Automatically Learning Patterns for Self-Admitted Technical Debt Removal

***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 and commit messages. 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 $\simeq 55\%$, recall of $\simeq 57\%$, 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 [9], 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. As a result, the source code might be partially functional, not robust, or hard to understand and maintain.

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 either adding a comment near the TD-affected source code or in the commit message [43]. In such cases, the TD is considered as a "self-admitted" TD (SATD). SATD comments often contain recurrent patterns: these patterns have been exploited in automated SATD identification using regular expressions [6], [43] or Natural Language Processing (NLP) techniques [28].

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 [6] found that over half (57%) of the SATD is actually being removed, and while in 63% of the cases this task is performed by the same developer who admitted it, in the remaining cases the SATD is addressed by somebody else. While in the former scenario the developer is aware of the nature of the problem, and might also know possible solution strategies, in the latter case the problem is left to the burden of somebody else. This may be especially true in projects with a very high turnover [26], [45], [54].

Concerning "how" SATD is being removed, Wehaibi *et al.* [56] found that SATD may lead, in general, towards complex software changes, while Maldonado *et al.* [10] 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.* [10], Zampetti *et al.* [61] analyzed the change patterns that lead towards SATD removal, by combining an automated analysis though GumTree [14] with manual analysis. Their study produced a taxonomy of six 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 (*e.g.,* precondition) (11-29%), exception handling, method calls (*e.g.,* API) changes, or method signatures changes. Noteworthy, regularities in program changes have also been found in previous work categorizing bug fixing patterns [41], but also in recent work attempting to learn program repairs from existing source code [50].

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.* [61], 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.* [61]. 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.

TABLE I
SATD REMOVALS DATASET BY ZAMPETTI *et al.* [61].

Change	Camel	Gerrit	Hadoop	Log4J	Tomcat	Total
Method Calls	165	16	42	8	59	290
Conditionals	61	8	13	7	59	148
Try-Catch	9	3	2	0	5	19
Method Signature	36	3	6	0	17	62
Return Stmt	15	3	4	0	7	29
Other changes	145	23	67	10	94	339
OVERALL	431	56	134	25	241	887

We apply SARDELE on the previously-available dataset [61], featuring 887 manually-classified method-level SATD removals, each one classified along 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 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.

Paper Structure. Section II details the context and the curated dataset used for defining and validating SARDELE. Section III describes how SARDELE works. Section IV reports the empirical study we conducted, as well as the calibration of the deep neural networks being used. Results are presented and discussed in Section V, whereas the SARDELE's limitations and the study threats are discussed in Section VI. After a discussion of the related literature (Section VII), Section VIII concludes the paper and outlines directions for future work.

The study dataset is available for replication purposes [2].

II. SATD REMOVAL TAXONOMY AND DATASET

The aim of this work is to support developers in addressing the SATD removal, by recommending one of the six removal strategies that have been identified in a previous study by Zampetti *et al.* [61]. Starting from a curated dataset of SATD by Maldonado *et al.* [10], 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;

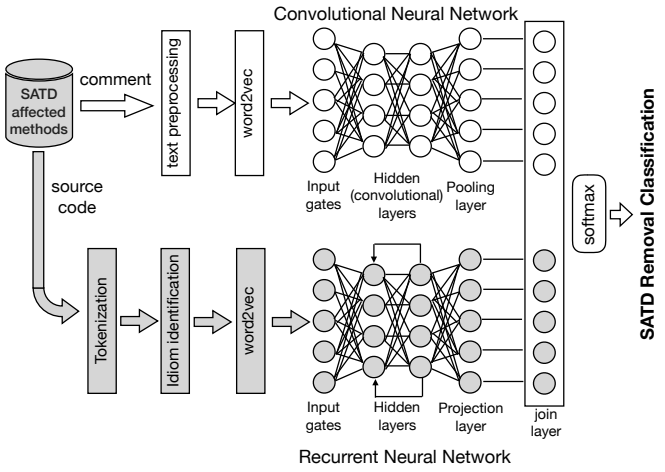


Fig. 1. SATD classification approach (the number of hidden layers and of nodes in the neural network varies based on the calibration).

3) **Try-catch blocks**, *e.g.,* addition of exception handling for previously-uncaught 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 action[1] 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 887 SATD removals, as detailed in Table I. Note that 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 (in future we could combine the proposed approach with refactoring recommenders).

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

[1] apache/tomcat/commit/b0b534f11d90126b9c399e916c456642e9b10e5c

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. We evaluate the two classifiers separately, and in combination.

