Automatically learning patterns for self-admitted technical debt removal

Citation for published version (APA):

Document status and date:
Published: 06/02/2020

Document Version:
Accepted manuscript including changes made at the peer-review stage

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
Automatically Learning Patterns for Self-Admitted Technical Debt Removal

Fiorella Zampetti  
University of Sannio  
Italy  
fiorella.zampetti@unisannio.it

Alexander Serebrenik  
Eindhoven University of Technology  
The Netherlands  
a.serebrenik@tue.nl

Massimiliano Di Penta  
University of Sannio  
Italy  
dipenta@unisannio.it

Abstract—Technical Debt (TD) expresses the need for improvements in a software system, e.g., to its source code or architecture. In certain circumstances, developers “self-admit” technical debt (SATD) in their source code comments. Previous studies investigate when SATD is admitted, and what changes developers perform to remove it. Building on these studies, we present a first step towards the automated recommendation of SATD removal strategies. By leveraging a curated dataset of SATD removal patterns, we build a multi-level classifier capable of recommending six SATD removal strategies, e.g., changing API calls, conditionals, method signatures, exception handling, return statements, or telling that a more complex change is needed. SARDELE (SATD Removal using DEep LEarning) combines a convolutional neural network trained on embeddings extracted from the SATD comments with a recurrent neural network trained on embeddings extracted from the SATD-affected source code. Our evaluation reveals that SARDELE is able to predict the type of change to be applied with an average precision of ≃ 55%, recall of ≃ 57%, and AUC of 0.73, reaching up to 73% precision, 63% recall, and 0.74 AUC for certain categories such as changes to method calls. Overall, results suggest that SATD removal follows recurrent patterns and indicate the feasibility of supporting developers in this task with automated recommenders.

Index Terms—Self-Admitted Technical Debt; Recommender Systems; Deep Learning; Neural Networks

I. INTRODUCTION

Technical Debt (TD) refers, according to a definition provided by Cunningham [8], to “not quite right code which we postpone making it right”. The reasons why TD occurs in software projects are many-fold: deadline pressure, e.g., the need for releasing a new feature or a bug fix, incapability to produce a suitable solution for a given development problem or, lack of a suitable component that solves a given task.

Keeping track of TD has, for developers, paramount importance for its management and removal [13]. To this aim, developers specify or “admit” the presence of TD by adding a comment near the TD-affected source code [40]. In such cases, the TD is considered as a “self-admitted” TD (SATD).

As SATD represents an admitted manifestation of a negative phenomenon (TD), it is of particular interest to understand whether this admission leads to appropriate corrective action, i.e., SATD removal. Bavota and Russo [5] found that over half (57%) of the SATD is actually being removed. While 63% of the removals are done by the same developer who admitted the SATD, in the remaining cases the SATD is addressed by somebody else. This may be especially true in projects with a very high turnover [25], [42], [50], and may make the removal quite challenging.

Concerning “how” SATD is being removed, Wehaibi et al. [52] found that SATD may lead, in general, towards complex software changes, while Maldonado et al. [9] conducted a survey on SATD removal and found that this happens either in the context of bug fixing, or feature addition. Starting from the results of Maldonado et al. [9], Zampetti et al. [58] analyzed the change patterns that lead towards SATD removal, by combining an automated analysis through GumTree [14] with manual analysis. Their study produced a taxonomy of SATD removal strategies, including complex changes (for different projects 40–55% of the removals belong to this category), but, also, recurring specific changes, such as changes in conditionals (11–29%), exception handling, method calls (e.g., API) changes, or method signatures changes. Regularities in program changes have also been found in previous work categorizing bug fixing patterns [38], and in recent work learning program repairs from existing source code [47].

Based on the results of previous studies, we conclude that SATD removal is often a necessity, frequently performed by the developer who did not introduce it, and in about half of the cases, it follows specific patterns.

We start from the observation of Zampetti et al. [58], and from the curated dataset they made available. We propose an automated approach, named SARDELE (SATD Removal using DEep LEarning), to recommend how SATD should be removed. That is, given a SATD occurring in the systems’ source code, we recommend one of the six removal strategies proposed by Zampetti et al. [58]. While this does not provide the concrete removal solution yet, we claim that it can help developers to better plan and ponder SATD removal solutions.

SARDELE is based on the conjecture that the SATD comment and the affected source code contain enough elements to determine the SATD removal strategy (i.e., category). Therefore, we propose an approach that combines two deep neural networks: (i) a convolutional neural network (CNN) trained on embeddings extracted from the SATD comments, and (ii) a recurrent neural network (RNN) trained on embeddings extracted from the SATD-affected source code.
We apply SARDELE on the previously-available dataset [58], featuring 779 manually-classified method-level SATD removals, each one classified according to the six categories. Although we are aware that an approach like the one we propose would surely benefit from a larger dataset, we opted to propose our solution and validate it on a very reliable curated dataset, leaving larger empirical evaluations for future work.

