), $F_{11}$ (0.859) and accuracy (0.949) than the four other algorithms from this group. Two out of the five models (i.e., RF and DT) from this group also significantly outperform (One-sample t-test) the key baseline (i.e., retrained DPCNN) in terms of accuracy and $F_{11}$-score.

5.3.2 Deep Neural Networks (DNN). 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 to 17 (i.e., groups: DNN1, DNN2, DNN3, and DNN4) of the Table 5 show the performances of the four DNN algorithms using three different embeddings. For each group, statistically significant improvements (paired-sample t-tests) over the other two configurations
Table 5. Mean performances of the ten selected algorithms based on 10-fold cross validations. For each group, shaded background indicate significant improvements over the others from the same group.

| Group | Models  | Vectorizer | Non-toxic | Toxic | Accuracy (A) |
|-------|---------|------------|-----------|-------|--------------|
|       |         |            | $P_0$ | $R_0$ | $F_1$ | $P_1$ | $R_1$ | $F_1$ |     |
| CLE   | DT      | tfidf      | 0.954 | 0.963 | 0.959 | 0.841 | 0.806 | 0.823 | 0.933 |
|       | GBT     | tfidf      | 0.926 | 0.985 | 0.955 | 0.916 | 0.672 | 0.775 | 0.925 |
|       | LR      | tfidf      | 0.918 | 0.983 | 0.949 | 0.900 | 0.633 | 0.743 | 0.916 |
|       | RF      | tfidf      | 0.956 | 0.982 | 0.969 | 0.916 | 0.810 | 0.859 | **0.949** |
|       | SVM     | tfidf      | 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 | en uncased | 0.971 | 0.976 | 0.973 | 0.901 | 0.876 | 0.887 | 0.957 |

are highlighted using shaded background. Our results suggest that choice of embedding does influence performances of the DNN algorithms. However, such variations are minor.

For DPCNN, only $R_1$ 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 $F_1$ scores than those based on fastText or word2vec. Since $F_1$ is one of the key measures to evaluate our models, we recommend the 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 $F_1$ scores than those based on GloVe or word2vec, we recommend fastText for BiLSTM in ToxiCR. These results also suggest that all the four selected DNN algorithms significantly outperform (one-sample t-test) the key baseline (i.e., retrained DPCNN) in terms of both accuracy and $F_1$-score.

5.3.3 **Transformer**. The bottom row of Table 5 shows the performance of our BERT based model. This model achieves the highest mean accuracy (0.957) and $F_1$ (0.887) among all the 18 models listed in Table 5. This model also outperforms the baseline DPCNN retrain on all the seven metrics.
Table 6. Best performing configurations of each model with optional preprocessing steps. Shaded background indicates significant improvements over its base configuration (i.e., no optional preprocessing). For each column, bold font indicates the highest value for that measure.

| Group | Algo | Vectorizer | Preprocessing | Non-toxic | Toxic | A |
|-------|------|------------|---------------|-----------|-------|---|
|       |      |            | profane-count | kwd-remove | id-split | $P_0$ | $R_0$ | $F_{10}$ | $P_1$ | $R_1$ | $F_{11}$ |
| CLE   | DT   | tfidf      | ✓             | ✓         | -       | 0.960 | 0.968 | 0.964 | 0.862 | 0.830 | 0.845 | 0.942 |
|       | GBT  | tfidf      | ✓             | ✓         | -       | 0.938 | 0.981 | 0.959 | 0.901 | 0.729 | 0.806 | 0.932 |
|       | LR   | tfidf      | ✓             | ✓         | -       | 0.932 | 0.981 | 0.956 | 0.898 | 0.698 | 0.785 | 0.927 |
|       | RF   | tfidf      | ✓             | -         | -       | 0.964 | 0.981 | 0.972 | 0.917 | 0.845 | 0.879 | 0.955 |
|       | SVM  | tfidf      | ✓             | ✓         | -       | 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  | -             | ✓         | -       | 0.970 | 0.978 | 0.974 | 0.907 | 0.874 | 0.889 | 0.958 |
5.4 Do optional preprocessing steps improve performance?