The top-side of the figure (white blocks) starts with pre-processing the SATD comment. Then, in order to reduce the dictionary size and capture relationships between adjacent words (*i.e.,* 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 along the six SATD removal strategies.

We have chosen to base our approach on deep neural network classification strategies, rather than on traditional machine learners, *e.g.,* decision trees. Truly, traditional machine learners may have advantages in terms of computational cost and of capability to provide an explanation of the predicted classification (*e.g.,* in terms of feature importance or classification tree). At the same time, in our circumstance, we have a scenario in which representing both comments and source code for machine learning would have required a complex, possibly manual, feature engineering, in order to avoid a very sparse representation and to avoid treating terms/tokens as independent events. Therefore, especially in view of previous applications of deep learning to source code analysis [50], [57], we opted for such a solution.

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 [42] stemmer. We evaluated SARDELE with and without stemming and stop words removal, and observed the best performance with stemming and without stop words removal.

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 [3]. 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 [8], [51]. Previous studies have shown that neural language models are able to capture both semantic and syntactic relationships between words [33], [34]. In this work, we use WORD2VEC [33]–[35], a well-known unsupervised word embedding approach, able to learn word representations exploiting the context in which the words appear. WORD2VEC has been used in various software engineering applications, *e.g.,* the identification of similar libraries in different languages [7], the identification of relations between software weaknesses [17], or to relate developers' questions to software documentation [25]. Specifically, we use a continuous skip-gram model [33], [34] 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. Concurring with the fact that 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 own 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 [19]. Kim reports that in NLP text classification tasks CNN built on top of WORD2VEC are able to achieve good performance with very little hyperparameter tuning. Previous studies have reported that CNN is able to capture long-range dependencies and learn to correspond to the internal syntactic structure of sentences, hence reducing the noise [19], [59].

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 on the SATD-related method in order 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 latter is needed in order to make the network able to learn non-linear decision boundaries. Previous literature has shown that it is possible to use several types of activation functions such as logistic sigmoid, hyperbolic tangent and Rectifying Linear Unit (ReLU). However, Nair and Hinton [38] have highlighted the benefits of using ReLU instead of both logistic sigmoid and hyperbolic tangent. For this reason, we choose to use ReLU as the activation function (*i.e.,* $ReLU(x) = max(0, x)$). The output of the activation functions is then passed to a pooling layer with the aim of reducing 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 [30] 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 use Adam optimizer [20] to update the network parameters and add the $L2$-norm regularization loss to avoid model overfitting.

B. Classifying SATD removals from the source code

Source code preprocessing. Given each method affected by SATD, we first extract its tokens using the tokenizer part of the GumTree differencing tool [14]. 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 would also limit the capability of the approach to learn specific features from the source code, *e.g.,* the presence of certain method calls, of certain values in a condition, etc.

A "middle ground" indexing strategy, also adopted in previous work on learning features from source code [50], 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) are a kind of neural network models having a self-connected hidden layer. The basic idea behind RNN models 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 Fig. 1). Then, 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) in order to represent each RNN layer. Indeed, RNNs have been previously used to model source code *e.g.,* in the context of automated program repair [50], or to detect program similarities [57].

The RNN architecture is composed of an input layer, followed by a number of hidden layers (we vary the number of hidden layers but also the number of neurons in each hidden layer in order 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 (Section III-C).

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 [20], and overfitting avoidance through $L2$-norm regularization.

C. 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.

$$p(y = j|X_{pooling}; W_s; b_s) = softmax_j(W_s \cdot X_{pooling} + b_s) \quad (1)$$

In $\mathbf{RQ}_1$ 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, with the aim of assessing its capability to recommend SATD removal categories (hereby referred as "SATD removals"). The *context*, described in Section II, consists of 887 SATD removals belonging to five Java open source projects. The study aims at addressing the following research questions:

- $\mathbf{RQ}_1$: *How different classifiers, based on comments and source code, perform for recommending SATD removals?* In Section III we explained how it could be possible to learn and recommend SATD removals from (i) the SATD comment itself, and (ii) from the source code of the SATD-affected method.
- $\mathbf{RQ}_2$: *Is it possible to improve the performance by combining different sources of information?* Following the previous research question, we look at whether it is possible to improve the performance by combining two classifiers working on different sources of information.
- $\mathbf{RQ}_3$: *To what extent the SATD comment might help a developer to know how to remove the SATD?* If the SATD-related comment already provides enough hints to the developer for removing the SATD, then our approach 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 a careful calibration of the networks' hyperparameter. Indeed, using properly-tuned hyperparameters can improve the overall performance of the neural model [21], [55], [57].