Overall, SARDELE achieves a precision of about 55%, a recall of about 57%, and AUC of 0.73. On the individual categories, the performance varies, also depending on the amount of data in the training corpus for the category, e.g., SARDELE reaches up to 73% precision, 63% recall, and 0.74 AUC for changes to method calls. We also found that the combination of comment-based classifier (CNN) and source code-based classifier (RNN) significantly outperforms the individual classifiers. Finally, SARDELE outperforms a simple machine-learning baseline using Random Forests, and a manually-produced baseline in which annotators guessed SATD removal strategies by looking only at the SATD comment.

To summarize, this work highlights the feasibility of outlining solution directions for SATD removal and paves the ways towards automatic recommenders for SATD removal strategies.

The study dataset is available for replication purposes [57].

II. SATD REMOVAL TAXONOMY AND DATASET

This work supports developers in removing the SATD, by recommending one or more of the six removal strategies that have been identified in a previous study by Zampetti et al. [58]. Starting from a curated dataset of SATD by Maldonado et al. [9], Zampetti et al. first analyzed, using the GumTree [14] fine-grained differencing tool, the commits in which the SATD comment disappeared from the system. Then, they performed a manual analysis aimed at identifying relevant categories of SATD removals. In the end, they identified the following six categories of changes (which include addition/removals of the mentioned items):

1) **Method calls**, e.g., changes to APIs;
2) **Conditional statements**, e.g., addition, removal, or change to preconditions;
3) **Try-catch blocks**, e.g., addition of exception handling for previously-uncought exceptions;
4) **Method signatures**, i.e., either the parameters or the return type of the SATD-affected methods are changed, for example because of a refactoring action1 or another improvement;
5) **Return statements**, e.g., the return object is changed to improve the method functionality/fixing a problem;
6) **Other (more complex)** changes, that could not be classified along the aforementioned dimensions.

The dataset provided by Zampetti et al. provides, for method-level SATD identified (and removed) in five Java open source projects, the following information: (i) commit where the SATD was introduced, along with the SATD-related comment, and method to which the SATD comment was attached, (ii) commit where the SATD was removed, and (iii) removal category, according to the aforementioned taxonomy. Overall, the dataset features a total of 779 SATD removals, as detailed in Table I. Note that, the total number (887) is greater than 779 since the SATD can be removed applying more than one removal strategies. Moreover, the original dataset also contains 912 SATD removals occurred by deleting the whole class or method, or without modifying the SATD-affected source code. We did not contemplate such a case in SARDELE, as it is likely that either the SATD was removed “by chance”, or in the context of a complex class refactoring.

III. APPROACH

Below we describe SARDELE, the proposed approach for automatically recommending SATD removal strategies. Given a method affected by SATD (for which we have the source code and the SATD-related comment), SARDELE suggests one of the six SATD removal categories detailed in Section II.

Fig. 1 provides an overview of SARDELE. As the figure shows, SARDELE consists of two classifiers, one based on the SATD comment, and one based on the source code of the SATD-affected method. We conjecture that both the comment left by the developer who introduced the SATD, and the source code itself contain meaningful elements for identifying the SATD removal strategy.

The top-side of the figure (white blocks) starts with preprocessing the SATD comment. Then, to reduce the dictionary size and capture relationships between adjacent words (i.e.,

![Figure 1: SATD classification approach](image-url)
within a window of words), skip-grams are extracted from the sequence of words, and are seed into a Convolutional Neural Network (CNN). The bottom-side of the figure (gray blocks) starts with extracting tokens from the SATD-affected method source code, and identifying idioms, i.e., frequent literals and identifiers, to retain. Similarly to the previous case, after having produced a stream of tokens/idioms, we extract skip-grams from them. Skip-grams are then seeded into a Recurrent Neural Network (RNN). Finally, the output of the two networks is combined and seed onto a softmax layer, which produces the final classification according to the six SATD removal strategies.

We base our approach on deep neural network classification strategies, rather than on traditional machine learners, e.g., decision trees. While the latter may have advantages in terms of computational cost and capability to explain the predicted classification, properly conveying the semantics behind comments and source code would have required a complex, possibly manual, feature engineering. Therefore, based on previous applications of deep learning to source code analysis [47], [53], we opted for such a solution. Nevertheless, in our empirical evaluation, we compare SARDELE with a simple machine learning-based classifier.

Note that we use different kinds of neural networks for comments and source code because RNN better preserves information related to sequences, particularly important when analyzing source code, as also Tufano et al. did [47]. Instead, following previous work identifying SATD-comments with CNN [41] we used CNN to analyze comments.

In the following, we describe the components of SARDELE.

A. Classifying SATD removal from comments

Comment preprocessing. Comments are preprocessed by (i) removing special characters and digits, (ii) converting to lower case, (iii) removing stop words and applying the Snowball stemmer [39]. We evaluated SARDELE with and without stemming and stop words removal, and observed the best performance with stemming and without stop words removal, likely because stop words such as “not” might convey useful information.

Extracting skip-grams from comments. Next, we could weight terms using a suitable weighting scheme, e.g., tf-idf, and represent documents as vectors of independent words [2]. However, this approach produces very sparse vectors and the word independence assumption is strong and not realistic.

