Neural Network-based Detection of Self-Admitted Technical Debt: From Performance to Explainability

XIAOXUE REN, Zhejiang University, China
ZHENCHANG XING, Australian National University, Australia
XIN XIA, Monash University, Australia
DAVID LO, Singapore Management University, Singapore
XINYU WANG, Zhejiang University, China
JOHN GRUNDY, Monash University, Australia

Technical debt is a metaphor to reflect the tradeoff software engineers make between short-term benefits and long-term stability. Self-admitted technical debt (SATD), a variant of technical debt, has been proposed to identify debt that is *intentionally introduced* during software development, e.g., temporary fixes and workarounds. Previous studies have leveraged human-summarized patterns (which represent n-gram phrases that can be used to identify SATD) or text-mining techniques to detect SATD in source code comments. However, several characteristics of SATD features in code comments, such as vocabulary diversity, project uniqueness, length, and semantic variations, pose a big challenge to the accuracy of pattern or traditional text-mining-based SATD detection, especially for cross-project deployment. Furthermore, although traditional text-mining-based method outperforms pattern-based method in prediction accuracy, the text features it uses are less intuitive than human-summarized patterns, which makes the prediction results hard to explain. To improve the accuracy of SATD prediction, especially for cross-project prediction, we propose a Convolutional Neural Network– (CNN) based approach for classifying code comments as SATD or non-SATD. To improve the explainability of our model's prediction results, we exploit the computational structure of CNNs to identify key phrases and patterns in code comments that are most relevant to SATD. We have conducted an extensive set of experiments with 62,566 code comments from 10 open-source projects and a user study with 150 comments of another three projects. Our evaluation confirms the effectiveness of different aspects of our approach and its superior performance, generalizability, adaptability, and explainability over current state-of-the-art traditional text-mining-based methods for SATD classification.

CCS Concepts: • **Software and its engineering** → *Software maintenance tools*;

Additional Key Words and Phrases: Self-admitted technical debt, convolutional neural network, cross project prediction, model explainability, model generalizability, model adaptability

This research was partially supported by the National Key Research and Development Program of China (2018YFB1003904), NSFC Program (No. 61602403), Project of Science and Technology Research and Development Program of China Railway Corporation (P2018X002), and the Fundamental Research Funds for the Central Universities.
Authors' addresses: X. Ren and X. Wang, Zhejiang University, College of Computer Science and Technology, Hangzhou, Zhejiang, 31000, China; emails: {xxren, wangxinyu}@zju.edu.cn; Z. Xing, Australian National University, College of Engineering and Computer Science, Canberra, Australia; email: Zhenchang.Xing@anu.edu.au; X. Xia and J. Grundy, Monash University, Faculty of Information Technology, Melbourne, VIC, Australia; emails: {xin.xia, john.grundy}@monash.edu; D. Lo, Singapore Management University, School of Information Systems, Singapore, Singapore; email: davidlo@smu.edu.sg.

1049-331X/2019/07-ART15 $15.00
https://doi.org/10.1145/3324916

ACM Reference format:
Xiaoxue Ren, Zhenchang Xing, Xin Xia, David Lo, Xinyu Wang, and John Grundy. 2019. Neural Network-based Detection of Self-Admitted Technical Debt: From Performance to Explainability. *ACM Trans. Softw. Eng. Methodol.* 28, 3, Article 15 (July 2019), 45 pages.
https://doi.org/10.1145/3324916

1 INTRODUCTION

Technical debt, proposed by Cunningham [1], is used to describe a debt that is accumulated when developers make wrong or unhelpful technical decisions—either intentionally or unintentionally—during software development. Zazworka et al. stressed that if these debts are not repaid, they can do great harm to the overall quality of software and cause cascading future defects in the long term [2]. A number of prior works have aimed to analyze technical debt in software engineering, to better support software maintenance and ensure high software quality, cf. References [3–5].

In recent years, Potdar and Shihab proposed the concept of self-admitted technical debt (SATD) [6], which considers debt that is intentionally introduced. For example, the comment "*Temporary—until we figure a better API*" in the Apache Ant project indicates that the corresponding code is a temporary solution that needs to be changed in future for better result. Further work of Wehaibi et al. [7] confirms that although the percentage of SATD in a project is not high, it can have a negative impact on software complexity. More concretely, they found that source code files that contain SATD have more bug-fixing changes, while files without SATD have more defects [7].

Many studies have utilized natural language processing (NLP) to analyze such comments from source code to detect possible defects [8, 9]. NLP methods have also been used to identify SATD. Potdar and Shihab [6] manually summarised 62 patterns that can be used to recognize SATD from comments in Java project source code. Recently, Huang et al. [10] proposed a text-mining-based approach to identify SATD. Compared with the 62 patterns, Huang et al.'s text-mining approach achieves a substantial improvement on F1-score. However, unlike the 62 intuitive patterns summarized by Potdar and Shihab [6], Huang et al.'s text-mining approach cannot provide the rationale for the prediction results. Furthermore, Huang et al.'s experiments show that using a text-mining approach has limited generalizability and adaptability for cross-project settings.

In this article, we first identify five characteristics of SATD comments that affect the performance, generalizability, and adaptability of pattern-based SATD detection [6] and traditional text-mining-based SATD classification [10]. To improve the accuracy of SATD prediction, especially for cross-project prediction, and also to improve the interpretability of machine-learning-based prediction results, we propose a Convolution Neural Network– (CNN) based approach to identify SATDs from source code comments. Our approach learns to extract the informative text features for the SATD prediction tasks from the comment data. This learning capability is crucial for achieving not only superior performance for the SATD prediction but also better generalizability and adaptability of the model for cross-project defect prediction. To understand the text features that our CNN learns, we developed a backtracking method to highlight the prominent key phrases in an input comment that contribute most to the decision whether the comment is a SATD or not. The highlighted key phrases provide an intuitive explanation of the CNN's prediction. They also reveal many less obvious, less frequent commenting patterns for SATD that are hard to identify just by human observation.

We then conducted an extensive set of experiments to evaluate the performance, generalizability, adaptability, and explainability of our approach. We used a dataset of source code comments from 10 open source projects containing a total of 259,229 comments, which is manually classified as SATD or non-SATD in Reference [11]. Our experiments show that (1) on average for

within-project prediction, our approach achieves an F1-score of 0.752, which improves Huang et al.'s text-mining approach [10] by 19.5%; (2) our CNN model is more generalizable than the text-mining approach for cross-project prediction, especially in a limited training datasetting; (3) our CNN model requires much less training data to adapt to a new unseen project, and the target model obtained through fine-tuning the source model can even outperform the model specifically trained with the target project comments, especially when the target project has limited training data; (4) the majority of our CNN-learned key phrases are relevant to SATD classification and align well with the human-annotated key phrases for SATD prediction; and (5) based on the CNN-learned key phrases, we can derive much more comprehensive SATD patterns than the 62 patterns identified in Reference [6]. These SATD patterns reveal the SATD characteristics of software projects and can qualitatively help us understand the performance, generalizability, and adaptability of our CNN-based approach. With the knowledge of the model's generalizability and adaptability, users would be able to deploy our approach more effectively depending on the SATD characteristics of software projects.

The main contributions of this article are as follows:

(1) We present a novel CNN-based approach to identify SATDs from source code comments, which is an imbalanced dataset. Our approach achieves a substantial improvement over text-mining approaches in both within- and cross-project settings;
(2) We have designed a backtracking method to extract and highlight key phrases and SATD patterns in the code comments, which can then be used to explain the SATD classification results by the CNN model;
(3) We have conducted extensive experiments to evaluate not only the performance of our approach but also its generalizability and adaptability, as well as the intuitiveness and explainability of the CNN-learned SATD features and patterns.

The rest of the article is organized as follows. In Section 2, we illustrate the challenges in SATD prediction by examples. We then describe the overall framework and technical details of our approach in Section 3. We present our experimental setup in Section 4 and report and analyse our experimental results in Section 5. We discuss the implications of our work as well as key threats to validity in Section 6. We review the related work in Section 7, and, finally, we conclude this article and discuss key directions for future work in Section 8.

2 MOTIVATION

Developers frequently remind themselves in source code via comments to make later changes, note unsatisfactory design decisions, question other developer's design decisions, and record work-in-progress to address major re-engineering or new engineering progress. Table 1 shows a range of examples from some representative, selected large-scale open source projects. These examples indicate different kinds of self-admitted technical debt (SATD). Recent research about SATD [7] has highlighted the need to analyse source code comments, locate instances of SATD, better understand their context, and address short- and long-term defect correction necessitated by the SATDs.

To classify a source code comments as SATD or non-SATD, an algorithm needs to determine the SATD-indicating features in the comments. However, as our examples in Table 1 show, SATD comments exhibit four characteristics: variant term frequency, project uniqueness, variable length, and semantic variation. These four characteristics pose a big threat to pattern-based SATD detection [6] or traditional text-mining-based SATD classification [10]. In addition, imbalanced data also requires special attention, i.e., the number of SATD comments is relatively low compared to other comments.

Table 1. Examples of SATD Comments from Four Projects

Project	SATD Comments
JEdit	// wtf?
	// Nasty hardcoded values
	// implement the recursion for getClassImpl()
Hibernate	// this perhaps not really necessary …
	// this is kinda the best we can do …
	// not absolutely necessary, but does help with aggressive release
EMF	// REVISIT: Remove this code.
	// Motif kludge: we would get something random instead of null.
	// EATM Demand create metadata; needs to depend on processing mode …
JRuby	// we might need to perform a DST correction
	// using IOChannel may not be the most performant way, but it's easy.
	// don't bother to check if final method, it won't be there (not generated, can't be!)

Key issues in SATD classification include the following:

(1) *Variant term frequency.* Developers often use terms like "todo," "hack," "fixme," and "temporary solution" to indicate SATDs. In addition to these frequently used SATD terms, there are many less evident SATD terms, such as "perhaps," "not necessary," "REVISIT," "remove," "might need to." Pattern-based SATD detection cannot enumerate all such less evident features, and feature selection process of text-mining method will very likely filter out such less frequently used features.

(2) *Project uniqueness.* Developers of different projects often have different conventions when describing SATDs. For example, SATD comments in JEdit tend to contain negative sentiment words (The so-called sentiment words are detected by a sentiment package of NLTK[1]) like "wtf" and "nasty," while developers of other projects may not have this tendency.

(3) *Variable length.* The length of SATD-indicating phrases in a source code comment can vary greatly, ranging from one word like "wtf," "REVISIT," to short phrases like "nasty hardcoded," "kinds the best," "not absolutely necessary," "don't bother to check," and to sentences like "may not be the most performant way, but it's easy." Traditional text-mining-based SATD classification cannot effectively model such variable-length text features.

(4) *Semantic variation.* Some semantically similar SATD comments may be expressed in different ways, such as "perhaps not really necessary" versus "not absolutely necessary," "kinda the best we can do" versus "may not be the most performant way." However, the same words or phrases may or may not indicate SATDs, depending on the overall comment context. For example, "implement" in "implement the recursion for" is considered an SATD feature, while "implement" in "we add an implement and implement_all methods to the class" is not. Depending on matching or modeling keywords lexically, pattern-based or traditional text-mining-based methods may miss semantically-similar-but-expressed-differently SATD comments or misclassify non-SATD comments as SATD comments.

(5) *Imbalanced data.* The ratio of SATD comments in a project is rather low compared with non-SATD comments in the project [7] (see also Table 3). This creates a classic imbalanced data issue for SATD classification. Huang et al.'s text-mining method [10] relies on feature

[1] http://www.nltk.org/api/nltk.sentiment.html.

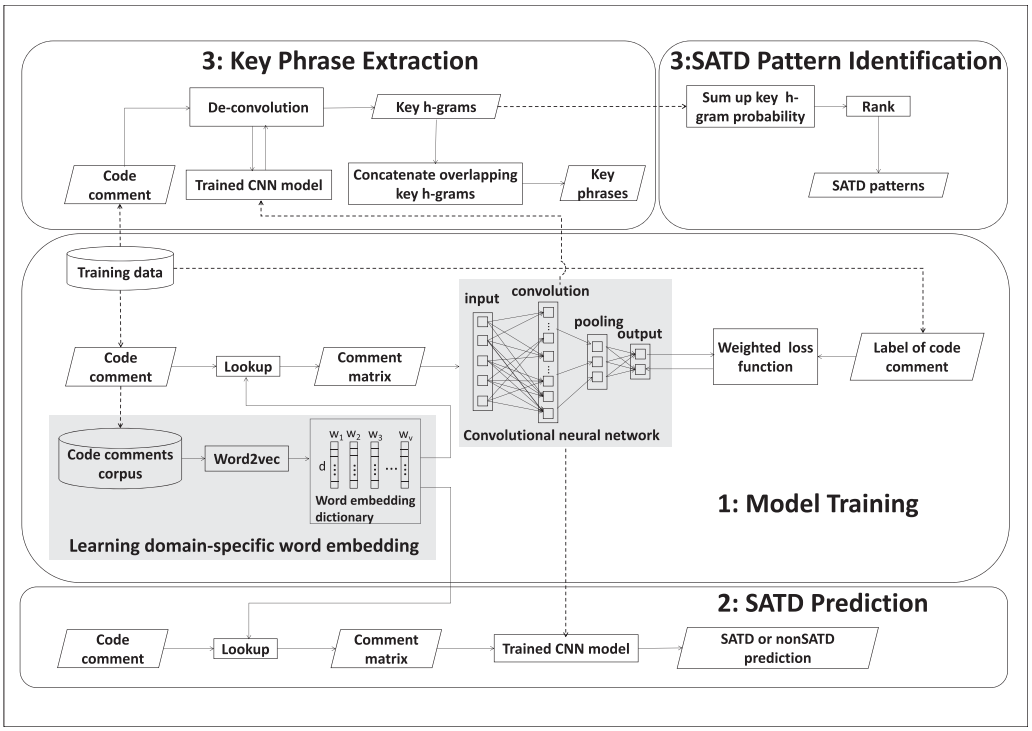

Fig. 1. The framework of our approach (filter size h = 2, 3, or 4).

selection to deal with imbalanced data, but it suffers the risk of filtering out many useful text features for SATD classification.

To address the above challenges, we propose a CNN-based approach for SATD classification. Our approach automatically learns diverse SATD features from input comments, thus removing the need of manual feature engineering in pattern-based or traditional text-mining-based methods (Challenge 1 and Challenge 2). It extracts and aggregates variable-length text features for SATD prediction (Challenge 3) and deals with semantic variations depending on the input word embeddings and the sentence-level feature embedding (Challenge 4). To overcome the imbalanced data issue, our approach implements a weighted cross-entropy loss (Challenge 5) rather than sacrificing the feature learning capability through downsampling or feature selection.

These features of our approach increases its generalizability and adaptability of the trained model for cross-project SATD prediction. Furthermore, our approach exploits the computational structure of CNN through backtracking to explain the CNN's prediction through the SATD features and patterns learned by our approach.

3 APPROACH

As shown in Figure 1, our approach includes four main phases: *Model Training*, *SATD Prediction*, *Key Phrase Extraction*, and *SATD Pattern Identification*. Predicting whether a source code comment is a SATD comment or not can be formulated as a binary text classification problem. The core of our approach is a CNN model for SATD classification. In the model training phase, the input is a set of source code comments and their corresponding SATD/non-SATD labels. These data are used to train the CNN model to learn to classify source code comments as SATDs or non-SATDs.

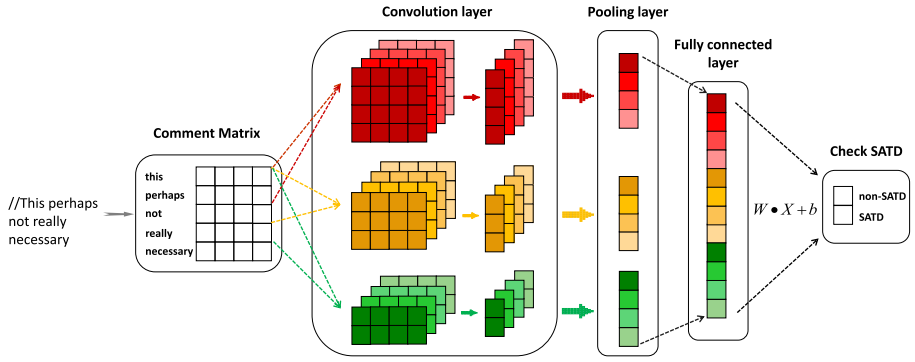

Fig. 2. Convolutional neural network architecture (four filters for each filter size ($h = 2, 3$, or 4)).

The trained model will then be used in two modes. First, the trained model is used to predict the SATD/non-SATD label given an unseen source code comment (prediction phase). Second, the trained model is de-convolutioned to extract SATD-indicating key phrases in the input code comments that contribute most to the model's classification decisions (key phrase extraction phase), which in turn are summarized into a set of intuitive SATD patterns (the SATD pattern identification phase).

3.1 SATD Classification by Convolutional Neural Network

We solve the problem of SATD classification using a CNN, which takes an input source code comment and predicts as output whether the comment is a SATD comment or not. In this subsection, we introduce the architecture of our CNN, the weighted loss function we designed to deal with the imbalanced data issue, the model training method, and the SATD prediction by the trained model.

3.1.1 The Architecture of CNN. Our CNN is inspired by the CNN architecture for sentence classification proposed by Kim [12], which is the a seminal work that applies CNN in sentence classification task and demonstrates the potentials of deep learning model in NLP. As shown in Figure 2, our CNN transforms an input code comment into a SATD or non-SATD classification through a stack of processing layers, including an input layer, a convolutional layer, a pooling layer, a fully connected layer, and, finally, an output layer.