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 instance, since we have 339 SATD removal instances being classified as requiring a complex change we put 272 instances in the training set, and the remaining 67 instances in the validation set. 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.

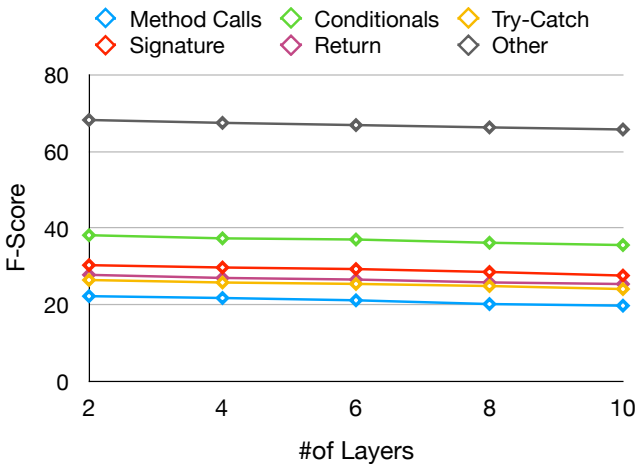
1) Word and Token Embedding Setting: As already reported in Section III, we use WORD2VEC [33]–[35] to learn embeddings from comments' words and from source code tokens. Similarly to what done in previous work using deep learning on source code [50], [57] but also on natural language [64], the skip-gram 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 (following previous literature), *i.e.,* 16, 32, 64, 128, and found that both precision and recall reach a maximum at 32 and then decrease. Therefore, we set the word embedding size to 32. Similarly, to calibrate the token embedding size we tried multiple lengths, *i.e.,* 16, 32, 64, 128, and 256 (here 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.

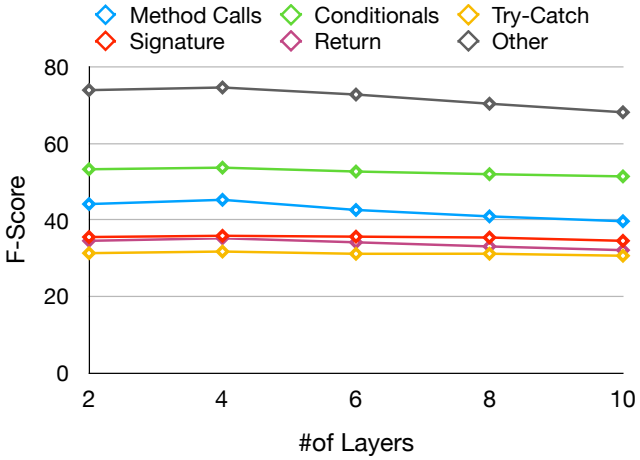
2) CNN and RNN hyperparameters: For the CNN we test different combinations varying (i) the window size, (ii) the number of hidden layers, (iii) the number of neurons in each layer, and (iv) the number of iterations. For the RNN model, we only test different combinations accounting for different numbers 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, *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 (*i.e.,* 2), of neurons in each layer (*i.e.,* 60), and iterations (*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 possible values, *i.e.,* 2, 4, 6, 8, 10. The same discrete values have been used for both the CNN and the RNN models. Fig. 2 shows the F_1-score obtained for



(a) CNN calibration.



(b) RNN calibration.

Fig. 2. Performance varying the number of hidden layers.

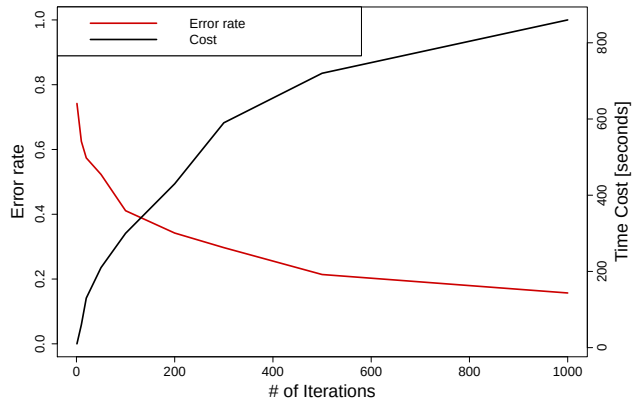
each SATD removal category varying the number of layers, keeping constant the number of neurons in each layer (*i.e.,* 60). For the CNN (Fig. 2(a)), the maximum F_1-score[2] has been reached using 2 hidden layers only. For the RNN (Fig. 2(b)), instead, the F_1-score peak is reached with 4 hidden layers.

