Received: 21 April 2025 | Revised: 12 September 2025 | Accepted: 7 October 2025 | Published online: 21 November 2025

RESEARCH ARTICLE

Journal of Computational and Cognitive Engineering
yyyy, Vol. XX(XX) 1–5
DOI: 10.47852/bonviewJCCE52025976

BON VIEW PUBLISHING

Automated Classification of Self-Admitted Technical Debt Using Advanced Word Embedding Techniques

Satya Mohan Chowdary Gorripati[1], Ali Altalbe[2] and Prasanna Kumar Rangarajan[1,*]

[1] *Department of Computer Science and Engineering, Amrita Vishwa Vidyapeetham-Chennai, India*

[2] *Faculty of Computing and Information Technology, King Abdulaziz University, Saudi Arabia*

Abstract: This research uses advanced word embedding techniques to improve the automatic classification of Self-Admitted Technical Debt (SATD). We evaluate how successfully n-gram Inverse Document Frequency (IDF) creates machine learning classifier-friendly feature sets. A publicly available dataset including Java source code comments from 10 open-source projects was used to assess SATD classification methods. This category included Random Forest, SVM, Logistic Regression, and XGBoost. We used instance hardness undersampling to handle the imbalance in the SATD dataset. We tested the classifier using accuracy, recall, F1-score, and Macro-Averaged Mean Cost-Error (MMCE). The Random Forest classifier with n-gram IDF features achieved an average accuracy of 87%. It performed similarly to the traditional TF-IDF and Bag-of-Words methods on average, and on certain projects and MMCE values, it performed better. In rare circumstances, n-gram IDF may reveal contextual phrase patterns and improve SATD recognition, especially when combined with ensemble learning. To enhance generalisability, future research will expand the dataset, investigate hybrid and deep learning models, and increase applicability across various programming languages and project areas.

Keywords: self-admitted technical debt, n-gram IDF, word embeddings, feature extraction, natural language processing

1. Introduction

Self-Admitted Technical Debt (SATD) occurs when developers are aware that they are creating and documenting Technical Debt (TD) and they are okay with the fact that their solutions are not optimal or that they may encounter problems later [1]. Many comments share recurring patterns that indicate the presence of SATD [2]. Because such comments are so frequent [2], it is essential to create automated ways to detect and fix TD in software projects. This need is particularly acute for SATD because developers are aware of and track TD. In a short time, deep learning has made significant progress. It appears to be able to help with hard text classification tasks, such as finding SATD in code comments. These methods use semantic information and links between different contexts to create a solid foundation for finding and sorting TD. There has been considerable research on how to find and sort SATD, which is often divided into groups such as design, defect, test, requirement, and documentation debts [1, 2]. It is better to understand the types of SATD than just knowing that it exists, because each type has its own effects and traits on a software project. Natural Language Processing (NLP) methods for binary classification have been widely used in this field.

A developer can inadvertently create TD by providing an incorrect parameter because they do not understand how the new system architecture works. This could cause bugs. On the other hand, purposeful TD occurs when engineers knowingly use parts of a system that do not function correctly and document these decisions to correct and update later. SATD is a debt you know you have and that you need to consciously deal with. Over time, this makes maintaining high-quality software extremely difficult [1, 2].

This situation can not only increase development costs but also lower the quality of the program, for example, by revisiting defects that have already been fixed. Developers intentionally place SATD in comments within the source code. Understanding SATD is important because it reveals the intentional trade-offs made during software development that will need to be corrected later. SATD is easier to monitor and manage than unintentional debts, which can occur accidentally because of reasons such as being new to something or not paying attention. Potdar and Shihab [2] have investigated how to search SATD in code comments because tracking these debts is very important. They noted similarities in the remarks that indicate indebtedness. Software projects often contain TD; therefore, it is crucial to automatically detect and repair it [3].

Secure data transport protocols and integrity verification procedures are utilised before classifying features to avoid data injection or manipulation attempts from corrupt SATD labels [3]. This necessity is particularly significant when developers have written about SATD. Advances in deep learning have shown promise in detecting SATD in code comments, a difficult text-categorisation challenge. These approaches simplify the identification and sorting of TD by using contextual linkages and semantic information. Knowing your SATD type is better than just knowing it exists because each type affects software projects differently. This research often uses NLP to classify comments into binary SATDs categories without identifying the comment itself. When monitoring multiple SATD models, these approaches can become cumbersome [4].