To overcome the above problems, we use neural language models, i.e., word embeddings, that generate a low-dimensional, distributed embedding of words [7], [48]. Previous studies have shown that neural language models are able to capture both semantic and syntactic relationships between words [31], [32]. In this work, we use WORD2VEC [31]–[33], a well-known unsupervised word embedding approach, able to learn word representations exploiting the context in which the words appear. Specifically, we use a continuous skip-gram model [31], [32] aimed at learning the word embedding of a central word (i.e., \( w_i \)) that is good at predicting the surrounding words in a specific context window.

To train WORD2VEC, we evaluated two different strategies, i.e., training the model from all SATD comments, or relying on the pre-trained model obtained from three million unique Google sentences. Since SATD comments are mainly composed of natural language, we did not observe differences between the two training strategies.

As output, we obtain a dictionary of words in which each word has its vector representation. Hence, each SATD comment, after preprocessing, is represented as a concatenation of the word embeddings of the words included in the comment.

Classifying comments through a Convolutional Neural Network. A Convolutional Neural Network (CNN) consists of interconnected neurons organized in an input layer, one or more hidden layers, and an output layer. Each convolutional layer applies a convolving filter to local features in the network. CNN models have been successfully applied to NLP tasks. Kim [18] reports that, in NLP text classification tasks, CNN build on top of WORD2VEC can achieve good performance with very little hyperparameter tuning. Previous studies have reported that CNN can capture long-range dependencies and learn to correspond to the internal syntactic structure of sentences, hence reducing the noise [18], [55].

Our CNN takes as input the skip-grams produced in the previous step, and tries to capture the most informative word features for classifying the type of change to be applied to the SATD-affected method to address the SATD comment. Calibration of parameters of the hidden layer(s) is discussed in Section IV-A.

Each convolutional layer is followed by a non-linear activation function applied element-wise to the output of the convolution operations. The output of the activation function is then passed to a pooling layer to reduce the size of the data with some local aggregation function. Our pooling layer works on every filter involved in the CNN. More in detail, we choose to adopt the max pooling operation that maps the feature map to a single value based on the maximum value inside each feature map.

The CNN has been trained to minimize the multi-class cross-entropy loss function [28] that computes the distance between the model’s expected output distribution and the actual one. We use a back-propagation algorithm to compute the gradients and Adam optimizer [19] to update the network parameters and add the \( L_2 \)-norm regularization loss to avoid model overfitting.

B. Classifying SATD removals from the source code

Source code preprocessing. Given each SATD-affected method, we first extract its tokens using the tokenizer of GumTree [14]. To preserve API and keyword naming, we did not apply stop word removal, nor stemming. When indexing source code tokens, there could be two extreme solutions. The first one is to treat each symbol, programming language keyword, identifier, or literal as a different dictionary term. This has the advantage to retain all elements contained in the source code, e.g., types, method names, and literals. However,
this would create a very sparse set of features and, ultimately, introduce noise. The second approach would be to replace identifiers and tokens with placeholders. While this reduces the dictionary size, it limits the capability of the approach to learning specific features from the source code, e.g., the presence of certain method calls, certain values in a condition, etc.

A “middle ground” indexing strategy, also adopted in previous work on learning features from source code [47], retains identifiers and literals that appear in the source code body very frequently, with the assumption that they can be useful to learn meaningful patterns (those have been previously referred to as *idioms*). For instance, it could be possible that integer literals such as 0, 1, -1, or identifiers such as `size`, `length` occur very frequently and should be retained. Based on the advantages and disadvantages of the three indexing strategies, we opt for the “middle ground” and retain the original text for literals and identifiers that are outliers in the frequency distribution.

**Extracting skip-grams from source code.** Similarly to Section III-A, we extract skip-grams from the source code tokens using WORD2VEC. In this case, WORD2VEC has been trained on the source code of the SATD methods part of our training set (see Section IV).

**Classifying source code using a Recurrent Neural Network (RNN).** Recurrent neural networks (RNNs) have a self-connected hidden layer. The basic idea is that each new element in the sequence contributes with some new information, and updates the current state of the model (see the loop-back arrows in Fig. 1). Hence, the network output depends on the current input and on the network state. We choose to use a kind of RNN with the capability of learning long-term dependencies, namely Long Short Term Memory (LSTM) to represent each RNN layer.

The RNN architecture is composed of an input layer, followed by several hidden layers (we vary the number of hidden layers but also the number of neurons in each hidden layer to identify the best configuration for our RNN network). The output produced by the LSTM cells in the last hidden layer is passed to a projection layer which generates the features that will be used by the output layer.

The training strategy of the RNN is similar to the one described above for the CNN: cross-entropy minimization, gradient computation through back-propagation, parameter update using the Adam optimizer [19], and overfitting avoidance through $L2$-norm regularization.

**Combining the two networks to classify SATD removal.**

To produce the desired classification, the output of the pooling layer of the CNN and the output of the projection layer of the RNN is passed to a fully connected softmax layer that evaluates the probability distribution over the class labels (as reported in Equation 1), where $W_s$ and $b_s$ are respectively the weight vector and the bias of the softmax classifier.

$$p(y = j | x_{\text{pooling}}; W_s, b_s) = \text{softmax}_j(W_s \cdot x_{\text{pooling}} + b_s) \quad (1)$$

In $RQ_1$ we also evaluate SARDELE when using CNN or RNN only. In such a case, the softmax layer receives inputs solely from the pooling layer of the CNN or from the projection layer of the RNN.