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 neurons+ in each layer for the RNN.

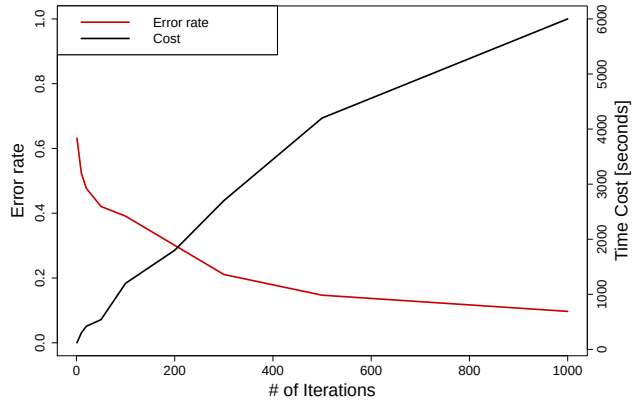
Finally, the number of iterations is another key parameter to be set when training a deep neural network, since the weights and biases will be adjusted iteratively in order to narrow down

[2]We omit separate graphs for precision and recall, which however both reach the peak at the same time, therefore consistent with the F_1-score.

[3]We consider ranges as reported in the literature [25], [55].



(a) CNN calibration.



(b) RNN calibration.

Fig. 3. Error rate and time cost varying the number of iterations.

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, *i.e.,* 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 $\simeq$150 for CNN and $\simeq$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.

B. Evaluation Metrics

Once the network has been calibrated, given the remaining 80%, we performed n-fold cross-validation, with $n = 10$ to evaluate the performance of the approach.

To assess the performance of our deep-learning approaches, we first represent the multi-class classification results as a 6×6 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 *i.e.,* harmonic mean of precision and recall ($F_1 = 2 \cdot (precision \cdot recall)/(precision + recall)$). In evaluating the performance of the different approaches we favor those highlighting a high precision without having a very low recall. However, even if the F_1-score gives an indication about the balance between precision and recall 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 Receiving 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 [31]. It is computed according to Equation 2:

$$MCC_j = \frac{(TP_j * TN_j) - (FP_j * FN_j)}{\sqrt{(TP_j + FP_j)(FN_j + TN_j)(FP_j + TN_j)(TP_j + FN_j)}} \quad (2)$$

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 (comment-based CNN vs. source-based RNN in RQ_1, comment-based CNN and source-based RNN vs. SARDELE in RQ_2) we use the McNemar's test [32], and report the Odds Ratio (OR) effect size (both performed using the *mcnemar.exact* function of the *exact2x2* [15] *R* [44] package). Since multiple comparisons are performed, we adjust the obtained p-values using the Holm's correction [18] which, ranks the n p-values obtained by the comparison and multiplies the first by n, the second-smallest by $n - 1$, and so on. In our statistical tests we assume a significance level $\alpha = 0.05$.

To address 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 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 of the subject program 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, identifying through a diff-based analysis and available in the used dataset by Zampetti *et al.* [61]). 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.

V. Study Results

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

TABLE II
COMMENT CLASSIFICATION WITH CNN: PERFORMANCES ACROSS THE SIX SATD REMOVAL CATEGORIES.

Category	Pr	Rc	F_1	AUC	MCC
Method Calls	50.32	34.05	40.62	0.56	0.13
Conditionals	38.02	38.66	38.33	0.60	0.19
Try-Catch	21.05	26.67	23.53	0.61	0.20
Method Signature	34.09	30.00	31.91	0.61	0.23
Return	34.62	39.13	36.73	0.67	0.33
Other	58.26	73.63	65.05	0.63	0.26
OVERALL	39.39	41.03	39.04	0.61	0.22

TABLE III
SOURCE CODE CLASSIFICATION WITH RNN: PERFORMANCES ACROSS THE SIX SATD REMOVAL CATEGORIES.

Category	Pr	Rc	F_1	AUC	MCC
Method Calls	58.68	30.60	40.23	0.59	0.21
Conditionals	47.48	55.46	51.16	0.69	0.35
Try-Catch	33.33	33.33	33.33	0.65	0.31
Method Signature	52.00	26.00	34.67	0.61	0.31
Return	33.33	30.43	31.82	0.63	0.28
Other	59.59	85.35	70.18	0.68	0.38
OVERALL	47.40	43.53	43.56	0.64	0.31

TABLE IV
COMBINED CLASSIFICATION (SARDELE) PERFORMANCES ACROSS THE SIX SATD REMOVAL CATEGORIES.