Several new field ideas show promise. Ren et al. [5] used a convolutional neural network to classify SATD code comments, achieving better performance than traditional text mining techniques.

***Corresponding author:** Prasanna Kumar Rangarajan, Department of Computer Science and Engineering, Amrita Vishwa Vidyapeetham-Chennai, India. Email: r_prasannakumar@ch.amrita.edu

Wang et al. [6] used an attention mechanism and ELMo (Embeddings from Language Models) to detect SATD in several projects. A SATD analysis request is generated by the system—defining the type of comments, amount of data, and quality criteria—and is assigned to the classification module. The classification module organises the preprocessing pipeline and assigns tokenisation, n-gram IDF feature extraction, and classification to the workflow components after receiving the request. Several non-machine learning approaches are used to ensure secure and reliable data processing in the modules.

Chen et al. [7] also discovered a novel method for sorting various types of SATD using feature selection and eXtreme Gradient Boosting (XGBoost). Therefore, new standards have been established in this field. However, it requires a great deal of effort to manually collect and organise these comments, as it is difficult to make it function effectively and thrive.

Early research has somewhat automated this process, but current approaches are not very practical due to the large volume of data and the difficulty of analysing comments. In NLP, we can use sophisticated word embedding models such as Word2Vec, FastText, and GloVe to discover a new method for automatically sorting SATDs. Using these models to examine text data makes it easy to distinguish various forms of TD. This is because these models are able to detect subtle linguistic changes based on context. Building on previous research, this article proposes a new SATD classification scheme, which transforms a simple binary system into a more intricate multi-class system [8, 9]. Our main idea behind our approach is to group comments into three categories: non-SATD, requirement SATD, and design SATD. Recent research directions have yielded promising results.

Unmanaged TD can harm software initiatives in the long term. It increases maintenance costs owing to bug patches and rework, and lowers software quality by affecting performance, maintainability, and scalability. Undocumented SATD makes future improvements more complicated and time consuming, reducing developer productivity. Therefore, SATD detection must be accurate and early. Teams may proactively manage and resolve such debt throughout development, minimising long-term risks and costs, and improving software quality. SATD classification is done using n-gram Inverse Document Frequency (IDF). Even though Bag-of-Words and Term Frequency (TF)–IDF work in some cases, they cannot capture the multi-word, phrase-level patterns in SATD comments. N-gram IDF preserves contextual signals, boosting classifier discrimination. Unlike deep contextual embeddings (e.g., BERT) that need large datasets and computational resources, n-gram IDF balances interpretability, computational efficiency, and accuracy, making it ideal for SATD detection in domains with smaller datasets.

2. Related Works

Da Silva Maldonado et al. [1] added four heuristic filters to improve this. Focus on design and requirement debts was facilitated by evaluating only comments that were likely to contain SATD. Da Silva Maldonado et al. [1] classified these patterns as design, requirement, defect, test, and documentation TD. Poor software can cause both intentional and unintentional debt. The code comment mining method proved effective in this study. Comments can provide the author with feedback on their code and highlight project faults. Thus, they are an excellent approach to learn TD types [2]. This strategy leverages the importance of these statements as they typically provide a brief yet compelling explanation of the challenges and software goals. This preliminary analysis suggests that comment patterns can identify SATD, but this requires a significant amount of work. Table 1 shows a comparative summary of the major existing approaches for identifying SATD. In a groundbreaking study, Potdar and Shihab [2] manually searched over 100,000 code comments for phrases related to SATD, such as ‘hack’ and ‘fixme’. Recent advances in machine learning and deep learning have created new SATD solutions. Potdar and Shihab [2] describe SATD as developer debt based on source code comments. We have added a schema to this SATD detection model.

Huang et al. [3] combined automated text mining with feature selection to construct a composite classifier. Classifiers from other projects can help this classifier find objects. They refined this approach by creating a SATD detection tool for integrated development environments that uses a naive Bayes multinomial classifier to detect SATD and information gain to identify the most important qualities. TD significantly impacts software development. Although SATD can be identified and classified in various ways, researchers aim to achieve greater accuracy with less effort, where it can be identified using pattern-based, machine learning, deep learning, and change-level techniques. Huang et al. [3] created a text mining technique that automatically identifies SATD comments using sophisticated feature selection. Machine learning is also increasingly used in this field.