Input layer: The raw input to the CNN is a source code comment in the form of a sequence of words. As words are discrete symbols, they need to be represented as vectors. In this work, we use word embeddings [13–15] as word representations, because word embeddings have been shown to be effective in capturing rich semantic and syntactic features of words and be robust in inferring semantic similarities of words. We learn domain-specific word embeddings using the continuous skip-gram model of word2vec [15] from the source code comments of some software projects. The output of word2vec is a dictionary of word embeddings for each word in the vocabulary V, denoted as $W \in \mathbb{R}^{d \times |V|}$, where d is the dimension of word embeddings and $|V|$ is the vocabulary size. d is a hyperparameter to be specified by users. A comment of n words is represented as $c = v_1 \bigoplus v_2 \bigoplus \cdots \bigoplus v_n$, which is used as the input (zero-padding at the beginning and the end to the maximum comment length) to the convolutional layer. v_i can be obtained by looking up the ith word's embedding in W and $\bigoplus$ is vector concatenation. That is, a comment is represented as a $n \times d$ matrix.

As an illustrative example, in Figure 2 the comment "this perhaps not really necessary" has five words ($n = 5$) and the dimension of word embeddings is four ($d = 4$).

Convolution layer: The convolutional layer treats the input comment matrix as an "image" and convests it into feature maps through convolution operations. A convolution operation applies a filter $w \in \mathbb{R}^{h \times d}$ and a bias term $b \in \mathbb{R}^h$ to h words in the input comment, followed by non-linear activation: $ReLU(w^T \cdot v_{i:i+h-1} + b)$. The filter w and the bias term b are parameters to be learned during model training. $ReLU(x) = max(0, x)$ is an non-linear activation function. The filter w is a matrix of $h \times d$ dimensions. As a row in the input comment matrix represents a word, it cannot be broken. So the width of the filter is equal to the dimension of word embeddings. To capture features of variable length, we vary the window size h of the filter, i.e., the number of adjacent words considered jointly. In Figure 2, we illustrate filters with three window sizes $h = 2$, 3, and 4, which can capture features of 2-, 3-, and 4-grams, respectively. With different size of filters, we can extract different h-gram features, so as to get variable length terms in the SATD comments. $v_{i:i+h-1}$ is the h-gram matrix, i.e., $v_i \bigoplus v_{i+1} \cdots \bigoplus v_{i+h-1}$, where $i = 1..n - h + 1$. $\cdot$ is the dot product between the filter matrix and the h-gram matrix. The filter is applied repeatedly to each possible h-gram in the input comment (i.e., $v_{1:h}, v_{2:h+1}, \ldots, v_{n-h+1:n}$), which produces a $(n - h + 1)$-dimensional feature map $f \in \mathbb{R}^{n-h+1}$. In Figure 2, we obtain a four-, three-, and two-dimensional feature map for a filter with the window size $h = 2$, 3, and 4, respectively. To learn complementary features from the same word windows, we use multiple filters of the same window size. In Figure 2, we use four filters for each filter window size. As such, we obtain four feature maps for each filter window size.

Pooling layer: As seen in Figure 2, the dimensions of the feature maps generated by the filters with different window sizes are different. A pooling operation should be applied to each feature map to induce a fixed-length vector. Pooling also helps reduce the number of features and decrease the phenomenon of overfitting. Max pooling is the most common pooling function in CNN architecture, which aims at extracting the most important n-gram feature compared with other n-gram features learned by a filter. In our work, we use 1-max pooling function that selects the maximum value of the feature map for each filter, and then concatenate the selected values for the same filter window size into a feature vector. In Figure 2, 1-max pooling produces three four-dimensional feature vectors, one for each filter window size.

Fully connected layer: Fully connected layer aggregates all features extracted by the convolutional and pooling layer. It essentially applies a convolution operation to all feature vectors produced by the pooling layer. In this work, we use a simple fully connected layer without learnable parameters that simply concatenates all feature vectors produced by the pooling layer. The output vector of this layer captures the sentence-level features of a SATD comment.

Output layer: The output layer makes classification based on the fully connected feature vector $X \in \mathbb{R}^{m \times 1}$, where m is the dimension of X. That is, the classification decision is based on all variable-length features in the overall context of the input comment. We use a linear classifier and the Softmax function to score each class. The line classifier performs a linear transformation of the feature vector X by $Y = W \cdot X + B$, where $W \in \mathbb{R}^{k \times m}$ and $B \in \mathbb{R}^k$ are learnable parameters for the classifier, and k is the number of classes. In this work, we have two classes: SATD or non-SATD, i.e., $k = 2$. Then, the softmax function is applied to normalize the values in Y so that each value represents the probability p_j of the input comment belonging to a class j. p_j is computed as $p_j = \exp(y_j) / \sum_{j=1..k} exp(y_j)$.

3.1.2 Weighted Loss Function. Assume an input comment belongs to the ith class. The ground-truth label of this comment is denoted as a vector $T \in \mathbb{R}^k$, in which the ith element value is 1 and all other element values are 0. We can use cross-entropy loss $Loss(Y, T) = -\sum_i T_i log(y_i)$ to compute the loss between a model's classification result Y and the ground-truth label T. However,

this cross-entropy loss does not work well in our SATD classification task, because SATD and non-SATD comments have a very imbalanced distribution [16]. As SATD comments are much fewer than non-SATD comments (e.g., the highest percentage of SATD comments in Table 3 is 5.57% in our experiment data), this cross-entropy loss may produce a model that is very likely to misclassify a SATD comment as non-SATD.

In our work, to deal with the class imbalance issue in code comments, we propose a weighted cross-entropy loss as the loss function of our CNN. Note that our weighted loss function is not used in the sentence classification model by Kim [12], as it does not explicitly deal with imbalanced data. In our weighted loss function, we set a weight for each class. The weight is estimated by the proportion of a class's data in the training data. For example, assuming there are n SATD comments and m non-SATD comments in the training data. Then, we obtain the weighs $n/(n+m)$ for SATD and $m/(n+m)$ for non-SATD. We add these weights to the cross-entropy loss function: $Loss(Y,T) = -(1 - weight_i) \sum_i T_i log(y_i)$. Our weighted loss function penalizes more the wrong classification of the small-class instances (e.g., SATD in our work) than large-class instances.

3.1.3 Model Training. The training process of a CNN is an optimization task that proceeds in multiple iterations. In each iteration, the CNN predicts the labels of comments in the training data and measures the loss between the predicted label and the ground-truth label of each comment. Then, the CNN uses back-propagation to adjust the convolutional layer and linear classifier parameters. In the subsequent iterations, it tries to lower the average loss on the training data. We solve the optimization problem using the Adam update algorithm [17].

3.1.4 SATD Prediction. Once we learn the CNN parameters, the trained CNN model can be used to predict the SATD or non-SATD class of an unseen comment. Given an input comment, it first converts it into an input comment matrix by looking up the word embedding dictionary, and then feeds the matrix through the convolutional layer, pooling layer, fully connected layer. The output layer finally gives the SATD and non-SATD probability for the input comment. The class with the higher probability is assigned as the label of the input comment.

3.2 SATD Key Phrase Extraction by De-convolution

In this work, we do not want to blindly trust the classification results of the CNN model. Instead, we want to understand and explain the basis of the CNN model's decisions. This explainability is a novel contribution of our work compared with Kim [12]. As seen in Figure 2, our CNN model performs convolution and 1-max pooling operations to extract a feature vector from the input comment, based on which it classifies the comment. During the convolutional and max pooling, the CNN reserves the spatial structure of the input comment matrix. As illustrated in Figure 3, a feature value in the the feature vector in the fully connected layer can be traced back to the window of words from which the feature value is derived.

We explore this input↔output traceability to extract the key phrases in the input comment that contribute most to the SATD classification. To extract such key phrases, we perform the following steps to answer the two questions: *where*, the location of key phrases in the input comment, and *what*, the predicted label of each key phrase (SATD or non-SATD):

(1) We feed a comment into a trained CNN model to obtain the feature vector $X \in \mathbb{R}^{m \times 1}$ of this comment in the fully connected layer. According to our CNN architecture, each feature x_i $(1 \le i \le m)$ in the feature vector X corresponds to a filter.
(2) For each feature x_i $(1 \le i \le m)$ in the feature vector X, we use the softmax function to determine the probability of the feature x_i belonging to a class j: $p(j|x_i) = exp(w_{ji}x_i)/\sum_{l=1..k} exp(w_{li}x_i)$, where k is the number of the classes to predict ($k = 2$ in

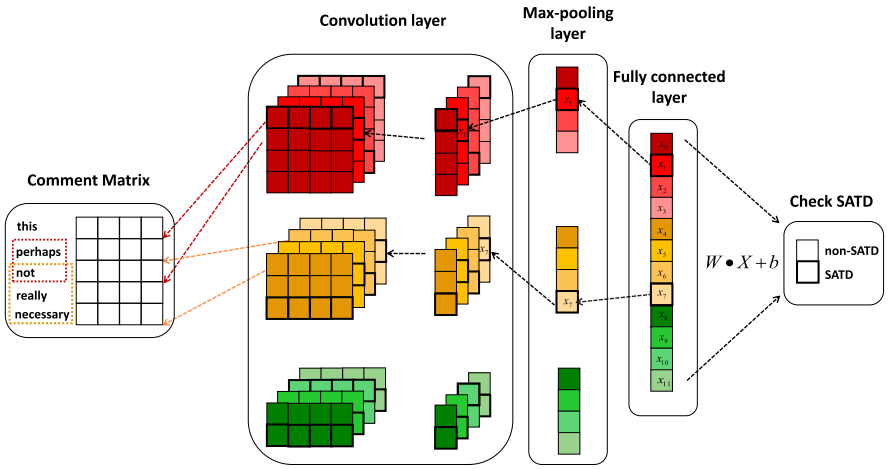

Fig. 3. De-convolution to identify key h-grams (filter size h = 2, 3, or 4).

this work) and w_{1i} is the parameter of the W matrix of the liner classifier. If the $p(SATD|x_i)$ is above 0.5, then the feature x_i is regarded as a prominent feature for the SATD class. In Figure 3, assume that two feature x_1 and x_7 are prominent features for the SATD class.

(3) For each prominent feature x_i, its corresponding filter extracts a key h-gram for the SATD class. We use backtracking to locate the key h-gram in the input comment. The probability $p(SATD|x_i)$ of the feature x_i is assigned to this h-gram as its probability of the SATD class. For example, the feature x_1 is traced back to the 2-gram "perhaps not," and the feature x_7 is traced back to the 3-gram "not really necessary." Note that different prominent features may be traced back to the same key h-gram. This is because different filters may all "feel" that this h-gram is important for the SATD class. In such cases, we average the probability $p(SATD|x_i)$ of these prominent features as the probability of the key h-gram.

(4) We merge the overlapping key h-grams into a key phrase based on which the CNN make a SATD classification. In Figure 3, the 2-gram "perhaps not" and the 3-gram "not really necessary" is merged as a key phrase "perhaps not really necessary" for the SATD class.

3.3 SATD Pattern Identification

Different comments within and across projects may share the same or similar key h-grams when developers describe SATDs. Such key h-grams can be summarized as SATD patterns that can help identify and understand the prominent SATD indicators in source code comments that developers use to describe SATDs in different software projects. Previous work [6] manually summarizes 62 such SATD patterns. In our work, we developed an automated method to summarize SATD patterns based on the SATD-probability and occurrence frequency of the key h-grams that the CNN extracts from the SATD comments. Note that we identify SATD patterns based on key h-grams rather than key phrases, because key h-grams are more elementary SATD indicators in the comments, while key phrases are mainly for human understanding of a specific classification result by the CNN.

To take into account both the SATD probability and occurrence frequency of the key h-grams, we add the SATD probability of all occurrences of a key h-gram in the training data and sort them according to their sum of SATD probabilities. The higher the SATD probability of a key h-gram, the more frequently this h-gram appears in the comments, the more likely it is a SATD pattern that developers commonly use. As we have filters with different window sizes, we are

able to identify SATD patterns of variable lengths. In our experiments, we observe that a SATD pattern is usually meaningful as a whole phrase, but parts of a SATD pattern may not always be meaningful SATD indicators. For example, "perhaps not" is a 2-gram SATD pattern and "not really necessary" is a 3-gram SATD pattern. However, the unigrams "perhaps" and "really" are not meaningful SATD indicators. "not" and " necessary" may or may not be meaningful SATD indicator depending on the surrounding context. For example, "not" in "do nothing if item is not visible" does not describe a SATD. In many NLP tasks, such unigrams are removed as stop words. However, we cannot remove them in our work, because they may constitute a SATD pattern, e.g., "perhaps not" and "not really necessary." Instead, we rely on our CNN model's sentence-level feature embedding to distinguish h-gram semantics in the input comments and the relations of these h-grams with SATDs. As such our approach can deal with the challenge of semantic variation. Additionally, due to the architecture of CNN, we can make full use of all features that are extracted by the model to identify SATD comments. We can also backtrack the CNN features to key phrases in the input comments to extract diverse vocabulary, and even project unique terms in SATD comments. As such, our approach can identify SATD patterns similar to those identified by human [6], which are much more intuitive and meaningful than those identified by the traditional text-mining approach [10].

In this study, we use Python package **TensorFlow** to implement our CNN-based approach.

4 EXPERIMENT DESIGN

In this section, we describe the research questions we investigate in our experiments, the two baseline methods, the experimental dataset, the evaluation metrics, and the experimental environment.

4.1 Research Questions

First, we developed three research questions (RQ1, RQ2, RQ3, and RQ4) to investigate different aspects of our approach. They help establish the confidence in our approach and the quality of the following performance experiments. Furthermore, the observed SATD patterns also help explain the results of performance comparative studies.

- **RQ1 Effectiveness of weighted loss for model training:** How effective is our weighted loss for imbalanced comment data compared with normal cross-entropy loss?
- **RQ2 Effectiveness of hyperparameters optimization:** How do different hyperparameters perform in our approach?
- **RQ3 Feature learning capability by CNN:** How effective is our CNN model in learning comment features for SATD classification?
- **RQ4 SATD patterns and their explanability:** How do our CNN-based SATD patterns compare with human-summarized SATD patterns [6]? Can CNN-based SATD patterns reveal the SATD characteristics of different projects?

Second, we have three research questions (RQ5, RQ6, and RQ7) to investigate the performance of our approach within and across projects compared with traditional text-mining-based SATD classification [10]. These comparative studies help identify the advantages of our approach over traditional text-mining-based methods in performance, generalizability, and adaptability.

- **RQ5 Within-project classification performance:** How effective is our approach for classifying SATDs in different projects that may contain different SATD characteristics?
- **RQ6 Cross-project model generalizability:** How generalizable is our CNN-based SATD classification model when trained using data taken from one or many other projects?

- **RQ7 Cross-project model adaptability:** Can transfer learning help address the training challenge of our CNN-based model in limited training datasetting? How adaptable is our CNN-based SATD classification model with limited fine-tuning data?

Finally, we conduct a user study to further confirm the prediction performance and explainability of our approach for SATD classification.

- **RQ8 User study of model performance and explainability:** How well do our model's SATD classification results align with human classification? How well do CNN-extracted key phrases align with the human-identified key phrases for SATD classification?

4.2 Baseline Methods

We have three baseline methods that are used in the comparative study of different RQs: our approach with normal cross-entropy loss function in RQ1, Kim's simple CNN text classification method [12] in RQ2, pattern-based SATD extraction [6] in RQ4, and traditional text-mining-based SATD classification [10] in RQ5, RQ6, and RQ7, as well as natural language processing-based SATD classification [11] in RQ5 and RQ6.

4.2.1 Our Approach with Normal Loss Function. As SATD comments are imbalanced data, we design a weighted cross-entropy loss function to deal with the imbalanced data. In the RQ1, we use our approach with normal cross-entropy loss function as the baseline to evaluate the effectiveness of this proposed weighted loss function.

4.2.2 Kim's Simple CNN for Sentence Classification. Kim [12] proposes a simple CNN (i.e., a simple CNN with one layer of convolution performs) trained on top of Mikolov's word embeddings [18], and then apply the CNN to sentence classification. Compared with our work, Kim's work does not consider the imbalance of dataset and cannot do hyperparameter optimization. In the RQ2, we optimize hyperparameters of CNN model and compared results with Kim's approach.

4.2.3 Pattern-based SATD Extraction. After reading many comments relating to technical debt, Potdar and Shihab manually summarized 62 patterns for identifying SATD [6]. If a comment contains at least one of the 62 patterns by string matching, then it will be regarded as an SATD and otherwise as non-SATD. In the RQ4, we use these 62 human-summarized patterns as the baseline to understand the characteristics and intuitiveness of the SATD patterns learned by our CNN model.

4.2.4 Traditional Text-Mining Based SATD Classification. Huang et al. [10] proposed a text-mining approach to identify SATD comments. Their approach utilizes feature selection technique, namely Information Gain (IG) [19] to select useful features for SATD classification. Then, they use the selected features to train a sub-classifier for each source project, and use a majority-voting strategy to combine the prediction of multiple classifiers. In the RQ5, RQ6, and RQ7, we use the text-mining approach in Reference [10] as the baseline to compare to the performance, generalizability, and adaptability of our approach. Through a series of experiments, Huang et al. [10] demonstrated that their text-mining approach outperforms several baseline methods including Naive Bayes Multinomial [20], Support Vector Machine [21], and k-Nearest Neighbor [22]. Therefore, we do not include these baseline methods. Additionally, we do not consider the approach proposed by Maldonado et al. [11] as a baseline, because Huang et al. [10] has been already compared against this work and the approach of Huang et al. [10] outperforms that of Maldonado et al. [11].