Category	Pr	Rc	F_1	AUC	MCC
Method Calls	73.13	63.36	67.90	0.74	0.50
Conditionals	58.47	57.98	58.23	0.73	0.47
Try-Catch	38.89	46.67	42.42	0.72	0.41
Method Signature	50.00	48.00	48.98	0.71	0.44
Return	42.31	47.83	44.90	0.72	0.42
Other	69.10	76.19	72.47	0.75	0.49
OVERALL	55.32	56.67	55.82	0.73	0.46

A. How 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 is able to reach a precision $\simeq 39\%$ and a recall $\simeq 41\%$. Moreover, the Overall AUC has a value of 0.61, meaning that the classifier is performing slightly better than a completely random classifier. This result 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 is able to obtain a good balancing between precision and recall (precision $\simeq 58\%$ and recall $\simeq 74\%$), unsurprisingly because this is the least specific category and the one with the largest percentage of samples. Moreover, for *Method Calls* the CNN classifier shows a precision greater than 50% and a recall of 30%. 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 ($\simeq 21\%$) and recall ($\simeq 27\%$).

Table III reports detailed results considering the performance 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. Specifically, the overall precision raises up to 47.40% with an increase also in the recall (*i.e.,* $\simeq 44\%$). 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. If we look deeper at the SATD removal categories we find that the performances improve everywhere, except for the *Return* category, where we observe a decrease in terms of both precision (from 34.62% to 33.33%) and recall (from 39.13% to 30.43%). Besides the *Other* category (precision $\simeq 60\%$ and recall $\simeq 85\%$), the source-based RNN classifier has a good compromise of precision and recall for *Method Calls* (precision $\simeq 59\%$ and recall $\simeq 31\%$) and *Conditionals* (precision $\simeq 47\%$ and recall $\simeq 55\%$). Moreover, for *Conditionals* the source-based RNN classifier reaches an AUC of 0.69.

In order 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. Focusing the attention on 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 1.5 times more chances to correctly classify than the CNN ($OR = 1.53$). Our conjecture is 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 in order to remove 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 because 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 is able to recognize that the action needed is related to *Conditional* statements.

RQ_1 summary: Leveraging the SATD comment only does not allow us to properly recognize (and classify) the corrective action to be applied for removing the SATD (precision $\simeq 39\%$, recall $\simeq 41\%$, and $AUC = 0.61$). When relying on source code (only), we obtain significantly better performances (precision $\simeq 47\%$, recall $\simeq 44\%$, and $AUC = 0.64$).

B. Is it possible to improve the performance by combining different sources of information?

Based on the results of RQ_1, we check whether it is possible to determine the SATD removal strategy combining

TABLE V
STATISTICAL COMPARISON (MCNEMAR'S TEST p-VALUES AND ODDS RATIO) BETWEEN DIFFERENT CLASSIFICATION APPROACHES.

Comparison	Adj. p-value	OR
Comments (CNN) vs Source code (RNN)	<0.001	1.53
Comments (CNN) vs SARDELE	<0.001	5.31
Source code (RNN) vs SARDELE	<0.001	2.87

TABLE VI
CONFUSION MATRIX COMPARING THE MANUAL CLASSIFICATION AND THE CNN ON THE SATD COMMENTS.

		CNN is correct		
		No	Yes	Total
Manual classification is correct	No	288	283	571
	Yes	70	71	141
	Total	358	354	712

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 $\simeq 10\%$ compared to the source-based RNN classifier, and obviously more compared to the comment-based CNN classifier. More specifically, the precision increases to $\simeq 55\%$ and the recall $\simeq 57\%$, with an AUC of 0.73 and a moderate correlation ($MCC = 0.46$).

Going deeper on the SATD removal categories, we can notice how for *Other*, *Method Calls*, and *Conditionals* SARDELE achieves precision and recall above 50%, with a precision of about 70% for *Method Calls* and *Other* and of 58% for *Conditionals*. For the remaining categories the precision ranges between 38% of *Try-Catch* to 50% of *Method Signature*. Similarly to the comment-based CNN classifier, the worst performance is reported for the minority class in our dataset, *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 sources allows SARDELE to clearly outperform a random classifier.