Ren et al. [5] predicted SATD using a Convolutional Neural Network, demonstrating the power of deep learning. TD substantially impacts software development. It outlines various aspects that can hurt software projects and reputations. TD is better understood through various classification efforts in that it is grouped into multiple categories. We also have design, coding, staff, and usability debt [10]. Practitioner perspectives on TD management have also been surveyed to understand the reasons for paying or deferring debt items [11]. Many automated tools help detect TD kinds today. Software product debt is often detected using specific metrics or informal methods. Design

Table 1
Comparative analysis of existing approaches for identifying SATD

Approach	Solution	Advantage	Disadvantage	Reproducibility
Pattern Matching	Utilises 62 patterns identified by expertsv	Simple and intuitive	Low accuracy due to human limits	Easy (open source)
NLP	Applies maximum entropy classifiers	High accuracy	Model complexity	Medium (partial open source)
Text Mining (TM)	Employs Naive Bayes classifier with voting	High accuracy	Model complexity; less intuitive	Easy (open source)
CNN	Uses convolutional neural networks	High accuracy; intuitive	Very complex model	Hard (not open source)
Jitterbug	Extracts patterns with human assistance	High accuracy; intuitive	Requires significant human effort	Easy (open source)

debt detection methods can employ "God Classes" detectors, which suggest that debt symptoms may be comparable. People's attitudes toward design and code debt vary based on indicators that reveal their relationship. Poor software practices can generate both types of debt. In this scenario, code comment mining proved helpful.

Methods Based on Patterns: Pattern-based SATD detection is one of the earliest methods that searches for code comment patterns that indicate TD. De Freitas Farias et al. [12] improved SATD recognition with a context-based language model. The model scores these patterns and shows how they are associated with different debt types. Many researchers have used different methods to classify SATD. Their proposed contextual vocabulary model advances research in the field by using code tags and word classes to detect TD in source code comments. This method simplifies SATD sorting. This comprehensive classification underpinned our investigation into the nine categories of developer source code comments. Experts used machine learning and conventional methods with Yu et al.'s [13] Jitterbug architecture to find challenging SATD problems. They employed an attention-based BiLSTM network to find the most important parts. A balanced cross-entropy loss function was used to correct the data imbalance. Wattanakriengkrai et al. [14] proposed the use of n-gram IDF to improve the detection of design and requirement SATDs. This method emphasises the importance of phrases rather than individual words, thereby reducing noise during classification.

Zhao et al. [15] developed a simplified deep forest model for just-in-time defect prediction in Android mobile applications. Their approach outperformed baseline methods in terms of AUC and cost-effectiveness, demonstrating the potential of ensemble learning for defect prediction.

Each approach has pros and cons. Traditional pattern-based methods require manual work and can be biased because pattern selection is subjective. Machine learning simplifies manual work, but uneven data and insufficient features hinder its application. Murillo et al. [16] conducted a systematic mapping study on the identification and management of TD. They emphasised challenges such as reliance on version history data, computational complexity, and uneven datasets, while also noting the increasing use of NLP techniques in SATD detection.

Flisar and Podgorelec [17] utilised word embedding to identify features to improve SATD predictions. Advanced computer models can help users understand source code comments.

Yu et al. [18] introduced a gated graph neural network framework for detecting and explaining SATDs. Their model not only improved classification accuracy but also provided interpretability by highlighting the relationships among code comment tokens that led to the identification of TD.

Due to SATD comments being unevenly distributed among projects, instance hardness undersampling is employed to balance datasets in SATD detection [19]. Eliminating noise and hard-to-classify samples can improve the classifier training datasets.

To address these challenges, our current research aims to enhance the identification and classification of SATD by leveraging advanced word embedding techniques [20, 21]. These techniques represent a significant leap forward from traditional NLP methods, promising to improve the accuracy and efficiency of SATD detection by better capturing the contextual and semantic nuances within code comments. Models that identify SATD types are developed and evaluated in this study to provide more targeted insights for software maintenance and evolution.