Table 2. Summary of the 10 Projects in Our Dataset

Project	Description	Release	Contributors
Apache Ant	Java library and Command-line Tool	1.7.0	74
ArgoUML	UML Modeling Tool	0.34	87
Columba	E-mail Client	1.4	9
EMF	Eclipse Model Driven Architecture	2.4.1	30
Hibernate	ORM Framework	3.3.2	226
JEdit	Text Editor	4.2	57
JFreeChart	Char library	1.0.19	19
JMeter	Performance Tester	2.10	33
JRuby	Ruby Interpreter	1.4.0	328
SQuirrel	SQL Client	3.0.3	46

Table 3. Statistics of the Comments in the 10 Projects

Project	Comments	# of comments after filtering	SATD	% of SATD
Apache Ant	21,587	4,137	131	0.60
ArgoUML	67,716	9,548	1,413	2.08
Columba	33,895	6,478	204	0.60
EMF	25,229	4,401	104	0.41
Hibernate	11,630	2,968	472	4.05
JEdit	16,991	10,322	256	1.50
JFreeChart	23,474	4,423	209	0.89
JMeter	20,084	8,162	374	1.86
JRuby	11,149	4,897	662	5.57
SQuirrel	27,474	7,230	285	1.04
Average	25,923	6,257	411	1.86
Total	259,229	62,566	4,110	-

4.3 Data Collection

To perform our experiments, we use the dataset provided in Reference [11]. This dataset has the classified comments from 10 open source Java projects, incuding Apache Ant, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby, and SQuirrel. Table 2 summarizes the basic information of these 10 projects. These 10 projects belong to different application domains and vary in the number of contributors and in size and complexity. We directly use the labelled code comment data from the dataset provided by Maldonado et al. [11]. The data processing process by Maldonado et al. [11] is as follows: After getting source codes of the 10 projects, Maldonado et al. use JDeodorant [23], which is an Eclipse plug-in that identifies design problems in software, to parse the source code and extract the comments and the related information, including the line that each comment starts, finishes and the type of comment (i.e., Javadoc, Line, or Block).

Table 3 lists the statistics of the comments in these 10 projects. We can see that these projects also vary in the number of comments. As only a small ratio of the source code comments describe SATDs, it will be time-consuming to label all comments manually. Thus, Maldonado and Shihab develop five filtering heuristics to eliminate comments that are unlikely to be classified as SATDs [24]. By applying these filtering heuristics and also removing duplicate comments (i.e., if

the text contents of multiple comments are the same, then only one comment is kept), the number of comments that require manual annotation is largely reduced from 259,229 to 62,566.

Then, Maldonado and Shihab manually examine each comment and classify it as SATD or non-SATD. To mitigate personal bias, they took a stratified sample of the full dataset, which is a sample that achieves a confidence level of 99% and a confidence interval of 5%. They invited another independent person to classify the stratified sample of the comments and measured the level of agreement between the two manual classification results. They reported a high level of agreement (Cohen's Kappa coefficient [25] of 0.81), which gives us a high confidence in the SATD classification results of the provided dataset.

As seen in Table 3, the percentage of SATD comments in each project is quite low (i.e., the percentage ranges between 0.41% and 5.57%, with an average of 1.86%). That is, we have a class imbalance problem [26] for SATD classification.

4.4 Evaluation Metrics

We define four statistics: FP (false positive) represents the number of non-SATD comments that are classified as SATD; FN (false negative) represents the number of SATD comments that are classified as non-SATD; TP (true positive) represents the number of SATD comment that are classified as SATD; and TN (true negative) represents the number of non-SATD comments that are classified as non-SATD. Using these four statistics, we compute Precision, Recall, and F1-score to evaluate the performance of a SATD classification method.

Precision. Precision represents the proportion of comments that are correctly classified as SATD comments among all comments classified as SATD.

$$precision = \frac{TP}{TP + FP}$$

Recall. Recall represents the proportion of all SATD comments that are correctly classified as SATD.

$$recall = \frac{TP}{TP + FN}$$

F1-score. F1-score is the harmonic mean of precision and recall, which can combine both of the two metrics above. It evaluates if an increase in precision (or recall) outweighs a reduction in recall (or precision), respectively

$$F1 = \frac{2 \times precision \times recall}{precision + recall}$$

The higher an evaluation metric, the better a method performs. Note that there is a tradeoff between precision and recall. F1-score provides a balanced view of precision and recall. In this study, we focus more on identifying SATD comments, hence we use precision, recall and F1-score for SATD comments as our evaluation metrics.

In our experiments, we randomly shuffle the experimental dataset into 10 parts: Nine of them are taken as training data in turns and the rest one is used as testing data. The evaluation metrics are obtained in each test. Then, we compute values of the evaluation metrics across the 10 runs to estimate the performance of a SATD classification method.

4.5 Model Configuration

Training and deploying a CNN model requires the proper setting of its hyperparameters, including word embedding dimension, the number of filters, and the window size of filters. Previous studies

(e.g., References [15, 27]) have shown that hyperparameter optimization is important for a prediction model to achieve better performance. In this work, we follow the guideline and the procedure proposed by Zhang and Wallace [28] to set or optimize the hyperparameters of our CNN model.

To determine appropriate model hyperparameters, we randomly sampled 10% of the after-filtering comments (whose statistics are given in the third column of Table 3) as the training data to train our model. We use 10% data, because they approximate the average number of comments in the 10 projects. We use the same learning rate as that in Kim's CNN model training (i.e., 10^{-4}). We use another 1% of randomly sampled after-filtering comments as the validation data to optimize model hyperparameters, including word embedding dimensions, the window size of filters, and the number of filters. In our optimization experiments, we randomly shuffle the validation dataset into 10 parts: Nine of them are taken as training data in turns and the remaining 1 part is used as testing data. We repeat the cross-validation for 10 times and then compute the mean values of the evaluation metrics across the 10 experiments to estimate the performance of the model.

In practice, it is impossible to enumerate the full search space of the combination of all hyperparameters. Therefore, when tuning hyperparameters, we first search and select the values of the first two hyperparameters in a greedy way. Specifically, we first configure the dimension of word embedding at 100, 200, and 300. We cap the experimental word embedding dimension at 300, because Zhang and Wallace [28] uses 300-dimensional word vectors in their experiment and 300 dimensions is also the default setting of pre-trained Google word2vec and GloVe word vectors. We collect all comments of the 10 projects in a text corpus. We learn word embeddings using the continuous skip-gram model of word2vec [15]. We then train a CNN with the pre-trained word vectors for each candidate word embedding dimension. Note that during the test process, all the other hyperparameters are set to their default values (i.e., the same values as those appearing in the published source code of Kim's CNN implementation based on TensorFlow[2]). Among all the candidate word embedding dimensions, we choose the one whose corresponding CNN achieves the minimum average loss on the training dataset.

Similarly, we then configure the number of filters from 64 to 256 with a step of 64, and also choose the one whose corresponding CNN achieves the minimum average loss. We choose the range 64 to 256 for the number of filters, because Zhang and Wallace [28] show that the model with 50 or below filters yields poor performance and the model with 300 or above filters has much more parameters to learn but yields only very marginal performance improvement. Note that capping word embedding dimensions and the number of filters below 300 also makes our experiments practical for the GPU resources we have. In addition, we consider 10 combinations of filter window sizes, including (1,2,3), (2,3,4), (3,4,5), (4,5,6), (1,2,3,4), (2,3,4,5), (3,4,5,6), (1,2,3,4,5), (2,3,4,5,6), and (1,2,3,4,5,6). We consider these six window sizes, because the length of human observed SATD patterns [6] falls into the range of 1 to 6. Furthermore, Zhang and Wallace's experiments suggest that the window size above 7 may lead to performance degradation in sentence classification tasks. After hyperparameter tuning (see the experiment results reported in Section 5.2), we set word embedding dimension at 300, use filters of six different window sizes (1, 2, 3, 4, 5, 6) (i.e., 1- to 6-grams), and use 128 filters for each window size. This setting achieves the best F1-score on the validation data. We use this setting for all the experiments in the eight RQs.

4.6 Experimental Environment

The experimental environment is a desktop computer equipped with Nvidia GTX 1080 GPU, Intel(R) Core(TM) i7-6700 CPU and 16GB RAM, running Ubuntu 16.04 LTS.

[2]https://github.com/dennybritz/cnn-text-classification-t.

Table 4. Precision, Recall and F1-score of Our Approach with Weighted Loss (WL) Versus Normal Loss (NL)

Projects	Precision			Recall			F1-score		
	NL	WL	Improv.	NL	WL	Improv.	NL	WL	Improv.
Apache Ant	0.475	0.584	22.95%	0.692	0.758	9.54%	0.563	0.660	*17.16%*
ArgoUML	0.813	0.816	0.37%	0.913	0.950	4.05%	**0.860**	**0.878**	2.08%
Columba	0.793	0.830	4.67%	0.793	0.875	10.34%	0.793	0.852	*7.44%*
EMF	0.718	0.793	10.45%	0.534	0.594	11.24%	0.612	0.679	*10.86%*
Hibernate	0.899	0.930	3.45%	0.726	0.743	2.34%	0.803	0.826	2.83%
JEdit	0.766	0.773	0.91%	0.457	0.489	7.00%	0.572	0.599	4.64%
JFreeChart	0.652	0.686	5.21%	0.734	0.802	9.26%	0.691	0.739	*7.01%*
JMeter	0.801	0.873	8.99%	0.786	0.787	0.13%	0.793	0.828	4.36%
JRuby	0.783	0.805	2.81%	0.884	0.930	5.20%	0.830	0.863	3.92%
SQuirrel	0.770	0.794	3.12%	0.685	0.692	1.02%	0.725	0.739	1.93%
Average	0.747	0.788	6.29%	0.720	0.762	6.01%	0.724	0.766	6.22%

The best F1-scores of each method are in bold and the worst ones are underlined.

5 EXPERIMENTAL RESULTS

In this section, we report and analyze our experiment results in the seven RQs.

5.1 RQ1: Effectiveness of Weighted Loss for Model Training

5.1.1 Motivation. To train a CNN model, an appropriate loss function needs to be used. To deal with the imbalanced SATD/non-SATD comments for SATD classification, we extend normal cross-entropy loss into weighted cross-entropy loss (see Section 3.1.2). We compare the effectiveness of normal loss versus weighted loss for model training.

5.1.2 Approach. We train two CNN models for SATD classification: one with normal loss and the other with weighted loss. Other than the use of different loss functions, the two models use the same model configuration (see Section 4.5). We use nine projects' comments as the training data and the other one project comments as the testing data. Therefore, we have 10 experiments for the model trained with normal loss and with weighted loss, respectively. For each experiment, we compute precision, recall, and F1-score.

5.1.3 Results. Table 4 show the the results of the 20 experiments. We can see that the models trained with weighted loss always outperforms the models trained with normal loss. On average, the weighted loss achieves 6.29%, 6.01%, and 6.22% improvement in precision, recall, and F1-score than the normal loss, respectively. The F1-score improvement is relatively bigger for the four projects (i.e., Apache Ant, Columba, EMF, and JFreeChart) compared with the other six projects. Among the 10 projects, Apache Ant, Columba, EMF, and JFreeChart have the least number of SATD comments (around 200 or less), and the percentage of SATD comments in these four projects is very low (less than 0.9%).

Our weighted loss function produces a better CNN model than normal loss when dealing with imbalanced SATD/non-SATD comments. *The more the data are imbalanced, the more effective our weighted loss function is for model training.*

5.2 RQ2: Effectiveness of Hyperparameters Optimization

5.2.1 Motivation. Kim [12] proposed a simple one-layer CNN that achieved state-of-the-art results for sentence classification across We utilize Kim's approach to make SATD classification as

Table 5. Cross-Project Precision, Recall, and F1-score of Traditional Text-mining Approach (TM) and Simple CNN Approach (Sim.)

Projects	Precision			Recall			F1-score		
	TM	Sim.	Improv.	TM	Sim.	Improv.	TM	Sim.	Improv.
Apache Ant	0.565	0.433	−23.36%	0.471	0.508	7.86%	0.513	0.468	−8.87%
ArgoUML	0.815	0.805	−1.23%	**0.856**	**0.905**	5.72%	**0.835**	**0.852**	2.05%
Columba	0.817	0.789	−3.43%	0.805	0.775	−3.37%	0.811	0.782	−3.58%
EMF	0.636	0.694	9.12%	0.473	0.428	−9.51%	0.543	0.529	−2.49%
Hibernate	**0.879**	**0.907**	3.19%	0.729	0.747	2.47%	0.796	0.819	2.92%
JEdit	0.772	0.720	−6.74%	0.364	0.323	−11.26%	0.495	0.446	−9.91%
JFreeChart	0.627	0.617	−1.59%	0.733	0.725	−1.09%	0.676	0.667	−1.38%
JMeter	0.858	0.811	−5.48%	0.770	0.754	−2.08%	0.881	0.781	−3.64%
JRuby	0.855	0.803	−6.08%	0.752	0.885	17.69%	0.800	0.842	5.25%
SQuirrel	0.784	0.750	−4.34%	0.597	0.688	15.24%	0.678	0.718	5.85%
Average	0.761	0.733	−3.67%	0.655	0.674	2.87%	0.696	0.690	−0.78%

The best results of each method are in bold and the worst ones are underlined.

Table 6. Average Results and Standard Deviation (σ) of Different Word Vector Dim

Dimensionality	Ave_precision (σ)	Ave_recall (σ)	Ave_F1-score (σ)
100-dim	0.533 (0.057)	0.771 (0.066)	0.630 (0.055)
200-dim	0.676 (0.046)	0.716 (0.081)	0.696 (0.069)
300-dim	0.705 (0.053)	0.734 (0.058)	**0.719** (0.054)

The best result of F1-score is in bold.

well. In Table 5, we can find that Kim's approach obtains worse results in some projects compared with Huang et al.'s [10] text-mining approach. To take advantage of the CNN method, we refer to the work of Zhang and Wallace [28] to optimize the model configuration. In that work, they conducted a sensitivity analysis of (and wrote a practitioners' guide to) CNN for sentence classification. They found that the dimension size of word embedding, the number of filters, and the combination of filter size are the most important hyperparameters that affect the model performance and thus should be optimized for new tasks. Therefore, we need to conduct hyperparameter optimization experiments to find the most appropriate combination of hyperparameters for our SATD classification task.

5.2.2 Approach. As the hyperparameters of a CNN model can affect the model performance [28], this section evaluates the model performance when it uses different word embedding dimensions, filter window sizes, or number of filters. When experimenting one of these three hyperparameters, we take a method called Control Variable, which refers to making other variables constant when evaluating one of the variables. In particular, we use as default hyperparameters in Kim's CNN architecture for sentence classification [12].

5.2.3 Results. The rest of this section will make analyses of experiments results with different hyperparameters.

For word embedding dimensions, we chose three appropriate dimension sizes: 100, 200, and 300. With each dimension size, we first use code comments from nine different JAVA project to train a CNN model. Then, we predict comments in the rest project with the pre-trained model, so as to calculate precision, recall, and F1-score, as well as σ for each average result. Table 6 shows the results. It can be seen that the larger the word embedding dimension, the better performance the

Table 7. Average Results and Standard Deviation (σ) with Different Combinations of Filter Window Sizes

Combination of filter window sizes	Ave-precision (σ)	Ave-recall (σ)	Ave-F1-score (σ)
(1,2,3)	0.696 (0.023)	0.648 (0.019)	0.671 (0.022)
(2,3,4)	0.713 (0.037)	0.687 (0.042)	0.700 (0.035)
(3,4,5)	0.758 (0.029)	0.711 (0.021)	0.734 (0.022)
(4,5,6)	0.744 (0.033)	0.672 (0.023)	0.706 (0.026)
(1,2,3,4)	0.734 (0.030)	0.701 (0.027)	0.717 (0.025)
(2,3,4,5)	0.776 (0.048)	0.742 (0.036)	0.758 (0.039)
(3,4,5,6)	0.740 (0.041)	0.767 (0.038)	0.753 (0.040)
(1,2,3,4,5)	0.732 (0.045)	0.756 (0.053)	0.744 (0.047)
(2,3,4,5,6)	0.765 (0.038)	0.732 (0.032)	0.748 (0.037)
(1,2,3,4,5,6)	0.778 (0.059)	0.759 (0.043)	**0.768** (0.044)

The best result of F1-score is in bold.

Table 8. Average Results and Standard Deviation (σ) with Different Number of Filters

The number of filters	Ave-precision (σ)	Ave-recall (σ)	Ave-F1-score (σ)
64	0.723 (0.056)	0.687 (0.047)	0.705 (0.052)
128	0.715 (0.034)	0.764 (0.020)	0.739 (0.028)
192	0.727 (0.043)	0.759 (0.038)	0.743 (0.039)
256	0.730 (0.046)	0.762 (0.051)	**0.745** (0.049)

The best result of F1-score is in bold.

model achieves. Additionally, from the values of σ, we can see that results of all dimension sizes are not very volatile. In this work, we choose 300 as word embedding dimension, which is also widely used setting in literature [15, 29].

For the combination of filter window size, we considered 10 combinations, including (1,2,3), (2,3,4), (3,4,5), (4,5,6), (1,2,3,4), (2,3,4,5), (3,4,5,6), (1,2,3,4,5), (2,3,4,5,6), and (1,2,3,4,5,6). These combinations are produced by an enumeration from 1-gram to 6-gram with at least three different window sizes for one combination. Although there are still other combinations, it is impractical to enumerate all the combinations. Table 7 shows the average results of each combination. From the values of σ, the results of all combinations are not volatile and it is obvious that the combination (1,2,3,4,5,6) performs the best. Some other combinations also have good results, such as (2,3,4,5) and (3,4,5,6) (which is similar to the results reported in Zhang and Wallace [28]). In our SATD classification task, filter window size refers to the size of potential SATD key phrases. To obtain a complete range of of SATD patterns, we choose (1,2,3,4,5,6) as the combination of filter window size in this work.