Finally, as also done for the previous research question, we have statistically computed the differences between (i) the comment-based (CNN) and SARDELE, and (ii) the source-based (RNN) and SARDELE (second and third rows in Table V). In both cases, 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.

RQ_2 summary: Our results confirm that, as conjectured in the introduction, the SATD comment and the source code of the SATD-affected method contain, in many cases, sufficient yet complementary elements for recommending the SATD action to apply for removing the SATD. The combined SARDELE approach significantly outperforms the individual classifiers having at least 2.87 more chances to achieve a correct classification.

C. To what extent the SATD comment might help a developer to know how to remove the SATD?

As explained in Section IV, to provide a baseline for comparison to our approach, we assess the extent to which a manual classification of the SATD comment 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 and RQ_2, is also our lower bound.

Table VI 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 category of the SATD removal.

Looking deeper at the different SATD removal categories we find that for three of them, namely *Try-Catch*, *Method Signature*, and *Return*, the manual classification performed over the SATD comment 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 needs for a complex change (*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". In other words, 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 reason 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, *i.e.,* a category, without giving a concrete resolution template. To this purpose, SARDELE could be possibly complemented with other techniques and tools, including API recommenders [27], [52], refactoring recommenders [5], [47], smell detectors [36], [47], and program repairing approaches, *e.g.,* [23], [24].

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 [10], [46]. Concerning the removal strategies, these have been addressed by multiple evaluators, as the paper proposing the removal taxonomy has explained [61].

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.

We are aware that, besides the baseline comparison of RQ_3, we have not shown that SARDELE actually helps developers. This requires a user study that we plan to perform in the future.

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, and following what done in previous literature, we have performed a hyperparameter calibration over a validation set (not used in the actual empirical evaluation) and justified the choices we made. Note that the calibration has been performed by searching values within certain ranges. We have set these ranges following what done in previous literature [25], [55], and we observed peaks of performance within these ranges or, in the case of iterations, a good compromise between time cost and error rate. Clearly, 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 [32]) 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 definitely need to extend the work on a larger dataset. Not only this would possibly help to improve the classification performance, but it would also allow considering removal categories not contemplated in this study.

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.* [62] pointed out the need for identifying and handling them in order 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 [43] found that developers tend to "admit" the presence of TD in the source code by means of comments (SATD), defining a catalog of 62 patterns for identifying them. Maldonado and Shihab [11], instead, looking at source code comments classified different types of TD reporting that design SATD are the most common. In a different study, Bavota and Russo [6] performed a finer categorization of SATD and reported that there is no correlation between SATD and code quality metrics evaluated at class-level. Moreover, looking at the change history, they quantified that $\simeq 57\%$ of SATD are removed from software projects. Finally, Zampetti *et al.* [60] developed an approach aimed at recommending developers when to admit a design TD dealing with code quality metrics and warnings raised by static code analysis tools.

The aforementioned works not only motivate SARDELE, highlighting the developers' need to cope with (SA)TD removal. However, while previous research has attempted at identifying the cause of the problem in the comment [6] or the source code [60], 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 and tools aimed at identifying

them [12], [16], [36]. Tufano *et al.* [49] investigated code smells introduction, survivability and removal looking at the change history of 200 open source projects. Their findings highlight that $\simeq 80\%$ 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.* [4], 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.

Code smells are not only related to production code but also to test code. Tufano *et al.* [48] conducted an empirical investigation focusing on the removal of test smells [53]. Their results indicate that test smells have very high survivability, and only 7% of test smells are actually removed compared to the total number of test smells in the test suite.

Going deeper on SATD, Maldonado *et al.* [10] conducted an empirical investigation aimed at analyzing the SATD comments removals 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 conducting a survey with 14 developers, Maldonado *et al.* [10] 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.* [61]. 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-related method. Their results highlight that between 20% and 50% of SATD comments are removed when either the whole class/method is removed. Moreover, they found that 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.

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 the past often in relation with bugs. Zhang *et al.* 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 [63]. Manjula and Florence combined DNN with genetic algorithms for metric-based software defect [29], while Xiao *et al.* [58] and Lam *et al.* [22] considered bug localization. Beyond bug localization and prediction, deep learning techniques have been applied to classify programs based on their functionality [37], [39] and migrate source code from Java to C# [40] through statistical machine translation.

Tufano *et al.* [50] have proposed an approach for automated program repairing. Their approach learns from a large set of changes. 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 recommends strategies for Self-Admitted Technical Debt (SATD) removal. Such strategies are based on a previously-proposed taxonomy of six kinds of SATD-removal patterns [61].