For each of the ten selected algorithms, we evaluated whether the optional preprocessing steps improve performances. Since ToxiCR includes three optional preprocessing (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 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 $F_1$ scores based on five time 10-fold cross validations. We also used pair sampled t-tests to check whether any improvement over its base configuration’s, as listed in the Table 5 (i.e., no optional preprocessing selected), is statistical significant (paired sample t-test, $p < 0.05$).

Table 6 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 pre-processing. To save space, we report the performances of only the best combination for each algorithm. Detailed results are available in our replication package [64].

These results suggest that optional pre-processing steps do improve the performances 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 $F_1$ improved from 0.859 to 0.879 with the profane-count preprocessing. During these evaluations, other CLE models also achieved between 0.02 to 0.04 performance boosts in our key measures (i.e., $A$ and $F_1$). Improvements from optional preprocessing also depend on algorithm choices. While the profane-count preprocessing improved performances of all the CLE models, kwrd-remove improved all except RF. On the other hand, id-split improved none of the CLE models.

All the DNN models also improved performances with the profane-count preprocessing. Contrasting 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 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 $F_1$) 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 0.889 $F_1$ score and 0.958 accuracy. The best performing model also significantly outperforms (one sample t-test, $p < 0.05$) the baseline model (i.e, DPCNN retrain in Table 4) in all the seven performance measures.

5.5 How do the models perform on another dataset?

To evaluate the generality of our models, we have used the Gitter dataset of 4,140 messages from our previous study [65]. 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 by using the code review dataset for training and the Gitter dataset for testing.

The top two rows of the Table 7 shows 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 $F_1$ (0.856). On the Gitter dataset, all the seven performance measures achieved by the BERT model are lower than those on the code review dataset. It may be due to the smaller size of the Gitter dataset (4,140 texts) than the code review dataset (19,571 texts). The bottom two rows of the Table 7 shows the results of our cross-domain predictions. Our BERT model achieved similar performances in terms of $A$ and $F_1$ in both...
Table 7. Performance of ToxiCR on Gitter dataset

| Mode             | Models       | Vectorizer   | Non-toxic | Toxic | Accuracy |
|------------------|--------------|--------------|-----------|-------|----------|
|                  |              |              | P         | R     | F1       | P         | R     | F1     |         |          |
| Cross-validation | RF           | TfIdf        | 0.851     | 0.945 | 0.897    | 0.879     | 0.699 | 0.779  | 0.859   |          |
|                  | BERT         | BERT-en-uncased | 0.931     | 0.909 | 0.919    | 0.843     | 0.877 | 0.856  | 0.898   |          |
| Cross-prediction | RF           | TfIdf        | 0.857     | 0.977 | 0.914    | 0.945     | 0.704 | 0.807  | 0.881   |          |
|                  | BERT         | BERT-en-uncased | 0.897     | 0.949 | 0.923    | 0.897     | 0.802 | 0.847  | 0.897   |          |

Table 8. Confusion Matrix for our best performing model (i.e., BERT) for the combined code review dataset

\[
\begin{array}{ccc}
\text{Actual} & \text{Toxic} & \text{Non-toxic} \\
\text{Toxic} & 3259 & 483 \\
\text{Non-toxic} & 373 & 15,446 \\
\end{array}
\]

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 19,571 texts from our dataset. Table 8 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 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 (FP) and False Negatives (FN). Following subsections detail those error categories.
5.6.1 General errors (GE). General errors 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 due to the presence of 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 to 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) [65] were responsible for 40% false positives and 43% false negatives. For example, the text “you probably wanted ‘die‘ here. error is not fatal.”, is incorrectly predicted as toxic due to the presence of the words ‘die‘ and ‘fatal‘. On the other hand, 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 [50, 65]. We found that almost 13% false positives and 10% 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 SD 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 few cases, developers have used acronyms with with alternate toxic expansion. For example, the webkit framework used the acronym ‘WTF’ -‘Web Template Framework’⁵, 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. On the other hand, “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’.

⁵https://stackoverflow.com/questions/834179/wtf-does-wtf-represent-in-the-webkit-code-base
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.**

According to our extensive evaluation with a large scale SE dataset, we have identified a model that has 95.8% accuracy and boosts 88.9% $F_1$ score in identifying toxic texts. This model’s performances are within 3% of the best model from the ‘2020 Semeval Multilingual Offensive Language Identification in Social Media task’, where the top team achieved an $F_1$ score of 92.04% [77]. This result suggests that with a carefully labeled large scale dataset, we can train an SE domain specific toxicity detector that achieves performances that are close to those of toxicity detectors from non-SE domains.