**IV. STUDY DESIGN**

The **goal** of the study is to evaluate SARDELE, assessing its capability to recommend SATD removal categories (hereby referred to as “SATD removals”). The context, described in Section II, consists of 779 SATD removals belonging to five Java open source projects. The study aims at addressing the following research questions:

- **RQ1**: How does SARDELE perform, compared to simple machine-learning baseline? Since SARDELE uses deep neural networks, it is possible that such an expensive approach is not beneficial if compared to simple Machine Learners (MLs). Therefore, we compare SARDELE with ML-based classifiers using Random Forests.

- **RQ2**: How does SARDELE perform, compared to simple machine-learning baseline? If the SATD-related comment already provides enough hints to the developer for removing the SATD, then SARDELE would not be very useful. In this research question, we assess to what extent this happens in our dataset.

In the following, we first discuss the calibration of SARDELE. Then, we describe the metrics used to evaluate the performance of the approaches.

**A. Approach Calibration**

The use of neural networks, for both the skip-gram models (WORD2VEC) and for the subsequent classifications, requires careful calibration of the networks’ hyperparameters. Indeed, using properly-tuned hyperparameters can improve the overall performance of the neural model [20], [51], [53].

- **For the calibration’s purposes**, we divided the dataset into two sets. Specifically, we used 20% of the data (validation set) to calibrate the proposed approach, while using the remaining 80% for training and testing, performing 10-fold cross-validation (i.e., in each iteration 72% training and 8% testing).

- **The two sets have been defined guaranteeing that each one contains the same proportion of SATD removal instances in the original dataset**. For each model and for each parameter, we train the model on the training set, and evaluate the performance using the evaluation metrics defined in Section IV-B computed on the validation set.

- **In the following**, we explain how we have calibrated the various components of SARDELE, namely the word embeddings, and the two neural networks.
1) Word and Token Embedding Setting: As already reported in Section III, we use \textsc{word2vec} \cite{mikolov2013distributed,mikolov2013efficient} to learn embeddings from comments’ words and from source code tokens. Similarly to what done in previous work using deep learning on source code \cite{wu2017empirical,liu2018learning} but also on natural language \cite{mikolov2013efficient}, the skip-gram model is used with a word vector size of 300, and a token vector size of 400. Similarly to the aforementioned papers, we set the maximum skip length between words to 5, while the maximum skip length between tokens is set to 10. In both cases, we use a softmax layer in order to optimize the output updates and train each model for 100 iterations.

To calibrate the word embedding size, we considered four different sizes, \textit{i.e.}, 16, 32, 64, 128, and found that both precision and recall reach a peak at 32. Therefore, we set the word embedding size to 32. Similarly, to calibrate the token embedding size we tried multiple lengths, \textit{i.e.}, 16, 32, 64, 128, and 256 (we set a higher upper bound because the method body is generally longer than a comment). In this case, the precision/recall peak is achieved with a size equal to 128.

2) CNN and RNN hyperparameters: For the CNN we test different combinations varying (i) the window size, (ii) the number of hidden layers, (iii) the number of neurons in each layer, and (iv) the number of iterations. For the RNN model, we only test different combinations accounting for different numbers of layers, neurons in each layer and iterations. The parameters tuning has been conducting for each model separately.

As regards the window size to consider for the CNN, we experiment with five discrete values, \textit{i.e.}, 1, 3, 5, 7, 9, and we evaluate the performance of each configuration on the validation set, keeping constant the value for the number of hidden layers (\textit{i.e.}, 2), of neurons in each layer (\textit{i.e.}, 60), and iterations (\textit{i.e.}, 100). For all six classification categories, the precision/recall peak is achieved using a window size of 5.

For what concerns the number of layers, we evaluate the models’ performance using five discrete values, \textit{i.e.}, 2, 4, 6, 8, 10. The same values have been used for both the CNN and the RNN models. Fig. 2 shows the $F_1$-score, \textit{i.e.}, harmonic mean of precision and recall, obtained for each SATD removal category varying the number of layers, keeping constant the number of neurons in each layer (\textit{i.e.}, 60). For the CNN (Fig. 2(a)), the maximum $F_1$-score\footnote{We omit separate graphs for precision and recall, since both reach the peak at the same time, therefore consistent with the $F_1$-score.} has been reached using 2 hidden layers only. For the RNN (Fig. 2(b)), instead, the $F_1$-score peak is reached with 4 hidden layers.

Concerning the number of neurons in each hidden layer, for the CNN we experiment with 20 discrete values with a step size equals to 10 in the range $[10 − 200]$, and for the RNN
with 16 discrete values in the range \([50 - 200]\).\(^3\) We fixed the number of iterations to 100, while the window size for the CNN, and the number of hidden layers for each network, have been set to the best configuration obtained in the previous tuning steps. We found an \(F_1\)-score peak using 50 neurons in each layer for the CNN and 120 for the RNN.