SARDELE takes as inputs the SATD comment and the source code of the SATD-affected method. Then, it combines two deep neural network classifiers, *i.e.,* a Convolutional Neural Network (CNN) aimed at classifying comments, and a Recurrent Neural Network (RNN) aimed at classifying the source code. Finally, the output of the two networks is combined to produce the classification.

We apply SARDELE on a curated dataset [61] of 887 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. At the same time, performances vary among categories, also depending on the available training data. We also show that the combination of comment and source code significantly improves the classification with respect to using comments only or source code only. Finally, SARDELE outperforms a human-produced baseline in which the category was guessed 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. At the same time, this would allow us to refine the taxonomy, producing a finer-grained, more informative recommendation of the removal. Last, but not least, interpretation of the provided recommendation is particularly important. 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.

References

[1] N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola, "Towards an ontology of terms on technical debt," in *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*. IEEE, 2014.

[2] Anonymous, "Automatically learning patterns for self-admitted technical debt removal. Detailed Results https://tinyurl.com/y52zv889."

[3] R. A. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[4] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, 2015.

[5] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: an improved method and its evaluation," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1617–1664, 2014.

[6] G. Bavota and B. Russo, "A large-scale empirical study on self-admitted technical debt," in *International Conference on Mining Software Repositories*. ACM, 2016.

[7] C. Chen, S. Gao, and Z. Xing, "Mining analogical libraries in q&a discussions - incorporating relational and categorical knowledge into word embedding," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, 2016, pp. 338–348.

[8] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. P. Kuksa, "Natural language processing (almost) from scratch," *Journal of Machine Learning Research*, vol. 12, pp. 2493–2537, 2011.

[9] W. Cunningham, "The WyCash portfolio management system," in *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications*. ACM, 1992.

[10] E. da S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik, "An empirical study on the removal of self-admitted technical debt," in *2017 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2017.

[11] E. da S. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical debt," in *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on*. IEEE, 2015.

[12] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: Are we there yet?" in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, 2018, pp. 612–621.

[13] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015.

[14] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 313–324.

[15] M. P. Fay, "Two-sided exact tests and matching confidence intervals for discrete data," *R Journal*, vol. 2, no. 1, pp. 53–58, 2010.

[16] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[17] Z. Han, X. Li, H. Liu, Z. Xing, and Z. Feng, "Deepweak: Reasoning common software weaknesses via knowledge graph embedding," in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, 2018, pp. 456–466.

[18] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian Journal of Statistics*, vol. 6, no. 2, pp. 65–70, 1979.

[19] Y. Kim, "Convolutional neural networks for sentence classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, 2014, pp. 1746–1751.

[20] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[22] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Bug localization with combination of deep learning and information retrieval," in *IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, May 2017, pp. 218–229.

[23] X. Le, D. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: Syntax- and semantic-guided repair synthesis via programming by examples," in *FSE*, 2017, pp. 593–604.

[24] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 54–72, 2012.

[25] J. Li, A. Sun, and Z. Xing, "Learning to answer programming questions with software documentation through social context embedding," *Inf. Sci.*, vol. 448-449, pp. 36–52, 2018.

[26] B. Lin, G. Robles, and A. Serebrenik, "Developer turnover in global, industrial open source projects: Insights from applying survival analysis," in *12th IEEE International Conference on Global Software Engineering, ICGSE 2017, Buenos Aires, Argentina, May 22-23, 2017*, 2017, pp. 66–75.

[27] B. Lin, F. Zampetti, G. Bavota, M. Di Penta, and M. Lanza, "Pattern-based mining of opinions in q&a websites," in *Proceedings of the 41th International Conference on Software Engineering, ICSE 2019, Montréal, Canada, May 25-31, 2019 [To Appear]*, 2019.

[28] E. Maldonado, E. Shihab, and N. Tsantalis, "Using natural language processing to automatically detect self-admitted technical debt," *IEEE Transactions on Software Engineering*, 2017.

[29] C. Manjula and L. Florence, "Deep neural network based hybrid approach for software defect prediction using software metrics," *Cluster Computing*, pp. 1–17, 2018.

[30] X. Mao, Q. Li, H. Xie, R. Y. K. Lau, and Z. Wang, "Multi-class generative adversarial networks with the L2 loss function," *CoRR*, vol. abs/1611.04076, 2016.