**Lesson 2: Performance from Random Forest’s optimum configuration may be adequate.**

While a deep learning-based model (i.e., BERT) achieved the best performances during our evaluations, that model is computationally expensive. Even with a high end GPU such as Titan RTX, our BERT model required on average 1,614 seconds for training. We found that RandomForest based models trained on a Core-i7 CPU took only 64 seconds on average. Yet, with the optional profane-count preprocessing, it achieved an average accuracy of 95.5% and $F_1$-score of 87.9%, which are within 1% of those from BERT. Therefore, if computation cost is an issue, a RandomForest based model may be adequate for many practical applications.

**Lesson 3: SE Domain specific preprocessing steps improve performances.**

We have implemented five mandatory and three optional preprocessing steps in ToxiCR. The mandatory preprocessing steps do improve performances of our models. For example, a DPCNN model without these preprocessing achieved 91% accuracy and 78% $F_1$ (Table 4). On the other hand, a model based on the same algorithm achieved 94.4% accuracy and 84.5% $F_1$ with these preprocessing steps. Therefore, we recommend experimenting with these steps for SE domain specific NLP tools.

**Lesson 4: Performance boosts from the optional preprocessing steps are algorithm dependent.**

The three optional preprocessing steps also improved performances of the classifiers. However, performance gains through the these steps were algorithm dependent. The profane-count preprocessing had the highest influence as nine out of the ten 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 1-3% in terms of $F_1$ scores. On the other hand, DNN algorithms had relatively minor gains (i.e., less than 1%) in both accuracies and $F_1$ 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% 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 [35, 75, 78]. Due to the abundance of self-deprecating texts among code review interactions [50, 65], we believe that this can be an area to improve on for future SE domain specific toxicity detectors.

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 which come from four FOSS projects. While 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 perspective API. Although, we manually verified all the texts classified as ‘toxic’ by the PPA, we randomly selected only \(5,510 + 9,000 = 14,510\) texts that had PPA scores of 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., 19,571), we believe that this threat is negligible.

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 hyperparameters for the CLE algorithms and accepted the default parameters. Therefore, some of the CLE models may have achieved better performances 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., RandomForest and DecisionTree) to identify the best parameter combinations. Our implementation explored six parameters with total 5,040 combinations for RandomForest and five parameters with 360 combinations for DecisionTree. 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 hyperparameter 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 used EarlyStopping. Still we may not have been able to achieve the best possible performances from the DNN models during our evaluations.

7.3 External validity
Although, we have not used any project or code review specific pre-processing, our dataset may not adequately represent texts from other projects or other software development interactions such as issue discussions, commit messages, or question/answers on StackExchange. Therefore, our pretrained models may have degraded performances on other contexts. However, our models can be easily retrained using a 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 [64].

\(^6\)Code review 1 dataset
7.4 Conclusion validity

To evaluate the performances of our models, we have standard metrics such as accuracy, precision, recall, and F-scores. For the algorithm implementations, we have extensively used state-of-the-art libraries such as scikit-learn [57] and TensorFlow [2]. We also used 10-fold cross-validations to evaluate the performances 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 paper presents 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 SE domain specific processing steps, and a large-scale labeled dataset of 19,571 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 88.9% F1 score.

We have released our dataset, pretrained models, and source code publicly available on Github [64]. 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] 2018. Toxic Comment Classification Challenge. https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge
[2] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). USENIX Association, Savannah, GA, 265–283.
[3] Ashutosh Adhikari, Achyudh Ram, Raphael Tang, and Jimmy Lin. 2019. Dochert: Bert for document classification. arXiv preprint arXiv:1904.08398 (2019).
[4] 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.
[5] Toufique Ahmed, Amiangshu Bosu, Anindya Iqbal, and Shahram Rahimi. 2017. SentiCR: a customized sentiment analysis tool for code review interactions. In 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 106–111.
[6] Conversation AI. [n.d.]. What if technology could help improve conversations online? https://www.perspectiveapi.com/
[7] Conversation AI. 2018. Annotation instructions for Toxicity with sub-attributes. https://github.com/conversationai/conversationai.github.io/blob/master/crowdsourcing_annotation_schemes/toxicity_with_subattributes.md.
[8] Basemah Abshemali and Jugal Kalita. 2020. Improving the reliability of deep neural networks in NLP: A review. Knowledge-Based Systems 191 (2020), 105210.
[9] 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.
[10] Anonymous. 2014. Leaving Toxic Open Source Communities. https://modelviewculture.com/pieces/leaving-toxic-open-source-communities
[11] Hayden Barnes. 2020. Toxicity in Open Source. https://boxofcables.dev/toxicity-in-linux-and-open-source/
[12] Joseph Berkson. 1944. Application of the logistic function to bio-assay. Journal of the American statistical association 39, 227 (1944), 357–365.
[13] 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.