Finally, the number of iterations is another key parameter to be set when training a deep neural network, since the weights and biases will be adjusted iteratively in order to narrow down the error rate. While increasing the number of iterations would unavoidably reduce the error rate, it would increase the training cost (i.e., needed time). We evaluate 9 possible numbers of iterations, \(1, 10, 20, 50, 100, 200, 300, 500, 1,000\) and set the other parameters with values obtained in the previous steps. As shown in Fig. 3(a) and Fig. 3(b), the compromise between error rate and time cost is at \(\approx 150\) for CNN and \(\approx 200\) for RNN. However, since we considered a further increase of the time cost still acceptable, we decided to set the number of iterations to 500. In correspondence of such a value, for both CNN and RNN the error rate curve exhibits a knee, and therefore a cost increase is no longer paid back in terms of reduced error rate.

\(\text{B. Evaluation Metrics}\)

Once the network has been calibrated, given the remaining 80%, we performed 10-fold cross-validation to evaluate the performance of the approaches.

First we represent the multi-label classification results as a \(6 \times 6\) confusion matrix (i.e., predicted vs. ground-truth classification). Then, to compare the three different deep-learning approaches (CNN, RNN, and SARDELE), we first use standard metrics in automated classification, namely Precision, Recall, and \(F_1\)-score. Our evaluation favors approaches with high precision without having a very low recall. However, we also want to reduce the possibility of classifications occurred by chance. For this reason, we also discuss the performance in terms of MCC and AUC. AUC, the Area Under the Operating Characteristic Curve metric reflects the extent to which the classifier outperforms a random classifier (i.e., \(AUC = 0.5\)). MCC, the Matthews correlation coefficient, is commonly used in assessing the performance of classifiers dealing with unbalanced data [29]. It is computed according to Equation 2:

\[
\text{MCC}_j = \frac{(TP_j \ast TN_j) - (FP_j \ast FN_j)}{\sqrt{(TP_j + FP_j)(FN_j + TN_j)(FP_j + TN_j)(TP_j + FN_j)}}
\]  \hspace{1cm} (2)

and can be interpreted as a correlation measure: \(MCC < 0.2\) is considered to be low, \(0.2 \leq MCC < 0.4\)—fair, \(0.4 \leq MCC < 0.6\)—moderate, \(0.6 \leq MCC < 0.8\)—strong, and \(MCC \geq 0.8\)—very strong.

To statistically compare results of different approaches in \(\text{RQ}_1\), we use the McNemar’s test [30], and report the Odds Ratio (OR) effect size. Since multiple comparisons are performed, we adjust the obtained \(p\)-values using the Holm’s correction [17], and we assume a significance level \(\alpha = 0.05\).

To address \(\text{RQ}_2\), we compare SARDELE with ML classifiers based on Random Forests (we also tried other classifiers which exhibited worse performance) rebalancing the training set by using SMOTE [6]. Since this is a multi-label classification problem (each instance may belong to multiple removal strategies), we implement it using six different binary classifiers, each one determining whether a SATD removal belongs to a given category, or not. The classifiers have been implemented using \textit{Weka} [16]. More specifically, Random Forest (RF) classifier takes as input the tokens extracted from the SATD comments and the source code of the SATD-affected methods modeled as bag-of-words (BOW). However, since there exist tokens that can be used in comments and source code, we have properly discriminated among them, e.g., the “if” token is used twice, once for identifying its usage in the text representing the comments, and a different token, namely “if\_code” is used to model the usage of the Java keyword “if” in the source code of the SATD-affected method. Both tokens extracted from SATD comments and tokens extracted from source code have been preprocessed as detailed in Section III-A and Section III-B. Finally, we perform a statistical comparison with the automated classifiers, to determine whether a simple ML approach would perform better than SARDELE.

To address \(\text{RQ}_3\), two authors independently looked at the comments of the SATD-affected methods, to determine whether such comments already provided enough indications to cope with the SATD (in such a case the proposed approach would not be useful). They then discussed and sorted out inconsistent classifications to produce a common baseline. Note that human annotators had the possibility to label as “Don’t know” a SATD-related comment, when they judged the available information insufficient to produce a (manual) classification. Having the classification performed by us (i.e., outsiders) and not by original developers simulates a scenario in which newcomers have to deal with SATD removal. Once the classification has been produced, we compare its outcome with the ground-truth (i.e., the actual SATD removal strategy, available in the used dataset by Zampetti \textit{et al.} [58]). Finally, we perform a statistical comparison with the automated classifiers, to determine whether a human-based guessing of the SATD removal strategy would perform better than SARDELE.

\(\text{V. Study Results}\)

This section reports the study results addressing the research questions formulated in Section IV.

\(\text{A. How do different classifiers, based on comments and source code, perform for recommending SATD removals?}\)

Table II reports the results obtained using the CNN classifiers based on the SATD comments only, and without considering the SATD-affected source code. The table reports performance indicators for each category, as well as the overall.

Overall, the CNN classifier reaches a precision \(\approx 39\%\) and a recall \(\approx 41\%\). Moreover, the Overall AUC has a value of

\(^3\)We consider ranges as reported in the literature [24], [51].
works only slightly better than a random classifier. This is confirmed by the MCC that reports a low correlation (i.e., 0.22).