For the number of filters, we chose four numbers (namely 64, 128, 192, and 256) to do experiments. According to Table 8, it can be seen that the more filters the model uses, the higher precision, recall, and F1-score it can achieve and the values of σ are quite small. Since each filter learns complementary features from the same windows in the input text, more filters means more features can be learned. However, increasing the number of filters above 128 results in only a slight performance improvement. As the more filters a model uses, the more parameters it has. To balance the model performance and its complexity, we set 128 filters of each filter window size.

After hyperparameter tuning, the CNN model with the optimized hyperparameters (with normal loss function) can achieve better results that the model using the hyperparameter settings in Kim [12], which is shown in Table 9. It can be seen that hyperparameter tuning can improve the

Table 9. Cross-Project Precision, Recall, and F1-score of Our Approach with Normal Loss Function (Our-NL) and Simple CNN Approach (Sim.)

Projects	Precision			Recall			F1-score		
	Sim.	Our-NL	Improv.	Sim.	Our-NL	Improv.	Sim.	Our-NL	Improv.
Apache Ant	0.433	0.475	9.70%	0.508	0.692	36.22%	0.468	0.563	20.42%
ArgoUML	0.805	0.813	0.99%	**0.905**	**0.913**	0.88%	**0.852**	**0.860**	0.93%
Columba	0.789	0.793	0.51%	0.775	0.793	2.32%	0.782	0.793	1.41%
EMF	0.694	0.718	3.46%	0.428	0.534	24.77%	0.529	0.612	15.59%
Hibernate	**0.907**	**0.899**	−0.88%	0.747	0.726	−2.81%	0.819	0.803	−1.98%
JEdit	0.720	0.766	6.39%	0.323	0.457	41.49%	0.446	0.572	28.27%
JFreeChart	0.617	0.652	5.67%	0.725	0.734	1.24%	0.667	0.691	3.65%
JMeter	0.811	0.801	−1.23%	0.754	0.786	4.24%	0.781	0.793	1.48%
JRuby	0.803	0.783	−2.49%	0.885	0.884	−0.11%	0.842	0.830	−1.43%
SQuirrel	0.750	0.770	2.67%	0.688	0.685	0-0.44%	0.718	0.725	1.02%
Average	0.733	0.747	1.92%	0.674	0.720	6.92%	0.690	0.724	4.90%

The best results of each method are in bold and the worst ones are underlined.

average F1-score by 4.9%. Only the F1-scores of Hibernate and JRuby achieved by the optimized model are slightly lower than those of the default model.

Our hyperparameter tuning approach can obtained better performance than simple CNN approach. Additionally, the dimension size of word embedding, the number of filters, and the combination of filter window size are the most important hyperparameters.

5.3 RQ3: Feature Learning Capability by CNN

5.3.1 Motivation. In contrast to manual enumeration of SATD patterns [6] or a separate feature selection step in the text-mining approach [10], our CNN model automatically learns variable-length filters to extract features in the input comments that are most relevant to SATD classification (see Section 3.1.1). We investigate the feature learning capability of our CNN model by analyzing key phrases extracted by backtracking, in particular, whether our CNN can extract meaningful key phrases from code comments.

5.3.2 Approach. We train our model with the 10 projects' comments. Then, we randomly sample 20 SATD comments for each project from the manually labeled SATD comments of these 10 projects. Through the key phrase extraction mechanism (see Section 3.2), we can backtrack the CNN features to key phrases in the input comments. We visualize the extracted key phrases and analyze their properties (e.g., length and SATD probability). Furthermore, two of the authors independently labeled the extracted key phrases as relevant or irrelevant to the SATD classification of the corresponding comments. We use Cohen's Kappa [25] to assess the inter-rater agreement.

5.3.3 Results. Figures 4, 5, and 6 shows the statistics of the sampled comments and the extracted key phrases. We can see that the length of sampled comments vary: 73.49% has fewer than 10 words, 21.69% has 10–30 words, 2.74% has 30–50 words, and 2.08% has more than 50 words. Our CNN model extracts in total 3,540 key phrases from the sampled comments. Most of the comments contain four, five, or six key phrases, but some contain fewer than four key phrases or more than six key phrases. The length of the extracted key phrases also vary: 29.60% has one word, 22.71% has 2 words, 19.58% has 3 words, 14.83% has 4 words, and 8.89% and 4.32% has 5 and 6 words. Tables 10, 11, 12, and 13 show four examples of the comments we examine, including the sampled comment,

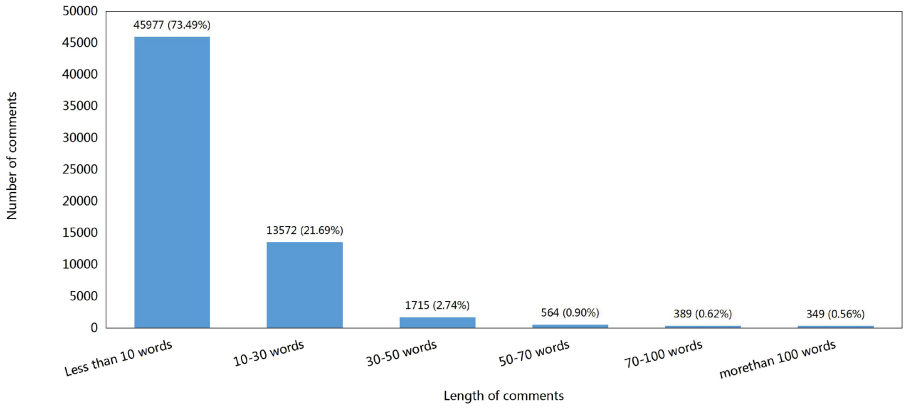

Fig. 4. Distribution of comment length.

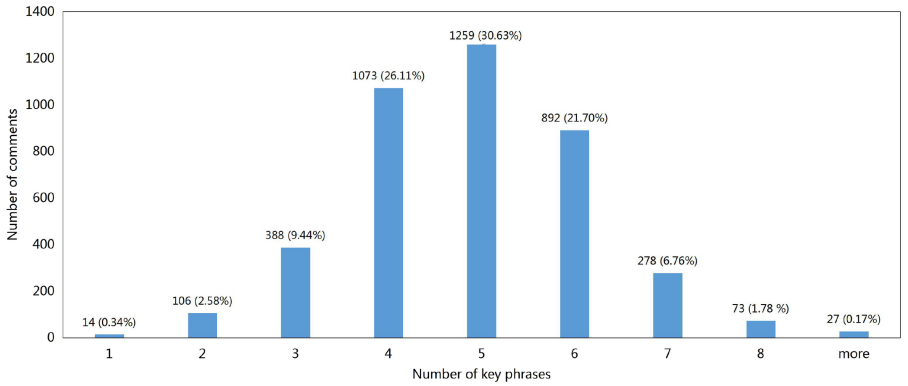

Fig. 5. Distribution of number of key phrases per comment.

the key *h*-grams and their corresponding SATD and non-SATD probability, and the key phrases. The overlapping key *h*-grams are concatenated into key phrases that are highlighted in orange.

Two of the authors examined such highlighted key phrases and determined their relevance to SATD classification. The Cohen's Kappa coefficient of the two authors' labels is 0.637, which indicates substantial inter-rater agreement. The two authors agreed that 2703 (76.36%) extracted key phrases are relevant to SATD classification, for example, "todo temporary hack to avoid unnecessary bug" in the Comment-1 in Table 10 , and "FIXME" and "probably not very efficient" in the Comment-2 in Table 11. Our model also gives high SATD probability to such key phrases. The two authors agree that 268 (7.57%) extracted key phrases are irrelevant to SATD classification, for example, "in a public setter" in Table 12. The authors consider these 268 key phrases as irrelevant for SATD classification, because they cannot use them to determine whether a comment is SATD or not. The two authors disagree on the relevance of 569 (16.07%) extracted key phrases for SATD classification, for example, "clone," "be around and we do not want to" in the Comment-4 in Table 13. These 569 key phrases usually have relative lower SATD probability by our model compared with the SATD probability of other more evident SATD-indicating key phrases. However, our model can still learn to extract them and use them as the basis for SATD classification. We will further compare our CNN-extracted key phrases with human-identified key phrases in the user study in RQ8 (see Section 5.8).

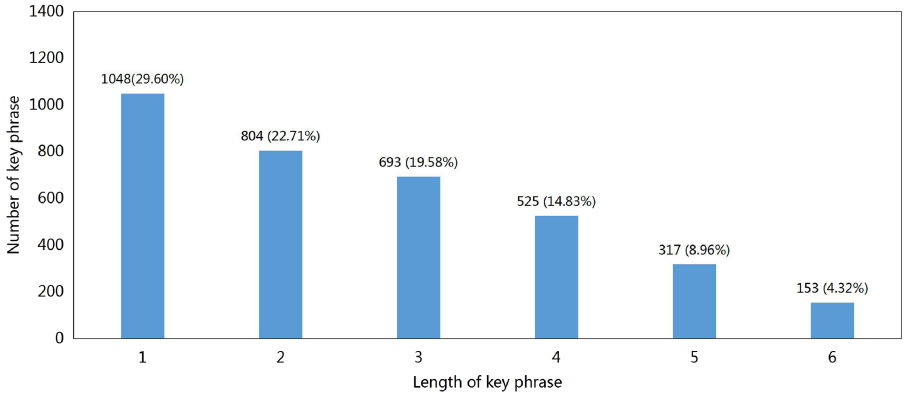

Fig. 6. Distribution of length of key phrases.

Table 10. Example-1 of CNN-extracted Key *h*-grams

Comment-1: // todo temporary hack to avoid unnecessary bug reports for subclasses		
Key *h*-grams	**non-SATD probalility**	**SATD probability**
todo	0.02136926	0.97863074
hack	0.06526527	0.93473473
todo temporary	0.00606645	0.99393355
todo temporary hack	0.00205301	0.99794699
hack to avoid	0.08711279	0.91288721
unnecessary bug	0.41933313	0.58066687

The words in orange are key phrases highlighted by our approach.

Table 11. Example-2 of CNN-extracted Key *h*-grams

Comment-2: FIXME: this is probably not very efficient, since it loads all methods for each call		
Key *h*-grams	**non-SATD probalility**	**SATD probability**
fixme	0.00653387	0.99346613
probably not very efficient	0.20815406	0.79184594

The words in orange are key phrases highlighted by our approach.

Our CNN model can effectively learn different numbers of variable-length text features from variable-length code comments for SATD classification. *Most of the extracted features are relevant and effective for SATD classification.*

5.4 RQ4: SATD Patterns and Their Explainability

5.4.1 Motivation. Previous work [6] manually summarizes the 62 most frequent SATD patterns for detecting SATD comments. Due to time and human limitations, it is very likely that they missed some less evident SATD patterns. In addition, although the text-mining approach [10] can achieve better performance in SATD identification than pattern-based SATD detection, it cannot explain the basis of its classification decisions. Our CNN-based approach can automatically extract SATD-indicating features from the training comments, and, based on the backtracking mechanism, we can map these CNN-extracted features backtrack to key *h*-grams and summarize key *h*-grams into SATD patterns (see Section 3.3). That is, our approach can potentially incorporate the

Table 12. Example-3 of CNN-extracted Key *h*-grams

Comment-3: Todo: not read, yet in a public setter		
Key *h*-grams	**non-SATD probalility**	**SATD probability**
todo	0.00587633	0.9941237
in a public setter	0.31935509	0.68064491

The words in orange are key phrases highlighted by our approach.

Table 13. Example-4 of CNN-extracted Key *h*-grams

Comment-4: //we clone dynamic scope, because this will be a new instance of a block any previously captured instances of this block may still be around and we do not want to start overwriting those values when we create a new one enebo once we make self, lastclass, and lastmethod immutable we can remove duplicate		
Key *h*-grams	**non-SATD probalility**	**SATD probability**
clone	0.39748379	0.60251621
around	0.23387226	0.76612774
be around	0.28207757	0.71792243
overwriting those values	0.32154414	0.67845586
be around and	0.37671376	0.62328624
be around and we do	0.34089262	0.65910738
overwriting those values when we create	0.23327953	0.76672047
this will be a new instance	0.37289290	0.62710710
because this will be a new	0.27444737	0.72555263
and we do not want to	0.38323630	0.61676370

The words in orange are key phrases highlighted by our approach.

performance advantage of a machine-learning-based approach and the explainability advantage of pattern-based approach. Based on the extracted key phrases, we further summarize a list of SATD patterns for helping developers identify and understand the prominent SATD indicators in source code comments. In this RQ, we evaluate the quality of these summarized SATD patterns. To answer this research question, we compared our CNN-based SATD patterns with the human-summarized patterns. We also wanted to investigate the SATD characteristics of the 10 projects listed in Table 3. Understanding these characteristics is important to explain the results of model performance, generalizability, and adaptability experiments.

5.4.2 Approach. We trained our model with the 10 projects' comments. We obtained a set of SATD patterns from the key *h*-grams that our CNN model extracts from all the manually labeled SATD comments of the 10 projects. We then compared the length distribution and the vocabulary of our CNN-based SATD patterns and the 62 human-summarized patterns. We also investigated the overlapping and distinction between our CNN-based SATD patterns and the 62 human-summarized patterns in Reference [6]. Finally, using the set of CNN-based SATD patterns, we collected statistics about the distribution of these SATD patterns in different projects, in terms of what patterns a project has and what project(s) a pattern appears in.

5.4.3 Results. We identified in total 700 SATD patterns. Figure 7 shows the distribution of SATD patterns (ours and those in Reference [6]) of different lengths. We can see that both sets of patterns are of variable length, ranging from one word to six words. For all different lengths, our CNN-based patterns are much more than human-summarized patterns. For example, our patterns include 175 one-word patterns, 168 two-word patterns, and 236 three-word patterns, while human

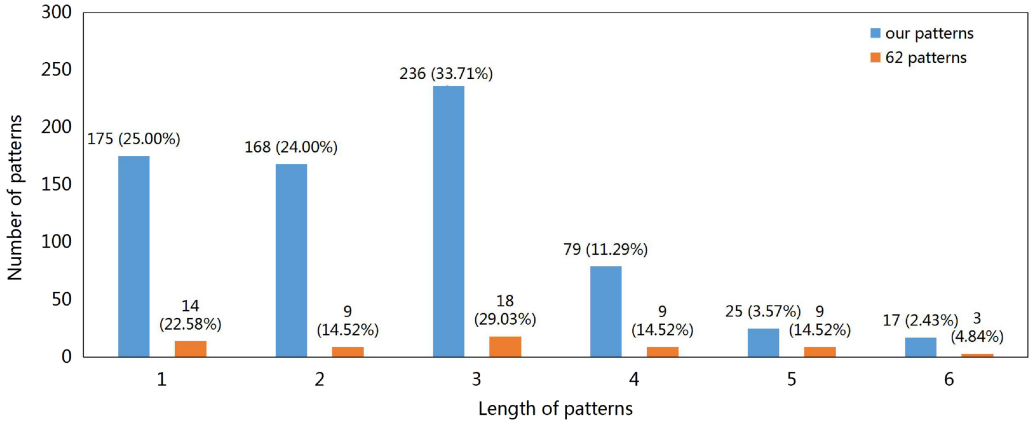

Fig. 7. Distribution of length of patterns.

Table 14. Top SATD Patterns Extracted by Our Approach

1-gram pattern	2-gram pattern	3-gram pattern
todo	really **ugly**	not very efficient
hack	this needs	not very nice
xxx	**remove this**	would be better
sss	not sure	this could be
workaround	how to	we really need
temporary	should **probably**	kind of **hacky**
fixme	**not used**	how to handle
implement	**temporary solution**	would be nice
4-gram pattern	**5-gram pattern**	**6-gram pattern**
should check for strict	this is the **temporary solution**	offenders need to be more strongly
element should be created	we may be screwed here	a better way to handle this
forget to add ownership	need some format checking here	this will also inefficiently handle arrays
do we need this	is this really necessary	if this code ever gets hit
not appear to be	**this can be a mess**	**stupid** recreation of whole menu model
currently only works for	not appear to be used	complicated by the use of parameters
this is a **temporary**	simplified these settings a little	move it to a good home
needs to be moved	not actually make any difference	a little bit of overkill here

only summarized 14 one-word, 9 two-word, and 18 three-word patterns. However, the distribution of patterns extracted by the two methods is similar. The 62 human-summarized patterns contain very few project-unique words, while our CNN-based SATD patterns contain approximately 50 project-unique words that are used in only one or two projects. For example, our method identifies "revisit" as a SATD pattern, but the human analysis did not identify it, because it appears only in EMF. That is, our CNN-based SATD patterns covers a much more diverse vocabulary (including not only commonly used words but also less-frequently used words) related to SATDs than human-summarized patterns.