[31] B. W. Matthews, "Comparison of the predicted and observed secondary structure of T4 phage lysozyme," *Biochimica et Biophysica Acta (BBA*, vol. 2, no. 405, pp. 442–451, 1975.

[32] Q. McNemar, "Note on the sampling error of the difference between correlated proportions or percentages," *Psychometrika*, vol. 12, no. 2, pp. 153–157, Jun 1947.

[33] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013.

[34] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States.*, 2013, pp. 3111–3119.

[35] T. Mikolov, W. Yih, and G. Zweig, "Linguistic regularities in continuous space word representations," in *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 9-14, 2013, Westin Peachtree Plaza Hotel, Atlanta, Georgia, USA*, 2013, pp. 746–751.

[36] N. Moha, Y. Guéhéneuc, L. Duchien, and A. L. Meur, "DECOR: A method for the specification and detection of code and design smells," *IEEE Trans. Software Eng.*, vol. 36, no. 1, pp. 20–36, 2010.

[37] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, 2016, pp. 1287–1293.

[38] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[39] A. T. Nguyen and T. N. Nguyen, "Automatic categorization with deep neural network for open-source java projects," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, 2017, pp. 164–166.

[40] A. T. Nguyen, T. D. Nguyen, H. D. Phan, and T. N. Nguyen, "A deep neural network language model with contexts for source code," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 323–334.

[41] K. Pan, S. Kim, and E. J. W. Jr., "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, 2009.

[42] M. F. Porter, "Snowball: A language for stemming algorithms," Published online, October 2001.

[43] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014.

[44] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2017.

[45] G. Robles and J. M. González-Barahona, "Contributor turnover in libre software projects," in *Open Source Systems, IFIP Working Group 2.13 Foundation on Open Source Software, June 8-10, 2006, Como, Italy*, 2006, pp. 273–286.

[46] G. Sierra, A. Tahmid, E. Shihab, and N. Tsantalis, "Is self-admitted technical debt a good indicator of architectural divergences?" in *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, 2019, pp. 534–543.

[47] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Trans. Software Eng.*, vol. 35, no. 3, pp. 347–367, 2009.

[48] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[49] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Trans. Software Eng.*, 2017.

[50] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 832–837.

[51] J. P. Turian, L. Ratinov, and Y. Bengio, "Word representations: A simple and general method for semi-supervised learning," in *ACL 2010, Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, July 11-16, 2010, Uppsala, Sweden*, 2010, pp. 384–394.

[52] G. Uddin and F. Khomh, "Opiner: an opinion search and summarization engine for APIs," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, 2017, pp. 978–983.

[53] A. van Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*, 2001.

[54] B. Vasilescu, D. Posnett, B. Ray, M. G. J. van den Brand, A. Serebrenik, P. Devanbu, and V. Filkov, "Gender and tenure diversity in GitHub teams," in *CHI Conference on Human Factors in Computing Systems*, ser. CHI. ACM, 2015, pp. 3789–3798.

[55] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 297–308.

[56] S. Wehaibi, E. Shihab, and L. Guerrouj, "Examining the impact of self-admitted technical debt on software quality," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*. IEEE, 2016.

[57] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," *CoRR*, vol. abs/1707.04742, 2017.

[58] Y. Xiao, J. Keung, Q. Mi, and K. E. Bennin, "Improving bug localization with an enhanced convolutional neural network," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, Dec 2017, pp. 338–347.

[59] L. Yu, K. M. Hermann, P. Blunsom, and S. Pulman, "Deep learning for answer sentence selection," *CoRR*, vol. abs/1412.1632, 2014.

[60] F. Zampetti, C. Noiseux, G. Antoniol, F. Khomh, and M. Di Penta, "Recommending when design technical debt should be self-admitted," in *2017 IEEE International Conference on Software Maintenance and Evolution*, 2017.

[61] F. Zampetti, A. Serebrenik, and M. Di Penta, "Was self-admitted technical debt removal a real removal?: an in-depth perspective," in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, 2018, pp. 526–536.

[62] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011.

[63] X. Zhang, K. Ben, and J. Zeng, "Cross-entropy: A new metric for software defect prediction," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, July 2018, pp. 111–122.

[64] Y. Zhang and B. C. Wallace, "A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification," in *Proceedings of the Eighth International Joint Conference on Natural Language Processing, IJCNLP 2017, Taipei, Taiwan, November 27 - December 1, 2017 - Volume 1: Long Papers*, 2017, pp. 253–263.