Going into specific categories, only for the Other category the CNN obtains a good balancing between precision and recall (precision $\approx 58\%$ and recall $\approx 74\%$), unsurprisingly because this is the least specific category and the one with the largest percentage of samples. However, for Method Calls the CNN shows a precision of 50% with a recall of 34%. Finally, the worst performance regards the Try-catch category, the one having the lowest number of samples in the dataset, resulting in a very low precision ($\approx 21\%$) and recall ($\approx 27\%$).

Table III reports the performances of the source-based RNN classifier relying on the source code of the SATD-affected methods, without considering the SATD comments. In general, the performance indicators are better than those obtained with the comment-based CNN. The Overall precision raises to 47.40% with an increase also in the recall (i.e., $\approx 44\%$). However, looking at the Overall AUC and MCC, it is possible to state that, even if both values increase with respect to the comment-based CNN, also the source-based RNN classifier works only slightly better than a random classifier.

To compare the performance of the two classifiers working on two different sources of information, Table V reports the results of the McNemar’s test and the Odds Ratio (OR), where an OR greater than one indicates that the second technique outperforms the first one. Looking at the first row, we see that the results obtained with the comment-based CNN and the source-based RNN classifiers are statistically different, with the RNN having 1.53 times more chances to correctly classify than the CNN. We conjecture that there are cases in which the SATD comment is somewhat too general, implying that only looking at the source code it is possible to determine the right action to apply for removing it. As an extreme case consider the SATD comment “FIXME” that could be used for identifying cases in which it is required to change the API since the actual one has a bug, but also for identifying missing functionality that can be addressed by a complex change. However, it is also possible to find cases in which the SATD comment contains the right action to be applied expressed in terms of source code elements such as “TODO: add null check” for which the comment-based CNN classifier recognizes that the action needed is related to Conditional statements.

We then investigate whether it is possible to determine the SATD removal strategy combining the information coming from both the SATD comments and the source code of the SATD-affected methods. The last row in Table IV highlights the overall performance of the combined approach (i.e., SARDELE). The results show that, for each metric, we obtain an improvement of $\approx 10\%$ compared to the source-based RNN classifier, and obviously more compared to the comment-based CNN classifier. More specifically, the precision increases to $\approx 55\%$ and the recall to $\approx 57\%$, with an AUC of 0.73 and a moderate correlation ($MCC = 0.46$). Note that, even if both precision and recall are not very high, they are still promising since SARDELE is performing a multi-class classification.
the worst performance is reported for the minority class, i.e., Try-catch, probably because a few samples does not allow to properly train the classifier on this category. However, for each category, the AUC is always greater than 0.7, indicating that the combination of different sources allows SARDELE to clearly outperform a random classifier.

Finally, as shown in Table V, SARDELE achieves significantly better results ($p$-value < 0.001) than the individual classifiers, and has 2.87 more chances to identify the correct SATD removal category compared to the source-based RNN classifier, and 5.31 more chances than the comment-based CNN classifier.

**B. How does SARDELE perform, compared to simple machine-learning baseline?**

Table VI reports the results obtained using Random Forest (RF) classifiers on the tokens extracted from the SATD comments and the source code of the SATD-affected methods. Overall RF reaches a precision of 45.03% with a very low recall (i.e., 17.68%). Moreover, the Overall AUC is only slightly better than a completely random classifier (0.59). The latter is also confirmed by the MCC value that has a value of 0.21 representing a very low correlation. The results are in line with the ones obtained using only the comment-based CNN classifiers. However, as reported in Table II, the CNN classifier improves the Overall recall (41.03) while degrading the Overall precision (39.39).

Going into specific categories, RF obtains the worst performances for Try-Catch, Method Signature and Return categories. Again, this result is not surprising since these are minority categories (despite rebalancing through SMOTE has been applied). Also in this case, only for the Other category the RF obtains an acceptable balancing between precision and recall, 51.4% and 37.5% respectively.

SARDELE outperforms a simple ML classifier, even if the latter rebalances the minority classes. Specifically, SARDELE improves the Overall precision of $\simeq 10\%$, and the Overall recall of $\simeq 37\%$. As also done for RQ$_1$, we have statistically computed the differences between RF and SARDELE. As reported in the last row of Table V, SARDELE achieves significantly better results than RF ($p$-value < 0.001) having 2.94 more chances to identify the correct SATD removal strategy.

**RQ$_2$ summary:** SARDELE significantly outperforms a simple machine-learning baseline, having 2.94 more chances to identify the correct SATD removal strategy.

**C. How does SARDELE perform, compared to a human baseline?**

As explained in Section IV, we assess the extent to which a manual classification of the SATD comments matches the ground-truth, and whether it is at least outperformed by the automated comment-based CNN classifier, which, based on the results of RQ$_1$, is our lower bound.

Table VII reports a confusion matrix showing the number of correct and incorrect classifications of the comment-based CNN classifier and of the manual classification on the 80% of the training/testing data (we are excluding the validation set).

The manual classification provides a correct outcome only in 141 out of 712 cases ($\simeq 20\%$), whereas the CNN is correct in 354 cases. There are only 70 cases in which the manual approach succeeds where the CNN fails, while the other way around happens in 288 cases. For instance, the comment “TODO we should play nice and only set this if it is null.” highlights the need for adding a pre-condition, however, the CNN predicts the need for a complex change. The comment “If s=0 is used in the URL, the user has explicitly asked us to not perform selection on the server side, perhaps due to it incorrectly guessing their user agent.” is correctly classified by the CNN, even if the comment does not explicitly report the need for adding a conditional. These examples suggest that the context in which the words are used support or hinder the CNN in determining the right SATD removal strategy.