Table 14 shows the top eight SATD patterns of different lengths identified by our method, and the patterns that match or contain the 62 human-summarized patterns are highlighted in bold. We can observe that our CNN-based SATD patterns are intuitive phrases, similarly to human-summarized

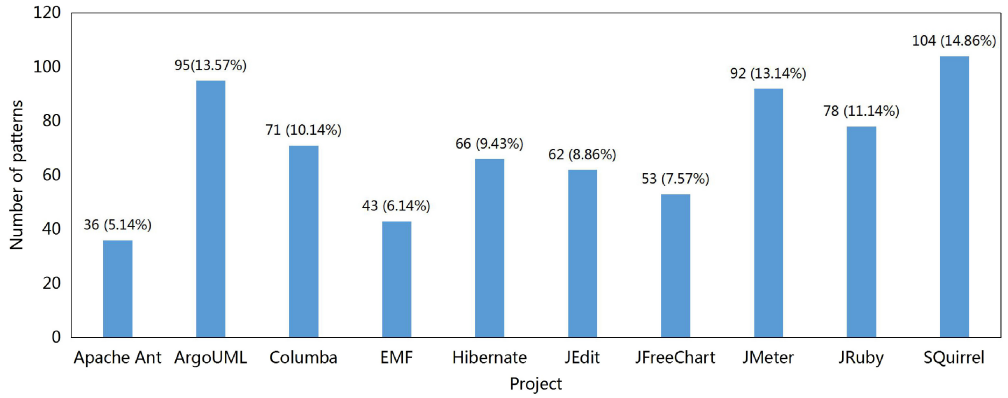

Fig. 8. Distribution of number of SATD patterns in the 10 projects.

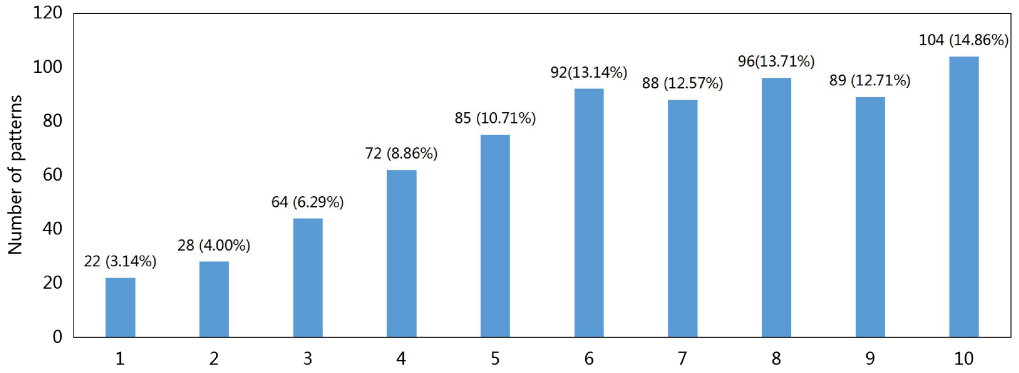

Fig. 9. Distribution of number of projects in which a pattern appears.

patterns. Our CNN-based SATD patterns cover 58 (93.6%) of the 62 human-summarized patterns. The four human-summarized patterns that are not in our SATD patterns are as follows: "this is bs," "kaboom," "barf," and "toss it." We search the 62,566 after-filtering comments in our dataset but do not find these four phrases. That is, these four phrases are used in the original 259,229 comments but are not included in the dataset that our method analyzes. Our method identifies many more SATD patterns that are not included in the 62 previously manually identified patterns. As shown in Table 14, many of these patterns do not involve commonly used SATD indicators (e.g., "todo," "hack," "fixme," "temporary solution"). Such patterns are hard to identify by just human observation.

Figure 8 shows the number of patterns that each of the 10 projects has. We can see that all projects use many different SATD patterns to describe SATDs. SQuirrel uses the most number of patterns (104, 14.86% of all our SATD patterns). ArgoUML and JMeter also use many patterns, which are 95 (13.57%) and 92 (13.14%). Apache Ant and EMF use relatively fewer numbers of patterns compared with other projects, which are 36 (5.14%) and 43 (6.14%). Considering the small number of SATD comments in many projects (e.g., Apache Ant, EMF, SQuirrel), we can see that even developers in the same project use very diverse SATD patterns when describing SATDs.

Figure 9 shows the distribution of the number of projects in which a SATD pattern appears. We can see that many patterns are commonly used in many projects. For example, 377 (53.86%) patterns appear in at least seven projects. At the same time, there are also a non-negligible number

Table 15. Top SATD Patterns in Each Project Extracted by Our Approach

Apache Ant	ArgoUML	Columba	EMF	Hibernate
xxx	todo	todo	todo	todo
todo	temporary	fixme	remove	remove
hack	hack	implement	**revisit**	yuck
error	fixme	workaround	hack	*ugly*
not sure	remove	better	workaround	fix
ugly	would be better	hack	why	hack
this is too hacky	remove this	fix	fix	should this
quick fix	do we need this	xxx	I don't know if	better to
remove	this can be a mess	*stupid*	kind of hacky	fixme
why not	more is required	is not thread-safe	this would be wrong	temporary
JEdit	**JFreeChart**	**JMeter**	**JRuby**	**SQuirrel**
hack	todo	todo	todo	todo
fix	fixme	hack	fixme	hack
todo	implement this	should this	need to	delete
xxx	not used	not used	check	kludge
stupid	unused	later	xxx	not yet
remove	hack	fixme	hack	fix
temporary	*ugly*	empty block	probably not	not used
better way	*negative*	problems occur	missing	remove
review	complete this	not yet handled	temporary	*evil*
misleading	handle	is it the best way	kludge	vertify this

Common SATD patterns are underlined, project-unique patterns are in bold and sentiment ones are in italic.

of SATD patterns appearing in only a few projects. For example, 114 (13.43%) patterns appear in three or less projects. Table 15 shows the top 10 SATD patterns in each of the 10 projects. We can also observe that many projects share some common SATD patterns (underlined), such as "todo," "xxx," "fixme" and "hack." At the same time, some SATD patterns appear in only one or two projects. For example, "revisit" is only used in EMF to describe SATDs. An interesting observation is that negative sentiment words are often used to describe SATDs, such as "ugly," "yuck," "stupid," "negative" and "evil." But the use of specific negative sentiment words differs from one project to another.

Our CNN-based method can extract intuitive SATD patterns, similar to human-summarized patterns, and it extracts more comprehensive SATD patterns with more diverse vocabulary compared with human-summarized patterns. *Our CNN-based SATD patterns capture not only common SATD features, but also less evident or project-unique features, as well as sentiment features. They help explain the common and distinct SATD characteristics of different projects.*

5.5 RQ5: Within-Project Classification Performance

5.5.1 Motivation. Our approach uses a CNN to extract SATD features from the comment data. We compare the effectiveness of our CNN-based features and the traditional text-mining-based features for predicting STADs and understand the applicability of our approach in projects with different SATD characteristics.

Table 16. Within-Project Precision, Recall, and F1-score of Our Approach and Traditional Text-mining Approach

Project	Precision			Recall			F1-score		
	TM	Our	Improv.	TM	Our	Improv.	TM	Our	Improv.
Apache Ant	0.450	0.323	−28.32%	0.818	0.719	−12.08%	0.581	0.445	−23.36%
ArgoUML	0.497	**0.905**	82.06%	0.828	**0.962**	16.19%	0.621	**0.932**	50.09%
Columba	0.458	0.611	33.33%	**0.917**	0.942	2.76%	0.611	0.741	21.25%
EMF	0.438	0.423	−3.41%	0.778	0.719	−7.51%	0.560	0.532	−5.00%
Hibernate	**0.692**	0.860	24.29%	0.878	0.916	4.28%	0.774	0.887	14.57%
JEdit	0.480	0.468	−2.60%	0.667	0.929	39.31%	0.558	0.622	11.44%
JFreeChart	0.688	0.679	−1.24%	**0.917**	0.958	4.49%	**0.786**	0.795	1.18%
JMeter	0.525	0.821	56.46%	0.808	0.918	13.68%	0.636	0.867	36.24%
JRuby	0.646	0.847	31.16%	0.756	0.918	21.38%	0.697	0.881	26.47%
SQuirrel	0.370	0.750	102.50%	0.625	0.888	42.04%	0.465	0.813	74.80%
Average	0.524	0.669	25.02%	0.799	0.887	12.45%	0.629	0.752	19.50%

The best results of each method are in bold and the worst ones are underlined.

5.5.2 Approach. We set up 10 within-project classification experiments with the 10 projects listed in Table 3 that extract SATD patterns from we projects. For each project, we randomly select 90% comments to train a SATD classification model and use the the remaining 10% comments as the testing data to test the performance of the trained model. We compare the performance of our approach with that of the traditional text-mining-based SATD classification [10]). We compute the improvement ratio of our approach over the text-mining method and natural processing language approach for the precision, recall, and F1-score. In addition, we use the Wilcoxon signed-rank test [30] to test whether the differences of precision, recall, and F1-score in the 10 experiments is statistically significant at the p-value < 0.05. We also use the Cliff's delta (δ) to quantify the amount of difference between the three approaches. The amount of difference is considered negligiable ($|\delta| < 0.147$), small ($0.147 \leq |\delta| < 0.33$), medium ($0.33 \leq |\delta| < 0.474$), or large ($|\delta| \geq 0.474$).

5.5.3 Result. Table 16 lists the precision, recall, and F1-score metrics in the 10 within-project classification experiments. Overall, our approach has a much better precision and a moderate improvement in recall compared with the text-mining method. On average, our approach achieves the 25.02%, 12.45%, and 19.50% improvement in precision, recall, and F1-score than the text-mining method, respectively. The Wilcoxon signed-ranked tests show that the improvements in precision, recall, and F1-score are all statistically significant at the p-value < 0.05. The Cliff's delta of F1-score is 0.5 (larger than 0.474), which indicates that our approach significantly improves F1-score over the text-mining approach by a large margin.

Our approach achieves the best F1-score (0.932) on ArgoUML among all the 20 experiments, while it achieves the worst F1-score (0.445) on Apache Ant and the third worst F1-score (0.532) on EMF among all the 20 experiments. For ArgoUML, the F1-score of our approach is 50.09% higher than that of the text-mining method, while for Apache Ant and EMF, the F1-score of our approach is 23.06% and 5.00% lower than that of the text-mining method, respectively.

These two extreme performance results coincide with the fact that ArgoUML has the largest set of SATD comments (1413), while Apache Ant and EMF have the two smallest sets of SATD comments (131 and 104, respectively).

Although our approach performs worse than the text-mining method when the dataset is very small, it consistently outperforms (statistically significant at p-value < 0.05 by the Wilcoxon signed-rank test) than the text-mining method on all the other seven datasets that have more than

Table 17. Examples of Comparison between Our Approach and Text-mining Approach

Type	Comment
Comments with project uniqueness	// wtf? // REVISIT: Remove this code.
Comments with semantic variation	// not absolutely necessary, but does help with aggressive release // this is kinda the best we can do …

200 SATD comments. Furthermore, for SQuirrel that the text-mining method performs the worst (0.465 F1-score), our approach performs significantly better (0.813 F1-score). However, for Apache Ant and EMF where our approach performs the worst (0.445 and 0.532 F1-score), the text-mining method is itself only moderately or marginally better (0.581 and 0.56 F1-score).

Among the 10 projects, our approach achieves more than 50% improvement in precision on SQuirrel (102.50%), ArgoUML (82.06%), and JMeter (56.46%) and achieve about 40% improvement in recall on SQuirrel (42.05%) and JEdit (39.21%). Comparing our CNN-based SATD patterns and the top 10 features selected by the text-mining method in SQuirrel, ArgoUML, JMeter, and JEdit, we find that our approach can extract much more meaningful SATD patterns while the text-mining method select many meaningless features such as "us," "and," and "thi." As a result, our approach achieves much better precision and recall than the text-mining method in these four projects.

Among the 10 projects, there is only one project (JFreeChart) on which our approach and the text-mining method achieve almost the same performance, and both methods achieve better recall on JFreeChart than 8 of the 10 other projects. The text-mining approach actually achieves the best F1-score on JFreeChart, but this performance is only the 6th ranked by our approach. We find that there are almost 70% overlaps between our CNN-based SATD patterns and the top 10 features selected by the text-mining method in JFreeChart. This explains why the two methods have a very close performance for this project.

To further understand the improvement of our approach over traditional text-mining approach, we further compare examples of code comments that are correctly classified by our approach but misclassified by the text-mining approach. We find that there are two kinds of comments that are frequently misclassified by the text-mining approach. Some examples are shown in Table 17:

- Comments with project uniqueness that refer to the comments with SATD patterns that only appear in one or two projects. This kind of comments often contains some rare but important features (e.g., "wtf" and "REVISIT"), which are highly related to SATD comments. However, the text-mining approach actually removes them during the feature selection phase, thus misclassifying this kind of comments as non-SATD. In contrast, our approach can extract such project-unique SATD patterns and make more accurate classification.
- Comments with semantic variation that refer to the comments containing context-sensitive semantic patterns. As shown in Table 17, comments with semantic variation is difficult to classify without considering the overall sentence context, such as "not absolutely necessary" and "kinda the best." For traditional text-mining approach, it often considers the meaning of words or phrases independent of the sentence context. In such cases, text-mining approach often misclassifies the comments as SATDs. On the contrary, our approach can capture sentence-level embeddings in the model and fully use the context to make accurate classification.

Our CNN-based approach can statistically significantly improve the SATD classification performance over the traditional text-mining-based methods. *This can be attributed to the superior capability of CNN to learn to extract more meaningful and more comprehensive*

Table 18. Precision, Recall, and F1-score of Our Approach and Traditional Text-mining Approach in 9 → 1 Setting

Project	Precision			Recall			F1-score		
	TM	Our	Improv.	TM	Our	Improv.	TM	Our	Improv.
Apache Ant	0.565	0.584	3.36%	0.471	0.758	60.93%	0.513	0.660	28.65%
ArgoUML	0.815	0.816	1.23%	**0.856**	**0.950**	10.98%	**0.835**	**0.878**	5.15%
Columba	0.817	0.830	1.59%	0.805	0.875	8.70%	0.811	0.852	5.06%
EMF	0.636	0.793	24.69%	0.473	0.594	23.47%	0.543	0.679	25.05%
Hibernate	**0.876**	**0.930**	6.16%	0.729	0.743	1.92%	0.796	0.826	3.77%
JEdit	0.772	0.773	0.13%	0.364	0.489	34.35%	0.495	0.599	21.01%
JFreeChart	0.627	0.686	9.41%	0.733	0.802	9.41%	0.676	0.739	9.32%
JMeter	0.858	0.873	1.75%	0.770	0.787	2.21%	0.811	0.828	2.09%
JRuby	0.855	0.805	−5.85%	0.752	0.930	23.67%	0.800	0.863	7.88%
SQuirrel	0.784	0.794	1.26%	0.597	0.692	15.91%	0.678	0.739	9.00%
Average	0.761	0.788	3.55%	0.655	0.762	16.34%	0.696	0.766	10.06%

The best results of each method are in bold and the worst ones are underlined.

SATD-indicating features that the traditional feature selection process of the text-mining method. However, we should be cautious that the CNN may not be well trained when the training data are too small.

5.6 RQ6: Cross-Project Model Generalizability

5.6.1 Motivation. As discussed in RQ4, different projects have different SATD characteristics, and these characteristics may affect the performance of SATD classification methods. Ideally, we want to deploy a model trained using some projects to a new project without much performance degradation. The more generalizable a model's SATD features are, the less a model degrades across projects, and thus the easier it is to deploy the model in practice. Therefore, we investigate the generalizability of SATD features learned by our approach across projects and compare our model's generalizability with that of traditional text-mining-based method [10].

5.6.2 Approach. We set up two experimental settings: a rich training datasetting and a limited training datasetting. In the rich training datasetting, we use nine projects' comments as the training data and the other one project's comments as the testing data. We refer to this setting as 9 → 1, and we have 10 experiments for each method in this setting. In the limited training datasetting, we use one project's comments as the training data and the other project's comments as the testing data. We refer to this setting as 1 → 1, and we have 90 experiments for each method in this setting. We use 1 →? to represent the nine experiments of the same training project, and ? → 1 to represent the nine experiments of the same testing project. For each experiment, we compute precision, recall, and F1-score. We show all three metrics for the 9 → 1 setting and show only F1-score for the 1 → 1 setting due to space limitations.[3] We use the Wilcoxon signed-rank test [30] to test whether the differences of F1-score in the experiments is statistically significant at the p-value < 0.05. We also use the Cliff's delta (δ) to quantify the amount of difference between two approaches. The amount of difference is considered negligiable ($|\delta| < 0.147$), small ($0.147 \leq |\delta| < 0.33$), medium ($0.33 \leq |\delta| < 0.474$), or large ($|\delta| \geq 0.474$).

5.6.3 Result. We first review the results of the ten 9 → 1 experiments (see Table 18). Our approach performs better than the text-mining method in all of the experiments, and the

[3] https://github.com/goodchar/CNN-based_SATD/blob/master/othermetrics.pdf.

improvement in F1-score is statistically significant by Wilcoxon signed-rank test at p-value < 0.05. The Cliff's delta (δ) of F1-score is 0.38 (in the medium level), which means that our approach improves F1-score over the text-mining approach by a medium margin. However, we can see that a rich training datasetting is more beneficial to the text-mining approach than to our approach. Comparing the average F1-score of the ten 9 → 1 experiments with that of the 10 within-project experiments, the text-mining method's average F1-score improves 10.6% (0.696 versus 0.629), while our approach's average F1-score improves 1.86% (0.766 versus 0.752). The F1-score improvement ratio narrows for 6 of the 10 projects (ArgoUML, Columba, Hibernate, JMeter, JRuby, and SQuirrel) from 14.57%–74.80% in the within-project experiments down to 3.77%–9.00% in the 9 → 1 experiments. Observing the SATD patterns in these six projects suggests that they share many prominent SATD patterns, such as "todo," "hack," and "fixme." As such, the text-mining model trained with some other projects can work well on one of these projects.