[14] 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.

[15] Amiangshu Bosu and Jeffrey C Carver. 2013. Impact of peer code review on peer impression formation: A survey. In 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. IEEE, 133−142.

[16] 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.

[17] Leo Breiman. 1996. Bagging predictors. Machine learning 24, 2 (1996), 123−140.

[18] Fabio Caletfato, Filippo Lamibile, and Nicole Novielli. 2017. EmoTxt: a toolkit for emotion recognition from text. In 2017 seventh international conference on Affective Computing and Intelligent Interaction Workshops and Demos (ACIIW). IEEE, 79−80.

[19] Kevin Daniel André Carillo, Josianne Marsan, and Bogdan Negoița. 2016. Towards Developing a Theory of Toxicity in the Context of Free/Open Source Software & Peer Production Communities. SIGOPEN 2016 (2016).

[20] 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.

[21] Savelie Cornegruta, Robert Bakewell, Samuel Withey, and Giovanni Montana. 2016. Modelling radiological language with bidirectional long short-term memory networks. arXiv preprint arXiv:1609.08409 (2016).

[22] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. Machine learning 20, 3 (1995), 273−297.

[23] Cristian Danescu-Niculescu-Mizil, Moritz Sudhof, Dan Jurafsky, Jure Leskovec, and Christopher Potts. 2013. A computational approach to politeness with application to social factors. arXiv preprint arXiv:1306.6078 (2013).

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

[25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018).

[26] Maeve Duggan. 2017. Online harassment 2017. (2017).

[27] 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.

[28] Nelly Elsayed, Anthony S Maida, and Magdy Bayouni. 2018. Deep gated recurrent and convolutional network hybrid model for univariate time series classification. arXiv preprint arXiv:1812.07683 (2018).

[29] Samir Faci. 2020. The Toxicity Of Open Source. https://www.esamir.com/20/12/23/the-toxicity-of-open-source/ (2020).

[30] 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 & Social Computing. 705−716.

[31] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. Annals of statistics 26, 2 (2001), 1189−1232.

[32] 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.

[33] 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.

[34] 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). 21−25.

[35] Laura Hanu and Unitary team. 2020. Detoxify. Github. https://github.com/unitaryai/detoxify.

[36] Tin Kam Ho. 1995. Random decision forests. In Proceedings of 3rd international conference on document analysis and recognition, Vol. 1. IEEE, 278−282.

[37] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. Neural computation 9, 8 (1997), 1735−1780.

[38] Hossein Hosseini, Sreram Kannan, Baosen Zhang, and Radha Poovendran. 2017. Deceiving google's perspective api built for detecting toxic comments. arXiv preprint arXiv:1702.08138 (2017).

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

ACM Trans. Softw. Eng. Methodol., Vol. 31, No. 0, Article 0. Publication date: 2022.
[40] N. Imtiaz, J. Middleton, J. Chakraborty, N. Robson, G. Bai, and E. Murphy-Hill. 2019. Investigating the Effects of Gender Bias on GitHub. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). 700–711.

[41] 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 2011 44th Hawaii international conference on system sciences. IEEE, 1–10.

[42] 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.

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

[44] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014).

[45] Kamran Kowsari, Donald E Brown, Mojtaha Heidarysafa, Kiana Jafari Meimandi, Matthew S Gerber, and Laura E Barnes. 2017. Iditex: Hierarchical deep learning for text classification. In 2017 16th IEEE international conference on machine learning and applications (ICMLA). IEEE, 364–371.