Looking deeper at the different SATD removal categories we find that for Try-Catch, Method Signature, and Return, the manual classification does not help the developer into properly determine the type of change to be applied for removing the SATD. Quite surprisingly, the SATD comment does not help also when determining the need for a complex change (the Other category) since that in many cases, in particular for refactoring activities or addition of a new piece of functionality, the comment is quite general.

Comparing proportions with the McNemar test, we obtain statistically significant differences ($p$-value < 0.001) with an
OR of 4 in favor of the comment-based CNN classifier. The OR reaches 5.98 when comparing with the source-based RNN classifier and 8.56 for the combined approach (SARDELE).

**RQ$_3$ summary:** The automated (CNN) classification of SATD comments — and above all the source-based classifier and the combined one — outperforms a SATD removal strategy identification based on looking at the SATD comment, thus indicating the potential usefulness of SARDELE.

VI. LIMITATIONS AND THREATS TO VALIDITY

In this section we discuss (i) the approach’s limitations, and (ii) the threats to the study’s validity.

A. **Limitations of SARDELE**

The main current limitation of SARDELE is that it works as a “black-box”. Since it is based on a combination of deep neural networks, it produces a classification (i.e., a predicted SATD removal strategy) without explaining how this classification has been produced. Section III has motivated the reasons for choosing a deep learning classifier instead of traditional machine learners, the latter being easier to interpret. While the purpose of this work is to show the potential of deep learning classifiers working on comments and source code to recommend SATD removal strategies, it is worthwhile to complement the work with approaches to interpret the deep neural networks’ classification based on the input features.

Another limitation is that SARDELE only provides a SATD removal strategy without giving a concrete resolution template. To this purpose, SARDELE could be possibly complemented with other techniques and tools, including API recommenders [26], [49], refactoring recommenders [4], [45], smell detectors [34], [45], and program repairing approaches, e.g., [22], [23].

B. **Threats to Validity**

Threats to construct validity concern the relationship between theory and observation. One important threat can be represented by mistakes in the used dataset, in terms of both SATD presence and introduction, removal occurrence and removal strategy. As explained in Section II, we mitigated this problem by using a dataset of SATD occurrences used in different research works [9], [44]. Concerning the removal strategies, these have been addressed by multiple evaluators, as the paper proposing the removal taxonomy has explained [58].

Another threat to construct validity is represented by the way the baseline used for RQ$_3$ has been created. Ideally, a better baseline should have been defined by the original developers who admitted the TD, or at least by other developers of the subject project. At the same time, as explained in Section IV, having outsiders involved in the baseline creation mimics the scenario in which newcomers have to cope with SATD, and would therefore potentially benefit of SARDELE.

Threats to internal validity concern factors, internal to our study, that could have influenced the results. A major factor that could impact the performance of SARDELE is represented by the various hyperparameters of the employed neural networks. As explained in Section IV-A, we have performed a hyperparameter calibration over a validation set (not used our empirical evaluation) and justified the choices we made. Note that the calibration has been done by searching values within certain ranges. We have set the ranges following what done in previous literature [24], [51], and we observed peaks of performance within these ranges or, in the case of iterations, a good compromise between time cost and error rate. Moreover, each parameter has been optimized “individually” since exploring all the possible combinations would be too expensive to run. However, except for the number of layers and the number of neurons in each layer, the value of the parameter does not influence the choice for the other ones [51]. Of course, we cannot exclude better performance outside the ranges.

Threats to conclusion validity concern the relationship between theory and outcome. As explained in Section IV-B, we use suitable statistical tests (McNemar’s test [30]) and effect size measures (Odds Ratio) to support our findings. Also, we report appropriate performance indicators (AUC) including indicators suited for unbalanced datasets (MCC) showing that, at the minimum, SARDELE works better than a random classifier.

Threats to external validity concern the generalizability of our results. As already discussed in Section II, at the moment we have based our evaluation on an existing curated dataset, to mitigate threats to construct validity. At the same time, we need to extend the work on a larger dataset.

VII. RELATED WORK

This section details the related literature to self-admitted technical debt (SATD) and its removal, and deep learning algorithms applied to source code.

A. Self-Admitted Technical Debt (SATD) and its removal

The presence of TD (and SATD) as well as its removal, and therefore the relevance of the problem we are going to cope with, has been investigated by several researchers. Alves et al. [1] showed that TD can be related to different software artifacts and life-cycle activities. In a different work, Zazworka et al. [59] pointed out the need for identifying and handling them to reduce their negative impact on software quality such as maintainability and comprehensibility. The latter has been confirmed by Ernst et al. [13] who showed how TD awareness is a problem in TD management.