However, rich training data from many other projects may not always boost the classification performance for an individual testing project. In fact, the average F1-score of the text-mining method degrades 10.4% in the 9 → 1 experiments for the four projects (Apache Ant, EMF, JEdit, and JFreeChart) compared with the average F1-score of the text-mining method for these four projects in the within-project experiments. This is because these four projects have small numbers of SATD comments (104–256) whose SATD patterns are often not the most prominent ones in other projects. For example, JEdit and JFreeChart have some unique SATD features (e.g., "nasty," "wtf," and "fudge") that are not commonly used in other projects. As a result, training the text-mining model with other projects actually leads to the model's bias to the prominent SATD patterns in other projects, but overlooking the testing-project-unique features, which degrades the model's generalizability across projects.

In contrast, our approach can learn more comprehensive SATD features so that it can keep a more balanced view between the most prominent features and those project-unique ones. This makes our model more generalizable across projects. This generalizability leads to small performance degradation (which is of 5.29% average decline in F1-score) in seven projects compared with the corresponding within-project experiments, but the extent is not as bad as the text-mining method (e.g., the performance degradation on JEdit and JFreeChart). However, the generalizability of our approach leads to significant improvements in F1-score for the three projects with the lowest F1-scores in the within-project experiments: Apache Ant (0.446 to 0.660), EMF (0.532 to 0.679), and Columba (0.741 to 0.852). Note that these three projects have the least numbers of SATD comments that are not sufficient for training a good CNN-based model. However, the CNN-based model can learn useful SATD features from other projects for predicting SATD comments in these three projects.

This generalizability advantage of our approach becomes even more evident in the limited training datasetting. Tables 19 and 20 show the results of 1 → 1 experiments for the text-mining and our approach, respectively. Rows represent ? → 1 experiments, while columns represent 1 →? experiments. For the ? → 1 experiments, the average F1-score (Ave-? → 1) of our approach is higher than that of the text-mining method on eight of the 10 testing projects, especially on Apache Ant (69.38% higher), EMF (96.32% higher), and SQuirrel (45.61% higher). This suggests that our approach has good generalizability even for projects with small numbers of SATD comments and less prominent SATD features. This is largely consistent with the results of 9 → 1 experiments in Table 18. For the 1 →? experiments, the average F1-score (Ave-1 →?) of our approach is higher than that of the text-mining method on seven of the 10 training projects, especially on ArgoUML (109.43%), Columba (38.16%), and Hibernate (40.60%). Note that ArgoUML and Hibernate have the largest (1413) and the third-largest (472) set of SATD comments, respectively. This suggests that

Table 19. Traditional Text Mining: F1-score in 1 → 1 Setting (row ? → 1 Refers to Other Projects are Used to Predict This One Project and Column 1 →? Means This One Project is Used to Predict Others)

	Apache Ant	ArgoUML	Columba	EMF	Hibernate	JEdit	JFreeChart	JMeter	JRuby	SQuirrel	Ave-? → 1
Apache Ant	-	0.176	0.302	0.274	0.267	0.268	0.270	0.244	0.302	0.220	0.258
ArgoUML	0.640	-	0.695	0.805	0.696	0.736	0.756	0.736	0.681	0.744	**0.721**
Columba	0.589	0.382	-	0.591	0.567	0.500	0.465	0.649	0.496	0.543	0.531
EMF	0.431	0.213	0.248	-	0.201	0.225	0.300	0.272	0.309	0.252	0.272
Hibernate	0.683	0.505	0.639	0.745	-	0.662	0.668	0.693	0.633	0.674	0.656
JEdit	0.358	0.347	0.357	0.288	0.377	-	0.394	0.338	0.459	0.435	0.373
JFreeChart	0.327	0.328	0.466	0.535	0.497	0.384	-	0.524	0.522	0.340	0.436
JMeter	0.677	0.413	0.541	0.593	0.576	0.617	0.592	-	0.568	0.585	0.574
JRuby	0.552	0.543	0.696	0.548	0.724	0.570	0.584	0.714	-	0.527	0.606
SQuirrel	0.496	0.310	0.383	0.556	0.458	0.432	0.515	0.491	0.462	-	0.456
Ave-1 →?	0.528	0.357	0.481	**0.548**	0.485	0.488	0.505	0.518	0.492	0.480	-

The best F1-scores of each method are in bold and the worst ones are underlined.

our approach can learn more generalizable features from projects with large numbers of SATD comments than the text-mining method.

However, the 1 → 1 experiments reveal again the challenge in training a good CNN-based SATD classification model with limited numbers of SATD comments. We can see that the Ave-? → 1 F1-score of our approach is actually worse than the text-mining approach on the testing projects ArgoUML (12.48% lower) and Hibernate (2.43% lower), which have larger numbers of SATD comments than the training projects. Furthermore, the Ave-1 →? F1-score of our approach is actually worse than the text-mining approach on the training projects Apache Ant (26.28% lower), EMF (14.19% lower) and JEdit (26.78% lower) that have smaller numbers of SATD comments than the testing projects. As we will show in the RQ5, this challenge can be well addressed with transfer learning mechanism, because our approach has superior adaptability.

Our CNN-based approach has better generalizability across projects than the traditional text-mining-based method*. When the projects' SATDs do not use the prominent SATD patterns commonly seen in other projects, our approach is statistically significantly better. But when the training projects have limited numbers of SATD comments or the testing projects have much larger numbers of SATD comments than the training projects, the generalizability of our approach may be limited.*

5.7 RQ7: Cross-Project Model Adaptability

5.7.1 Motivation. In RQ5 and RQ6, we see the superior performance and generalizability of our approach over the text-mining approach. However, we also identify the challenge in effectively training our CNN-based model with limited training data. Increasing the size of training data requires significant efforts or even is impossible when the raw dataset itself is small. Transfer learning is a proven-effective mechanism to address this limited training data challenge. Transfer learning essentially adapts the knowledge gained while solving one task to a different but related task. It is beneficial for narrowing down the scope of possible model parameters on a task by reusing and adjusting the parameters of a trained model on a different but related task [31]. As such, it can reduce the amount of training data required for the target task. We apply transfer learning to address the training challenge of our CNN-based model and also to investigate the adaptability of our model across projects.

Table 20. Our Approach: F1-score in 1 → 1 Setting (row ? → 1 Refers to Other Projects Are Used to Predict This One Project and Column 1 →? Means This One Project Is Used to Predict Others. The best F1-scores of each method are in bold and the worst ones are underlined.

	Apache Ant	ArgoUML	Columba	EMF	Hibernate	JEdit	JFreeChart	JMeter	JRuby	SQuirrel	Ave-? → 1	Ave-? → 1 of TM	Imp.-2
Apache Ant	-	0.608	0.468	0.257	0.450	0.328	0.306	0.367	0.564	0.587	0.437	0.258	69.38%
ArgoUML	0.427	-	0.817	0.459	0.808	0.381	0.603	0.821	0.620	0.739	0.631	0.721	−12.48%
Columba	0.409	0.773	-	0.398	0.655	0.443	0.483	0.603	0.604	0.463	0.537	0.531	1.13%
EMF	0.398	0.652	0.591	-	0.653	0.341	0.496	0.698	0.476	0.501	0.534	0.272	96.32%
Hibernate	0.388	0.868	0.819	0.450	-	0.411	0.582	0.756	0.621	0.699	0.622	0.656	−0.61%
JEdit	0.326	0.569	0.357	0.434	0.504	-	0.350	0.380	0.473	0.330	0.414	0.373	10.99%
JFreeChart	0.359	0.661	0.550	0.432	0.638	0.355	-	0.460	0.498	0.428	0.487	0.436	11.70%
JMeter	0.402	0.873	0.798	0.459	0.816	0.356	0.633	-	0.704	0.708	0.639	0.574	11.32%
JRuby	0.397	0.850	0.795	0.688	0.806	0.355	0.603	0.788	-	0.723	**0.667**	0.606	10.07%
SQuirrel	0.397	0.875	0.786	0.655	0.807	0.246	0.654	0.823	0.734	-	0.664	0.456	45.61%
Ave-1 →?	0.389	**0.748**	0.665	0.470	0.682	0.357	0.523	0.633	0.588	0.575	-	-	-
Ave-1 →? of TM	0.528	0.357	0.481	0.548	0.485	0.488	0.505	0.518	0.492	0.480	-	-	-
Imp.-1	**−26.28%**	109.43%	38.16%	**−14.19%**	40.60%	**−26.78%**	3.63%	22.18%	19.56%	19.86%	-	-	-

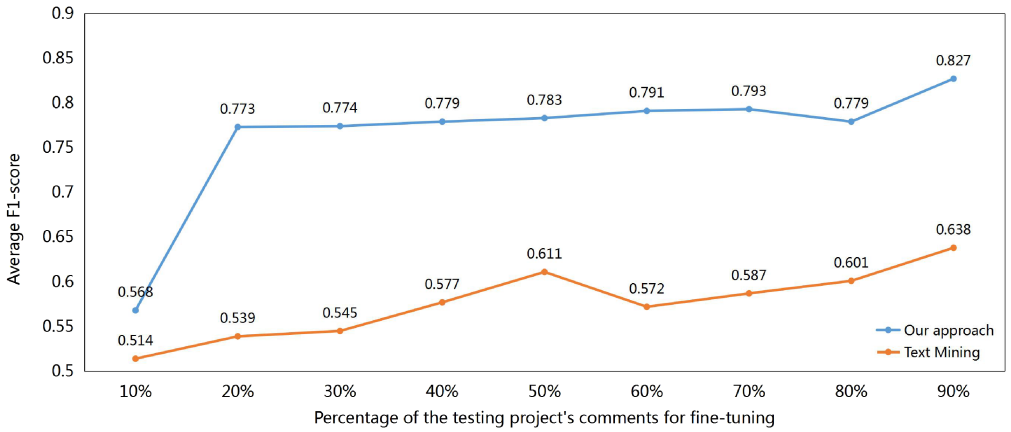

Fig. 10. Average of F1-score in the nine $n/10$ ($1 \leq n \leq 9$) fine-tuning settings.

5.7.2 Approach. We use the traditional text-mining-based method [10] as the baseline. As we are interested in the model adaptability, we use the limited training datasetting, i.e., $1 \rightarrow 1$ setting, in which the model trained with the data of a source project will be transferred to the date of a target project. When it comes to the transfer learning strategy, we choose fine-tuning the Convolutional Neural Network to not only retrain the classifier on top of the CNN on the target project dataset, but to also fine-tune the weights of the pre-trained network by continuing the backtracking through network layers. That is, we fine-tune all the layers of the CNN in this work. This fine-tuning strategy is motivated by the observation that the lower-layer features of a CNN contain more generic features (e.g., "TODO," "Fixme," etc.) that should be useful to many projects, but higher-layers of the CNN becomes progressively more specific to a project. We first train a model using a source project's comments. Then, we fine-tune the trained model with a target project's comments. We randomly split the target project's comments into 10 subsets. For each fine-tuning experiment, we use n ($1 \leq n \leq 9$) subsets as the fine-tuning data and the rest $10 - n$ subsets as the testing data.

Therefore, we have 810 (90*9) experiments for each method. We compute the precision, recall, and F1-score for each fine-tuning experiment. We compute the average F1-score of a method for each n setting and also compute the average F1-score of a method in the $? \rightarrow 1$ experiments and the $1 \rightarrow ?$ experiments.

5.7.3 Result. Figure 10 shows the average F1-score of the fine-tuning experiments with 10% to 90% target project's comments as fine-tuning data. We can see that our approach always perform better than the text-mining method. Although the two methods achieve similar F1-score (about 0.55) at 10% fine-tuning data, the F1-score of our approach jumps to 0.79 at 20% fine-tuning data. Furthermore, the F1-score of our approach at $>$ 20% fine-tuning data remains very stable and has about 0.05 further improvement at 90% fine-tuning data. In contrast, the best F1-score of the text-mining method is only about 0.6 at 50% fine-tuning data and at 90% fine-tuning data. That is, even with much more fine-tuning data, the text-mining method does not even come close to the performance of our approach with much less fine-tuning data.

The F1-score 0.79 at 20% fine-tuning data by our approach is 39.82% higher than that of the 90 $1 \rightarrow 1$ experiments without fine-tuning (0.565), and is even higher than the average F1-score (0.752) of the within-project experiments. That is, with only 20% of a target project's comments for fine-tuning, a CNN-based model trained with a source project can be effectively adapted to the

target project. The resulting target CNN model significantly improves the original source CNN model for predicting the target project's SATDs.

Tables 21 and 22 present the F1-score of the 90 1 → 1 experiments with 20% fine-tuning data by the text-mining method and our approach, respectively. We can see that the F1-scores of our approach with fine-tuning are significantly higher than those of the text-mining method with fine-tuning in all 90 1 → 1 experiments. The improvement ratios for the average F1-score of the ? → 1 experiments and the 1 →? experiments can be found in the *Imp*. − 4 column and the *Imp*. − 3 row, respectively.

Table 21 shows that fine-tuning the text-mining model in the ? → 1 setting is effective only for the test projects with small numbers of SATD comments, such as Apache Ant and EMF. Note that the improvement ratio is very high, because the average F1-score without fine-tuning is extremely low (0.258 for Apache Ant and 0.272 for EMF). For the other eight testing projects in the ? → 1 setting, fine-tuning leads to only small improvements (less than 11.39%) in F1-score. For the two testing projects ArgoUML and Columba, fine-tuning results in a worse target model. In the 1 →? setting, similar observation can be made. For the source project EMF whose model is not of high-quality due to the limited training data, fine-tuning also results in a worse target model.

In contrast, Table 22 shows that fine-tuning is very effective in boosting the F1-score of our CNN-based model. First, fine-tuning never results in a worse target model. Even for the testing project like ArgoUML that have much larger set of SATDs than the training project, fine-tuning with 20% of its comments still improves the target model's F1-score by 21.39% (0.631 to 0.766). However, even for the source model obtained from the large training data like ArgoUML and Hibernate, fine-tuning it with some testing-project's comments can still improve the target model's performance. Second, comparing the F1-score improvement ratios with and without fine-tuning by our approach and those by the text-mining method, we can see that the improvement ratios by our approach is much higher in both ? → 1 (*Imp*. − 2) and 1 →? (*Imp*. − 1) settings. Note that this higher improvement ratio is on the basis that our approach without fine-tuning already perform better than the text-mining approach without fine-tuning in seven of the 10 projects.

Furthermore, we can see that fine-tuning can effectively address the limited training data challenge for our approach, even when the source model is not of high quality due to the limited training data. Look at the column of Apache Ant and EMF in Table 22. Using one of these projects as the training data, the F1-score of our approach without fine-tuning in the 1 →? setting is very poor (F1-score is 0.389 and 0.470, respectively). They are even worse than that of the text-mining method without fine-tuning (see Table 20). However, with 20% fine-tuning data, the target model performance improve significantly. That is, starting with a project with only a small number of SATD comments like Apache Ant or EMF and only 20% of the training data from the other projects, we can obtain very high quality SATD classification models (the average F1-score for *ApacheAnt* →? is 0.800 and the average F1-score for *EMF* →? is 0.778).