[46] Keita Kurita, Anna Belova, and Antonios Anastasopoulos. 2019. Towards Robust Toxic Content Classification. arXiv preprint arXiv:1912.06872 (2019).

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

[48] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. arXiv preprint arXiv:1711.05101 (2017).

[49] 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. 3111–3119.

[50] 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 International Conference on Software Engineering, ICSE, ACM (2022) (ICSE). IEEE, ACM.

[51] Pushkar Mishra, Helen Yannakoudakis, and Ekaterina Shutova. 2018. Neural character-based composition models for abuse detection. arXiv preprint arXiv:1809.00378 (2018).

[52] Rajshakhar Paul, Amiangshu Bosu, and Kazi Zakia Sultana. 2019. Expressions of Sentiments during Code Reviews: Male vs. Female. In Proceedings of the 26th International Conference on Software Engineering, ICSE, ACM (2019). 700–711.

[53] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. the Journal of machine Learning research 12 (2011), 2825–2830.

[54] 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). 1532–1543.

[55] J. Ross Quinlan. 1986. Induction of decision trees. Machine learning 1, 1 (1986), 81–106.

[56] Israr Qureshi and Yulin Fang. 2011. Socialization in open source software projects: A growth mixture modeling approach. International Conference on Software Engineering, New Ideas and Emerging Results (ICSE). ACM, TBD.

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

[58] 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.
Automated Identification of Toxic Code Reviews: How Far Can We Go?

Jaydeb Sarkar, Asif Turzo, Ming Dong, and Amiangshu Bosu. 2022. ToxiCR: Replication package. Github. https://github.com/WSU-SEAL/ToxiCR.

Jaydeb Sarkar, Asif Kamal Turzo, and Amiangshu Bosu. 2020. A Benchmark Study of the Contemporary Toxicity Detectors on Software Engineering Interactions. (2020), 218–227.

Carl-Erik Särndal, Bengt Swensson, and Jan Wretman. 2003. Model assisted survey sampling. Springer Science & Business Media.

Robert E Schapire. 2003. The boosting approach to machine learning: An overview. Nonlinear estimation and classification (2003), 149–171.

Megan Squire and Rebecca Gazda. 2015. FLOSS as a Source for Profanity and Insults: Collecting the Data. In 2015 48th Hawaii International Conference on System Sciences, IEEE, 5290–5298.

Saurabh Srivastava, Prerna Khurana, and Vartika Tewari. 2018. Identifying aggression and toxicity in comments using capsule network. In Proceedings of the First Workshop on Trolling, Aggression and Cyberbullying (TRAC-2018). 98–105.

Igor Steinmacher and Marco Aurélio Gerosa. 2014. How to support newcomers onboarding to open source software projects. In IFIP International Conference on Open Source Systems. Springer, 199–201.

Pannavat Terdchanakul, Hideaki Hata, Passakorn Phannachitta, and Kenichi Matsumoto. 2017. Bug or not? bug report classification using n-gram idf. In 2017 IEEE international conference on software maintenance and evolution (ICSME). IEEE, 534–538.

Ameya Vaidya, Feng Mai, and Yue Ning. 2019. Empirical Analysis of Multi-Task Learning for Reducing Model Bias in Toxic Comment Detection. arXiv preprint arXiv:1909.09758 (2019).

Betty van Aken, Julian Risch, Ralf Krestel, and Alexander Löscher. 2018. Challenges for toxic comment classification: An in-depth error analysis. arXiv preprint arXiv:1809.07572 (2018).

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. arXiv preprint arXiv:1706.03762 (2017).

Susan Wang and Zita Marinho. 2020. Nova-Wang at SemEval-2020 Task 12: OffensEmblert: An Ensemble of Offensive Language Classifiers. In Proceedings of the Fourteenth Workshop on Semantic Evaluation. 1587–1597.

Mengzhou Xia, Anjaliie Field, and Yulia Tsvetkov. 2020. Demoting Racial Bias in Hate Speech Detection. arXiv preprint arXiv:2005.12246 (2020).

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 Fourteenth Workshop on Semantic Evaluation. 1425–1447.

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.

Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. Advances in neural information processing systems 28 (2015), 649–657.

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 The IEEE International Conference on Computer Vision (ICCV).