Potdar and Shihab [40] found that developers tend to “admit” the presence of TD in the source code through comments (SATD), defining a catalog of 62 patterns for identifying them. Maldonado and Shihab [10], instead, looking at source code comments, classified different types of TD reporting that design SATD are the most common. Bavota and Russo [5] performed a finer categorization of SATD and reported that there is no correlation between SATD and code quality metrics evaluated at class-level. Looking at the change history, they quantified that
\[ \approx 57\% \text{ of SATD are removed from software projects. Zampetti et al. [56] developed an approach aimed at recommending developers when to admit a design TD dealing with code quality metrics and warnings raised by static analysis tools.}

Besides a simple pattern-matching of keywords in comments [10], [40], different approaches for detecting SATD-related comments have been proposed in the literature. Specifically, Maldonado et al. [11] used a Natural Language Processing approach to classify SATD. Also, Ren et al. [41] proposed the use of CNN to classify SATD, outperforming previously-proposed approaches. We share with Ren et al. the use of CNN to classify SATD-related comments. However, SARDELE combines such a CNN with an RNN that processes source code to determine SATD resolution strategies.

The aforementioned works motivate SARDELE, highlighting the developers’ need to cope with (SA)TD removal. Moreover, while previous research has attempted to identify the cause of the problem in the comments [5] or the source code [56], it did not try to exploit comments and source code to recommend solution strategies like SARDELE is doing.

In the past years, the research community has investigated in depth a particular kind of TD, namely code smells. Specifically, they have developed approaches aimed at identifying them [12], [15], [34]. Tufano et al. [46] investigated code smells introduction, survivability and removal looking at the change history of 200 open source projects. Their findings highlight that \( \approx 80\% \) of code smells survive in the system, and only 9% of code smell removal happens together with refactoring operations. Their results are in line with the ones by Bavota et al. [3], who showed that refactoring activities do not result in source code quality metrics improvement but also that only 7% of code smells are removed together with refactoring operations.

Going deeper on SATD, Maldonado et al. [9] conducted an empirical investigation aimed at analyzing the removal of the SATD comments by looking at the change history of five Java open source projects. Their results highlight that (i) there is a high percentage of SATD comments being removed, (ii) most of them are self-removed (i.e., removed by the same developers who have introduced them), and (iii) their survivability varies by project. Moreover, by surveying 14 developers, Maldonado et al. [9] went through the reasons behind the removal of SATD comments. They show that developers tend to remove SATD comments from source code during bug fixing activities, but also when adding new features.

The work that is most related to ours is the one by Zampetti et al. [58]. They conducted an in-depth investigation of SATD removal studying the relation between the removal of the comments and the changes applied to the SATD-affected method. Their results highlight that between 20% and 50% of SATD comments are removed when either the whole class/method is removed. Moreover, even if in addressing SATD developers tend to apply complex source code changes there are many cases in which the SATD removal occurs changing method calls and conditional statements.

Finally, Sierra et al. have recently published a survey of the research work on SATD [43].

B. Deep Learning Algorithms on Source Code

Deep learning algorithms such as Deep Neural Network (DNN) have been applied to the source code, for example in relation to bugs. Zhang et al. [60] used RNN to learn regularities in the source code, define cross-entropy metrics based on the model trained and predict defect-proneness based on these metrics. Manjula and Florence [27] combined DNN with genetic algorithms for metric-based software defect, while Xiao et al. [54] and Lam et al. [21] considered the bug localization. Beyond bug localization and prediction, deep learning techniques have been applied to classify programs based on their functionality [35], [36] and migrate source code from Java to C# [37] through statistical machine translation.

Tufano et al. [47] have proposed an approach for automated program repairing. While they treat source code similarly to us, the approach by Tufano et al. uses an encoder-decoder model to perform neural-machine translation, and therefore recommend repairs. Also, they train their model on bug-fix diffs, whereas in our case we train the RNN on the entire SATD-affected method source code.

In summary, while the aforementioned works share with us the techniques being adopted (deep neural networks), our work differs (i) for its purpose, i.e., to the best of our knowledge, this is the first work aimed at automatically recommending SATD-removal strategies; and (ii) because we combine two different pieces of information, i.e., the SATD comment and the SATD-affected source code, and use two different deep neural networks, i.e., a CNN and an RNN — then combined through a softmax layer — to recommend SATD removals.

VIII. Conclusion

In this paper, we proposed SARDELE, an approach that leverages deep neural network classifiers to recommend strategies for Self-Admitted Technical Debt (SATD) removal. Such strategies are based on a previously-proposed taxonomy of six kinds of SATD-removal patterns [58].

We apply SARDELE on a curated dataset [58] of 779 SATD removals from five Java open source projects. Results of the study indicate the capability of the approach to successfully recommend SATD-removal strategies with a precision of about 55\%, a recall of about 57\%, and an AUC of 0.73. SARDELE outperforms a machine-learning classifier based on Random Forests, and human-produced baseline in which the category was guessing based on the comments’ content.

There are several directions to continue and improve the work. First, although we have deliberately chosen to perform this evaluation on a curated, reliable dataset, there is the need for producing and using a larger dataset to improve the performances of the proposed approach. This would also allow us to refine the taxonomy, producing a finer-grained, more informative recommendation of the removal. Moreover, we plan to use approaches that associate the trained network’s weights with word/token n-grams to provide an explanation of the generated classifications. Last, but not least, we want to conduct a user study aimed at verifying that SARDELE helps developers in removing SATD from the source code.