Our approach has superior adaptability than the traditional text-mining-based method. *This adaptability allows a source model to be adapted into a high-quality model for a target project with 20% of the target project fine-tuning data. This will help reduce the amount of training data needed to deploy a high-quality SATD classification model for multiple projects.*

5.8 RQ8: User Study of Model Performance and Explainability

5.8.1 Motivation. The RQ1 to RQ6 evaluate the explainability, performance, generalizability, and adaptability of our approach using a public dataset in Reference [11]. Last, we further confirm the performance of our approach for cross-project SATD prediction and the intuitiveness and

Table 21. Traditional Text Mining: F1-score in 1 → 1 Setting with 20% Fine-Tuning Data (Row ? → 1 Refers to Other Projects are Used to Predict This One Project and Column 1 →? Means This One Project is Used to Predict Others)

Target	Apache Ant	ArgoUML	Columba	EMF	Hibernate	JEdit	JFreeChart	JMeter	JRuby	SQuirrel	Ave-? → 1	Ave-? → 1 (w/o ft)	Imp.-2
Apache Ant	-	0.243	0.433	0.382	0.438	0.354	0.400	0.422	0.383	0.388	0.383	0.258	48.45%
ArgoUML	0.620	-	0.631	0.668	0.694	0.664	0.675	0.671	0.665	0.681	0.663	0.721	−8.04%
Columba	0.430	0.268	-	0.422	0.354	0.373	0.436	0.480	0.358	0.434	0.395	0.531	−25.61%
EMF	0.602	0.458	0.632	-	0.634	0.604	0.576	0.605	0.649	0.589	0.594	0.272	118.38%
Hibernate	0.663	0.554	0.725	0.645	-	0.686	0.712	0.703	0.679	0.723	0.677	0.656	3.20%
JEdit	0.369	0.381	0.435	0.417	0.408	-	0.451	0.361	0.485	0.420	0.414	0.373	10.99%
JFreeChart	0.463	0.363	0.528	0.478	0.465	0.448	-	0.506	0.558	0.500	0.479	0.436	9.86%
JMeter	0.705	0.477	0.668	0.625	0.609	0.601	0.659	-	0.647	0.686	0.631	0.574	9.93%
JRuby	0.702	0.577	0.688	0.605	0.711	0.731	0.646	0.698	-	0.719	0.675	0.606	11.39%
SQuirrel	0.513	0.350	0.514	0.485	0.448	0.547	0.489	0.531	0.470	-	0.483	0.456	5.92%
Ave-1 →?	0.563	0.408	0.584	0.525	0.529	0.556	0.560	0.553	0.544	0.571	-	-	-
Ave-1 →? (w/o ft)	0.528	0.357	0.481	0.548	0.485	0.488	0.505	0.518	0.492	0.480	-	-	-
Imp.-1	6.63%	14.29%	21.41%	−4.20%	9.07%	13.93%	10.89%	6.76%	10.57%	18.96%	-	-	-

Table 22. Our Approach: F1-score in 1 → 1 Setting with 20% Fine-Tuning Data (Row ? → 1 Refers to Other Projects Are Used to Predict This One Project and Column 1 →? Means This One Project Is Used to Predict Others)

Target	Apache Ant	ArgoUML	Columba	EMF	Hibernate	JEdit	JFreeChart	JMeter	JRuby	SQuirrel	Ave-? → 1	Ave-? → 1 of TM	Imp.-4	Ave-? → 1 (w/o ft)	Imp.-2
Apache Ant	-	0.834	0.759	0.583	0.774	0.723	0.728	0.802	0.727	0.816	0.750	0.383	95.82%	0.437	71.62%
ArgoUML	0.795	-	0.821	0.790	0.825	0.514	0.603	0.878	0.792	0.872	0.766	0.663	15.54%	0.631	21.39%
Columba	0.828	0.836	-	0.763	0.790	0.623	0.719	0.891	0.785	0.853	0.788	0.395	99.49%	0.537	46.74%
EMF	0.759	0.736	0.738	-	0.756	0.682	0.721	0.730	0.673	0.752	0.727	0.594	22.39%	0.534	36.14%
Hibernate	0.803	0.898	0.897	0.752	-	0.442	0.615	0.801	0.654	0.720	0.731	0.677	7.98%	0.622	17.52%
JEdit	0.552	0.642	0.579	0.749	0.793	-	0.554	0.586	0.601	0.545	0.622	0.414	50.24%	0.414	50.24%
JFreeChart	0.804	0.879	0.768	0.832	0.897	0.564	-	0.723	0.699	0.702	0.763	0.479	59.29%	0.487	56.67%
JMeter	0.890	0.913	0.894	0.847	0.903	0.658	0.869	-	0.898	0.904	0.864	0.631	36.93%	0.639	35.21%
JRuby	0.871	0.899	0.923	0.816	0.918	0.647	0.858	0.916	-	0.907	0.862	0.675	27.70%	0.664	29.24%
SQuirrel	0.899	0.920	0.925	0.87	0.928	0.457	0.875	0.929	0.903	-	0.856	0.483	77.23%	0.667	28.92%
Ave-1 →?	0.800	0.840	0.812	0.778	**0.843**	**0.590**	0.727	0.806	0.748	0.786	-	-	-	-	-
Ave-1 →? of TM	0.563	0.408	0.584	0.525	0.529	0.556	0.560	0.553	0.544	0.571	-	-	-	-	-
Imp.-3	42.12%	105.80%	38.96%	48.19%	59.29%	6.12%	29.80%	45.79%	37.50%	37.59%	-	-	-	-	-
Ave-1 →? (w/o ft)	0.389	0.748	0.665	0.470	0.682	0.357	0.523	0.633	0.588	0.575	-	-	-	-	-
Imp.-1	**105.68%**	12.25%	22.04%	**65.53%**	23.56%	**65.11%**	38.92%	27.35%	27.16%	35.57%	-	-	-	-	-

Table 23. Three Java Projects for User-Study

Project	Number of Comments
Guava	6944
RxJava	2802
Commons-lang	3844

Table 24. Statistics of User-Study Comments

	Guava	RxJava	Commons-lang
Number of Comments	50	50	50
Number of SATDs Predicted by Our approach	9 (18%)	15 (30%)	11 (22%)
Number of SATDs by Majority Strategy	11 (22%)	18 (36%)	14 (28%)
Number of SATDs by All-agree Strategy	5 (10%)	7 (14%)	7 (14%)

explainability of the SATD features that our CNN model learns, using a separate dataset and by a user study.

5.8.2 Approach. We train our model with the 10 projects' comments. We collect the comments from the source code of three other three Java projects (Commons-lang, Guava, and RxJava). We randomly select the comments from each project and use the trained model to predict whether a selected comment is SATD or not. As we are interested in SATDs, we ensure a reasonable number of SATDs in the selected comments. Thus, we randomly drop the selected comments that are predicted as non-SATDs. The selection stops when we obtain 50 comments for each project. Tables 23 and 24 list the information about the three subject projects and the selected comments. For the comments predicted as SATDs by our model, we also retrieve the key phrases in the comments using our key phrase extraction approach.

We invited five graduate students from our school to participate in our study. These students have at least 3 years software development experience. We gave each participant background information about SATD and non-SATD comments. We then gave each participant a sheet of the 150 selected comments and asked them to label each comment as SATD or non-SATD. If the participants identify a comment as SATD, then they are asked to highlight the phrases in the comment they used as the basis for their judgment.

After collecting the participants' labeling results, we used Fleiss Kappa [32] to assess the inter-rater agreement on SATD/non-SATD decisions among the five participants. We also considered our model's SATD/non-SATD classification as an additional rater and use Fleiss Kappa to evaluate the inter-rater agreement among our model and the five participants. Then, we used two strategies to develop the ground-truth SATD/non-SATD labels for the 150 comments. First, the majority strategy labels a comment with the majority vote (i.e., 3 or above) by the five participants. Second, the all-agree strategy labels a comment as SATD if all five participants label it as SATD and otherwise as non-SATD. We used the obtained ground-truth SATDs to evaluate the precision, recall, and F1-score of our model's SATD prediction.

Finally, we analyzed the overlap between the set of key phrases in a comment extracted by our CNN model and the set of key phrases in the comment identified by each participant. We consider the two set of key phrases as two bags of words and compute the Jaccard coefficient to measure their similarity. This helps us understand the similarity of our CNN-extracted key phrases and the

Table 25. Precision, Recall, and F1-score of User-Study with Majority Strategy

Target	Precision	Recall	F1-score
Guava	0.900	0.818	0.857
RxJava	0.933	0.875	0.903
Commons-lang	0.786	0.846	0.815
Average	0.873	0.846	0.860

Table 26. Precision, Recall, and F1-score of User-Study with All-agree Strategy

Target	Precision	Recall	F1-score
Guava	0.600	0.857	0.706
RxJava	0.533	0.800	0.640
Commons-lang	0.556	0.833	0.667
Average	0.563	0.830	0.671

human-identified key phrases for SATD classification. The higher the similarity, the more intuitive and explainable our CNN-extracted key phrases.

5.8.3 Result. The Fleiss Kappa coefficient of the five participants' SATD/non-SATD labels is 0.652, which indicates a substantial agreement between the participants' labels. The Fleiss Kappa coefficient of our model and the five participants' SATD/non-SATD labels is 0.683, which is also a substantial agreement. Tables 25 and 26 show the precision, recall and F1-score of our model's SATD prediction against the ground truth obtained with the majority and all-agree strategy, respectively. The results show that for the SATD comments that at least three human annotators agree, our model achieves very high precision and recall, with the average F1-score 0.861 across the three projects. Even with a much stricter ground-truth SATD comments, our model still achieves very high recall. Of course, precision in this strict setting is low as expected, but it is still much better random guess, because the percentage of SATD comments are imbalanced.

By comparing Tables 25 and 26, it can be seen that the results with the majority-win versus all-agree strategy has large gap. To understand the underlying reasons, we collected feedback from the study participants about their own opinions on what should or should not be highlighted as SATD patterns. Participants suggest that they generally look for clear SATD indicators such as "todo" and "fixme." When such clear indicators are present in the code comments, all participants agree that the code comments are SATDs. However, when such clear indicators are not present in the code comments (e.g., "there is a need to check here"), there are often disagreements between participants. In 34 (22.67%) of such cases, the majority votes align with the model's SATD prediction. Our manual analysis suggests that our model is correct for 32 cases and is wrong for 2 cases. For the remaining 116 cases, the majority votes do not align with the model's prediction. For example, the two participants annotated "there is a need to check here" as non-SATD, while our model predicts it as SATD. Depending on the comment semantics, we believe our model's prediction is correct.

Table 27 shows the average key-phrases Jaccard coefficient for the 50 comments of each project between each participant and our model. We can see that 14 Jaccard coefficient metrics are in the range of 0.64 to 0.72, which indicates that the key phrases extracted by our model and those

Table 27. Key Phrases Jaccard Coefficient

	Guava	RxJava	Commons-lang	Average
Participant-A	57.64%	63.91%	65.16%	62.24%
Participant-B	68.92%	70.33%	71.33%	70.19%
Participant-C	66.35%	68.52%	72.20%	69.02%
Participant-D	71.36%	69.80%	70.65%	70.60%
Participant-E	65.48%	68.79%	69.49%	67.92%
Average	65.95%	68.27%	69.77%	68.00%

Table 28. Average Time of Model Training and Application

Approach	Training time (s)	Application time (s)
Text Mining	19.632	0.038
Simple CNN	1185.744	1.026
Our Model	3548.593	1.150

identified by human are similar. The dissimilar parts are mainly because human tends to highlight a continuous sequence of words, such as "FIXME: this is probably not very efficient," while our CNN usually focuses on only the relevant phrases such as "FIXME" and "probably not very efficient" but not the phrases like stop words "this is."

Our user study confirms the high performance of our approach for SATD predication. *It also confirms that the key phrases extracted by our model largely match those that humans identify for SATD prediction.*

6 DISCUSSION

Our approach outperforms the text-mining approaches for SATD classification [10]. More importantly, it demonstrates explainability of the prediction results that used to require significant human effort in manually examining the data. Furthermore, the generalizability and adaptability of our approach makes it easily deployable in practice. The explainability and deployability are largely overlooked in the existing study of machine-learning-based SATD classification. In this section, we discuss about time efficiency of our approach as well as the threats to validity of our study.

6.1 Time Efficiency

Table 28 presents the average model training and inference time across the 10 projects. We can see that the model training and inference time of our approach is reasonable. On average, we need about one hour to train our SATD prediction model, and the trained model takes about 1.6s to predict the label of a code comment and highlight SATD patterns of the comment in the testing dataset. Note that the model does not need to be updated all the time and it can be used to label many code comments. Our model training time is longer than that of Kim's simple CNN method and Huang et al.'s traditional text-mining approach. Our model inference time is longer than those of the other approaches such as References [10, 12], but we believe it is still acceptable (once the model has been trained, it can quickly label and highlight patterns in code comments in seconds).

6.2 Threats to Validity

Threats to internal validity relate to errors in our implementation and personal bias in data labeling and user studies. To avoid implementation errors, we have carefully reviewed our implementation and its hyperparameter settings. For personal bias in manual classification of code comments, the authors of Reference [24] used a statistically significant sample of classified comments and invited an independent annotator, who is not an author of their paper to manually classify code comments. They reported a high level of agreement between the classification given by different human annotators (Cohen's Kappa coefficient of 0.81). This gives us high confidence in the dataset used in our article. For a project data in the Maldonado dataset, we consider as SATD comments only the manually tagged lines that are not "WITHOUT_CLASSIFICATION." However, we do not use the fine-grained SATD categories proposed in Maldonado et al. [11]. In our work, we consider only the binary classification of code comments, i.e., SATD versus non-SATD. Therefore, design debt, requirement debt, defect debt, documentation debt, and test debt are simply considered SATD in this work. Our RQ3 also involves manual labeling of key phrase relevance, and our RQ8 involves manual classification of code comments and manual identification of key phrases. We perform Cohen's kappa coefficient analysis in RQ3 and RQ8, which indicates substantial inter-rater agreement.

Threats to external validity relate to both the quantity and quality of our experimental dataset and the generalizability of our experiment results and findings. To guarantee the quantity and quality of our dataset, we study 10 open source projects that vary in the number of comments as well as comment characteristics. In total, we have analyzed 62,566 comments.

Since open source communities are highly transparent and developers are usually "forced" to do a lot of communication through source code comments (as they are distributed), they could be more likely to admit technical debt in comments. In contrast, it is possible that developers in a company are hesitant to admit technical debt in the source code, as it might affect the evaluation of their job quality. Therefore, it is still an open question if our approach can be applied to non open-source projects. Furthermore, we use a medium size training and testing dataset in this work. This allows us to perform manual analysis to understand the capability and limitations of our approach. In the future, we will reduce this threat by extending our approach to larger software projects.

Threats to construct validity refers to the suitability of our evaluation metrics. We use precision, recall and F1-score, which are also used by past studies to evaluate the performance of various automated software engineering techniques [33–35]. Thus, we believe there is little threat to construct validity.

7 RELATED WORK

In this article, we propose a deep learning-based method to tackle various challenges of SATD classification of source code comments (see Section 2). Therefore, the related work includes two main parts: code comment analysis and deep learning in software engineering.

7.1 Code Comment Analysis

There are a large number of studies focus on detecting and managing technical debts. For example, many empirical studies use the findings of technical debt in software maintenance [3, 36, 37]. Some studies also investigate the performance differences between automated tools and human for detecting technical debts [38–40]. According to their findings, there is a small overlap between automated tools and human. Automated tools are more efficient in finding defect-related debts, while human can realize more abstract categories of technical debts.

Potdar and Shihab [6] proposed the concept of self-admitted technical debt (SATD), which considers debt that is intentionally introduced (e.g., temporary fixes or work-around) by developers

and explicitly recorded in source code comments. In this work, Potdar and Shihab manually summarized 62 patterns that can be used to identify SATD comments, after reading more than 100,000 source-code comments from different Java projects. Based on this first work, Wehaibi et al. [7] examined the relation between self-admitted technical debts and defects. They found that SATD is not related to defects, rather making the system more difficult to change in the future. In addition, Maldonado and Shihab [24] further divided SATD into five types, namely design debt, defect debt, documentation debt, requirement debt, and test debt. The most similar work of our work is the traditional text-mining-based method for SATD classification proposed by Huang et al. [10]. In their work, they utilize feature selection to select useful features for classifier training and combine multiple classifiers from different source projects to build a composite classifier that identifies SATD comments in a target project.

Further more, Maldonado et al. [11] proposed an approach to auto-matically identify the two most common types of SATD comments (i.e., design debt and requirement debt). In their work, they built a maximum entropy classifier based on natural language processing (NLP). They use the same dataset and same experiment setting as that of Huang et al. [10]. Huang et al. also make comparison with Maldonado et al. [11] by following their work to do basic preprocessing (i.e., stemming and removing punctuation characters) and build the maximum entropy classifier to predict whether a comment contains SATD or not. Note that compared with Huang et al. [10], Maldonado et al. [11] did not use feature selection or ensemble learning in their approach, and thus the approach by Huang et al. [10] outperforms that of Maldonado et al. [10].

From a serious of experiments, it can be found that our experiments in RQ3 and RQ4 and our user study in RQ8 show that our CNN-based approach can learn effective SATD features for SATD classification and the CNN-learned features correspond to intuitive SATD patterns that human identifies. Observing the CNN-extracted features, we find many characteristics of SATD comments, including common features (e.g., "todo," "fixme," "hack," "xxx"), project-unique features (e.g., "revisit" in EMF), as well as negative sentiment words (e.g., "ugly," "stupid," "negative," "evil"). Although human is able to identify frequent SATD features, many project-unique or less-evident features are hard to identify by just human observation. For example, Potdar and Shihab manually summarized only 62 SATD patterns, after reading more than 100,000 source-code comments from four Java projects, while our approach identifies 700 SATD patterns with much diverse vocabulary in the 10 projects.

Our work suggests that the automatic feature learning by a deep learning method can mitigate very well the limitations of human observation. This opens up opportunities for deeper understanding of SATD practices in software development and its impact on software evolution. For example, in this work we exploit the CNN-based SATD patterns to compare the characteristics of SATD practices in different software projects, which in turn helps explain why our approach performs well or does not perform well in different training and testing settings. This explainability helps turn our experiment results in RQ5, RQ6, and RQ7 into practical guideline for effective model training and adaptation depending on the SATD characteristics of software projects.

Some previous work on source code comments focused on investigating the relationship between comments and code [41]. For example, Tan et al. [42] proposed a tool called iComment that automatically analyzes comments written in natural language to extract implicit program rules, and then they use these rules to automatically detect inconsistencies between comments and source code, indicating either bugs or bad comments. In their follow-up work, they studied the inconsistencies between Javadoc comments and method bodies [43]. Malik et al. presented a large empirical study to better understand the rationale for updating comments, and they used the Random Forests algorithm to accurately predict the likelihood of a comment being updated [44].

Fluri et al. proposed an approach to map code and comments to observe their co-evolution over multiple versions [45].

Other work focused on using comments to assist in software development and maintenance. For example, Khamis proposed an automatic approach for assessing the quality of inline documentation using a set of heuristics, targeting both the quality of language and the consistency between source code and its comments [46]. Padioleau et al. [47] studied 1,050 comments randomly sampled from Linux, FreeBSD, and OpenSolaris and found that 52.6% of these comments could be leveraged for improving reliability. Storey et al. investigated how task annotations can be used to support a variety of activities fundamental to articulation work within software development. They found that the use of task annotations varies from individuals to teams, and if incorrectly managed, they could negatively impact the maintenance of a system [48].

7.2 Deep Learning in Software Engineering

Recently, a number of studies explored deep learning techniques for software engineering tasks, such as bug localization [49, 50], defect prediction [51], software community question retrieval [52, 53], and so on. When using deep learning to deal with software text, it is necessary to find an appropriate sentence representation. Traditional text representation usually uses vector space model (VSM), which was proposed by Jacobs and Zernik [54]. In VSM, text is viewed as a set of feature items. It uses weighted feature terms to construct a vector for text representation and uses word frequency information to weight text features. However, this representation suffers from lexical gaps in text [55]. To address this issue, there has been a surge of work proposing to represent words as dense vectors, derived using neural-network language modeling [56, 57]. These representations are called "word embedding." Our model uses word embeddings as the input that can better model word semantics [58, 59]. Word embeddings have also been used to improve document retrieval in software engineering [60]. In addition, Nguyen et al. [61] proposed API embedding method to capture the regularities of the relations of APIs in API usages.

In the NLP community, the development of CNN architectures for sentence-level and document-level text processing is under intensive research. Some recent work utilizes CNN to learn the semantic relations between two pieces of texts. For example, Kim [12] proposed a CNN trained on top of word embeddings [62] and then applied the CNN to sentence classification. Xu et al. [52] adopt word embeddings and CNN to capture word- and document-level semantics of knowledge units. Chen et al. [18] proposed a novel cross-lingual question retrieval based on word embeddings and CNN. Chen et al. [63] identified software-specific terms by contrasting software-specific and general corpuses and inferred morphological forms of software-specific terms by combining distributed word semantics, domain-specific lexical rules and transformations, and graph analysis of morphological relations.

Training CNN requires initial setting of multiple hyperparameters (e.g., the number of filters). Previous studies [27, 64] have shown that hyperparameter tuning is important for a prediction model to achieve better performance. Zhang and Wallace [28] conducted a sensitivity analysis of (and wrote a practitioners' guide to) CNN for sentence classification [50]. They found that the dimension size of word embedding, the number of filters, and the combination of filter heights are the most important hyperparameters that have a large effect on performance and should be tuned.

To sum up, our work is different from all the work mentioned above as follows:

- From the aspect of method, our model uses CNN for SATD classification and make automatic hyperparameters tuning referring to the guidance of Zhang and Wallace [28]. In other words, Kim [12] and Zhang and Wallance [28] inspire the design of our approach and experiments in this work. However, our work is not a simple adoption of these two works

on software engineering data. Instead, our work contains considerable new technical and experimental contributions, which distinguishes our work from these two references. Furthermore, we develop a backtracking mechanism to understand and explain the CNN's results in text classification tasks. To the best of our knowledge, little work has been done for such understanding and explanation of CNN results in previous studies.

Moreover, many approaches relying on machine-learning methods for software engineering tasks (not limited to SATD classification) often overlook the availability of training data and the characteristics drift between the training data and the testing data. Unfortunately, these two factors significantly affect the practical use of the proposed methods. If the effort to prepare the training data is high, for example, then the rich training datasetting in RQ6 needs to label nine projects' comment for obtaining a good prediction model for one project, then it becomes less practical to deploy the approach even it has good performance. However, if the data of a project are very limited, for example, Apache Ant and EMF, then it would be difficult to use this project's data alone to produce a high-quality model. Furthermore, if the data characteristics of a source project used to train the model differ from those of the target project to deploy the trained model (see analysis in Section 5.4), then the model performance will inevitably degrade. All of these scenarios should be carefully investigated when designing and evaluating machine-learning-based methods for software engineering tasks. Motivated by these scenarios, our work conducts extensive experiments in RQ6 and RQ7 on the model's performance, generalizability, and adaptability in rich and limited training datasettings and across projects with similar or different SATD characteristics. With the knowledge of the model's generalizability and adaptability, users would be able to deploy our approach more effectively depending on the SATD characteristics of software projects.

- From the aspect of task, it provides a foundational technique for the effective management of SATDs that are highly related to the software quality. Our work also contributes new actionable knowledge to the research and practice on SATD detection in particular and data mining for software engineering in general.

At the surface, both code comments and Twitter messages are short text. But code comments are a core information source for documenting software knowledge and a fundamental input to various software engineering tasks, such as code search, fault prediction, and bug localization. In this work, we focus on a particular type of software development knowledge embedded in code comments, i.e., self-admitted technical debt (SATDs). Empirical studies [6, 7] confirms that "although the percentage of SATD in a project is not high, it can have a negative impact on software complexity. More concretely, they found that source code files that contain SATD have more bug-fixing changes, while files without SATD have more defects." Therefore, making SATDs explicit and managing them proactively is an important quality issue in the lifecycle of a software system.

Imagine a developer introduced a SATD in the system and then left the project. This SATD could become a "hidden bomb" due to this developer turnover and the growth of the system complexity. Without knowing the presence of this hidden SATD, the project team or the new comers in the team may have to pay a high interest later. As another example, for an open source project where its contributors come from different knowledge and linguistic background, making its SATD key phrases explicit would help the project team detect "unexpected" use of certain SATD key phrases or enforce the norm of SATD descriptions. Furthermore, making the hidden SATD patterns explicit can help developers learn better SATD annotation practices from other projects. However, the above-mentioned effective management of SATDs in the software life cycle is impossible without an effective

technique to make SATDs explicit. Our work contributes such a reliable and explainable SATD detection technique.

Technically, our approach uses a CNN-based model for SATD detection. The approach is lightweight (no need for program analysis). Not only does it lead to superior performance than traditional text-mining approach, but it also opens up the door to investigate the model's explanability and adaptability. These the two aspects have been largely overlooked in the existing research on data mining for software engineering that focus on only the model performance. Our work suggests that the automatic feature learning by a deep learning method can mitigate very well the limitations of human observation of software data. This will open up the door for the community to investigate the explainable AI for assisting software data analysis tasks. Furthermore, many approaches relying on machine-learning methods for software engineering tasks (not limited to SATD classification) assume the availability of training data and overlook the characteristics drift between the training data and the testing data. Our experiments show that these two factors significantly affect the practical use of the proposed methods. Our study points out that researchers and practitioners should carefully investigate the model's generalizability and adaptability when designing and evaluating machine learning-based methods for software data. The above insights from our study contribute new actionable knowledge in software data analytics.

8 CONCLUSION

In this article, we identify two key challenges in machine-learning-based SATD classification, i.e., explainability and deployability. The deployability challenge is rooted in the vocabulary diversity, project uniqueness, variable length, and semantic variations of SATD features in code comments, while the explainability challenge lies in the difficulty of deciphering the correspondence between the features that a machine-learning method uses for prediction and its prediction results. With the objective of tackling these two challenges, we design a CNN-based approach for SATD classification with a holistic consideration of model performance, deployability, and explainability.

Our evaluation shows that our CNN-based approach can effectively learn to extract variable-length text features in code comments for SATD classification. Relying on this feature learning capability, our approach can extract intuitive SATD features and patterns that align well with human-identified features and patterns. However, the SATD patterns that our approach automatically extracts from the comment data are more comprehensive and with a more diverse vocabulary, which is hard to summarize just by human observation. Based on the extracted SATD features and patterns, our approach can provide a good overview of the SATD characteristics of software projects and also provide an intuitive explanation of the model's prediction results. In addition to the superior explainability that traditional text-mining-based method lacks, our approach also demonstrates superior performance for both within-project and cross-project prediction. Our CNN model can learn much generalizable features and is much easier to adapt, especially in the limited training datasetting for cross-project prediction. The generalizability and adaptability of our approach make it easy to deploy a high-quality SATD classification model for multiple projects.

In the future, we will enhance our classification model with more comment data. With a richer dataset, we expect that more SATD features and patterns will be extracted. It is also interesting to further investigate the correlations between the extracted SATD features and patterns and other project properties, e.g., defects and maintenance cost. Additionally, we believe that the identification of prominent key phrases used in SATDs would also be useful for practitioners. For example, for an open source project, making its SATD key phrases explicit would help the project detect "unexpected" use of certain SATD key phrases or enforce the norm of SATD descriptions. Furthermore, developers may learn better SATD annotation practices from other projects. In this work,

we simply put the mined SATD key phrases in a long list for the purpose of explaining our model's prediction. How to make better use of the mined SATD key phrases within or across projects for practitioners is beyond the scope of this article. We leave it as part of our future work.

REFERENCES

[1] Ward Cunningham. 1993. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger* 4, 2 (1993), 29–30.

[2] Nico Zazworka, Michele A Shaw, Forrest Shull, and Carolyn Seaman. 2011. Investigating the impact of design debt on software quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 17–23.

[3] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, et al. 2010. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. ACM, 47–52.

[4] Yuepu Guo and Carolyn Seaman. 2011. A portfolio approach to technical debt management. In *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 31–34.

[5] Chris Sterling. 2010. *Managing Software Debt: Building for Inevitable Change (Adobe Reader)*. Addison-Wesley Professional, New York, NY.

[6] Aniket Potdar and Emad Shihab. 2014. An exploratory study on self-admitted technical debt. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*. IEEE, 91–100.

[7] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. 2016. Examining the impact of self-admitted technical debt on software quality. In *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, Vol. 1. IEEE, 179–188.

[8] Radu Marinescu. 2004. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance 2004*. IEEE, 350–359.

[9] Radu Marinescu, George Ganea, and Ioana Verebi. 2010. Incode: Continuous quality assessment and improvement. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering (CSMR'10)*. IEEE, 274–275.

[10] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. 2018. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering* 23, 1 (2018), 418–451.

[11] Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Trans. Softw. Eng.* 43, 11 (2017), 1044–1062.

[12] Yoon Kim. 2014. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP'14)*. Association for Computational Linguistics, 1746–1751. https://www.aclweb.org/anthology/D14-1181.

[13] Duyu Tang, Furu Wei, Nan Yang, Ming Zhou, Ting Liu, and Bing Qin. 2014. Learning sentiment-specific word embedding for twitter sentiment classification. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, Vol. 1. 1555–1565.

[14] Siwei Lai, Kang Liu, Shizhu He, and Jun Zhao. 2016. How to generate a good word embedding. *IEEE Intell. Syst.* 31, 6 (2016), 5–14.

[15] 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.

[16] Michael Kampffmeyer, Arnt-Børre Salberg, and Robert Jenssen. 2016. Semantic segmentation of small objects and modeling of uncertainty in urban remote sensing images using deep convolutional neural networks. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW'16)*. IEEE, 680–688.

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

[18] Guibin Chen, Chunyang Chen, Zhenchang Xing, and Bowen Xu. 2016. Learning a dual-language vector space for domain-specific cross-lingual question retrieval. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 744–755.

[19] Fabrizio Sebastiani. 2002. Machine learning in automated text categorization. *ACM Comput. Surv.* 34, 1 (2002), 1–47.

[20] Andrew McCallum, Kamal Nigam, et al. 1998. A comparison of event models for naive bayes text classification. In *Proceedings of the AAAI-98 Workshop on Learning for Text Categorization*, Vol. 752. Citeseer, 41–48.

[21] Simon Tong and Daphne Koller. 2001. Support vector machine active learning with applications to text classification. *J. Mach. Learn. Res.* 2, (Nov. 2001), 45–66.

[22] Eui-Hong Sam Han, George Karypis, and Vipin Kumar. 2001. Text categorization using weight adjusted k-nearest neighbor classification. In *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, Berlin, 53–65.

[23] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2008. JDeodorant: Identification and removal of type-checking bad smells. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering 2008 (CSMR'08)*. IEEE, 329–331.

[24] Everton da S. Maldonado and Emad Shihab. 2015. Detecting and quantifying different types of self-admitted technical debt. In *Proceedings of the IEEE 7th International Workshop on Managing Technical Debt (MTD'15)*. IEEE, 9–15.

[25] Jacob Cohen. 1968. Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit.*Psychol. Bull.* 70, 4 (1968), 213.

[26] Haibo He and Edwardo A. Garcia. 2009. Learning from imbalanced data. *IEEE Trans. Knowl. Data Eng.* 21, 9 (2009), 1263–1284.

[27] Wei Fu, Tim Menzies, and Xipeng Shen. 2016. Tuning for software analytics: Is it really necessary? *Inf. Softw. Technol.* 76 (2016), 135–146.

[28] Ye Zhang and Byron Wallace. 2015. A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. *arXiv preprint arXiv:1510.03820* (2015).

[29] Gerhard Wohlgenannt and Filip Minic. 2016. Using word2vec to build a simple ontology learning system. In *Proceedings of the International Semantic Web Conference (Posters & Demos)*.

[30] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Biometr. Bull.* 1, 6 (1945), 80–83.

[31] Simon S. Haykin, Simon S. Haykin, Simon S. Haykin, and Simon S. Haykin. 2009. *Neural Networks and Learning Machines*. Vol. 3. Pearsons, Upper Saddle River, NJ.

[32] Joseph L. Fleiss. 1971. Measuring nominal scale agreement among many raters.*Psychol. Bull.* 76, 5 (1971), 378.

[33] Erik Arisholm, Lionel C. Briand, and Magnus Fuglerud. 2007. Data mining techniques for building fault-proneness models in telecom java software. In *Proceedings of the 18th IEEE International Symposium on Software Reliability 2007 (ISSRE'07)*. IEEE, 215–224.

[34] Foyzur Rahman, Daryl Posnett, and Premkumar Devanbu. 2012. Recalling the imprecision of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 61.

[35] Tian Jiang, Lin Tan, and Sunghun Kim. 2013. Personalized defect prediction. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 279–289.

[36] Carolyn Seaman, Yuepu Guo, Clemente Izurieta, Yuanfang Cai, Nico Zazworka, Forrest Shull, and Antonio Vetrò. 2012. Using technical debt data in decision making: Potential decision approaches. In *Proceedings of the 3rd International Workshop on Managing Technical Debt*. IEEE Press, 45–48.

[37] Philippe Kruchten, Robert L Nord, Ipek Ozkaya, and Davide Falessi. 2013. Technical debt: Towards a crisper definition report on the 4th international workshop on managing technical debt. *ACM SIGSOFT Softw. Eng. Not.* 38, 5 (2013), 51–54.

[38] Erin Lim, Nitin Taksande, and Carolyn Seaman. 2012. A balancing act: What software practitioners have to say about technical debt. *IEEE Softw.* 29, 6 (2012), 22–27.

[39] Nico Zazworka, Rodrigo O Spínola, Antonio Vetro, Forrest Shull, and Carolyn Seaman. 2013. A case study on effectively identifying technical debt. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 42–47.

[40] Gabriele Bavota and Barbara Russo. 2016. A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR'16)*. IEEE, 315–326.

[41] Andrian Marcus and Jonathan I. Maletic. 2003. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*. IEEE, 125–135.

[42] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: Bugs or bad comments?* In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 145–158.

[43] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *Proceedings of the 2012 IEEE 5th International Conference on Software Testing, Verification and Validation (ICST'12)*. IEEE, 260–269.

[44] Haroon Malik, Istehad Chowdhury, Hsiao-Ming Tsou, Zhen Ming Jiang, and Ahmed E. Hassan. 2008. Understanding the rationale for updating a function's comment. In *Proceedings of the IEEE International Conference on Software Maintenance 2008 (ICSM'08)*. IEEE, 167–176.

[45] Beat Fluri, Michael Wursch, and Harald C. Gall. 2007. Do code and comments co-evolve? on the relation between source code and comment changes. In *Proceedings of the 14th Working Conference on Reverse Engineering 2007 (WCRE'07)*. IEEE, 70–79.

[46] Ninus Khamis, René Witte, and Juergen Rilling. 2010. Automatic quality assessment of source code comments: The JavadocMiner. In *Proceedings of the International Conference on Application of Natural Language to Information Systems*. Springer, Berlin, 68–79.

[47] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. 2009. Listening to programmers taxonomies and characteristics of comments in operating system code. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 331–341.
[48] Margaret-Anne Storey, Jody Ryall, R. Ian Bull, Del Myers, and Janice Singer. 2008. TODO or to bug. In *Proceedings of the ACM/IEEE 30th International Conference on Software Engineering 2008 (ICSE'08)*. IEEE, 251–260.
[49] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 631–642.
[50] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. 2010. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Volume 1*. ACM, 45–54.
[51] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech recognition with deep recurrent neural networks. In *Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'13)*. IEEE, 6645–6649.
[52] Bowen Xu, Deheng Ye, Zhenchang Xing, Xin Xia, Guibin Chen, and Shanping Li. 2016. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 51–62.
[53] Lin Ma, Zhengdong Lu, and Hang Li. 2016. Learning to answer questions from image using convolutional neural network. In *AAAI*, Vol. 3. 16.
[54] Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A vector space model for automatic indexing. *Commun. ACM* 18, 11 (1975), 613–620.
[55] Paul S. Jacobs and Uri Zernik. 1988. Acquiring lexical knowledge from text: A case study. In *AAAI*, Vol. 88. 739–744.
[56] Yoshua Bengio, Rejean Ducharme, Pascal Vincent, and Christian Janvin. 2003. A neural probabilistic language model. *J. Mach. Learn. Res.* 3, 6 (2003), 1137–1155.
[57] Ronan Collobert and Jason Weston. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*. ACM, 160–167.
[58] Joseph Turian, Lev Ratinov, and Yoshua Bengio. 2010. Word representations: A simple and general method for semi-supervised learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 384–394.
[59] Ruiji Fu, Jiang Guo, Bing Qin, Wanxiang Che, Haifeng Wang, and Ting Liu. 2014. Learning semantic hierarchies via word embeddings. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, Vol. 1. 1199–1209.
[60] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. 2016. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 404–415.
[61] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. 2017. Exploring API embedding for API usages and applications. In *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE'17)*. IEEE, 438–449.
[62] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
[63] Chunyang Chen, Zhenchang Xing, and Ximing Wang. 2017. Unsupervised software-specific morphological forms inference from informal discussions. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 450–461.
[64] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. Using bad learners to find good configurations. *arXiv preprint arXiv:1702.05701* (2017).

Received June 2018; revised March 2019; accepted March 2019