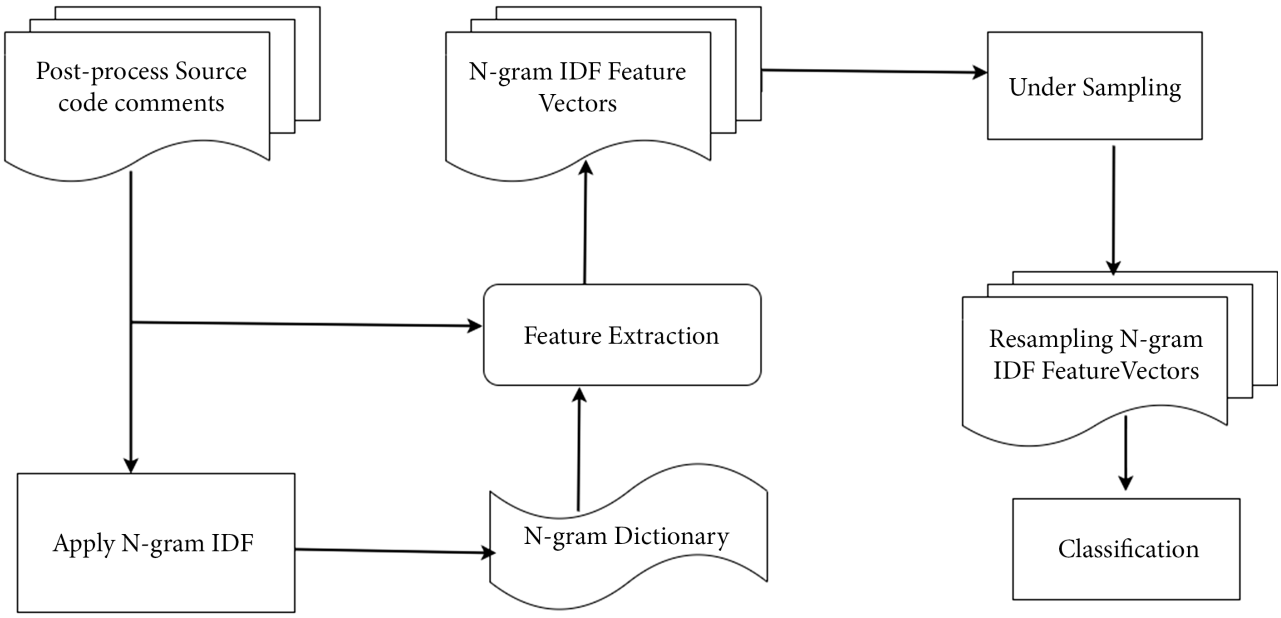
3. Research Methodology

Through the use of word embedding methods, our comprehensive approach to recognising SATD simplifies both the analysis of source code comments and the identification of SATDs. Machine learning and n-gram IDF feature extraction are used to discover the SATD problem. The preprocessing up to the classification stage is illustrated in the model architecture shown in Figure 1.

3.1. Model construction and preprocessing

We start our model by carefully preprocessing the source code

Figure 1
Process of our SATD classification method



comments. To avoid skewing in the classification results, the input given to the model must be normalised and free from inconsistencies. Our first step in building the model for automatic classification of SATD using sophisticated word embedding is the preprocessing of source code comments [22]. Our innovative word embedding approach to recognising SATD starts with thorough preprocessing. First, normalise the source code comments. This step corrects textual inconsistencies, standardises lexical representations, and removes non-contributory segments that can cause system noise. Normalisation converts all text to standard case to ensure dataset uniformity. Non-alphanumeric characters, which frequently have implicit value in programming, are converted to interpretable tokens with care. The question marks can be encoded as 'questionmark', which plays a significant role in n-gram analysis. After purification, the comments are tokenised into n-grams. The IDF extraction process relies on these n-grams. We use this method to encode each comment as a coherent vector of n-grams for feature selection and machine learning. Our objective is to standardise the comments so that our n-gram IDF extraction program can handle them. We focus on non-alphanumeric characters because they can be important in code comments. Encoding characters such as '?' as 'questionmark' is descriptive. Encoding these characters allows the n-gram IDF extraction tool to recognise and analyse them as part of the language model, rather than discarding them as noise. Post-preprocessing comments are ready for n-gram IDF extraction. This extraction will dissect cleaned comments into n-gram phrases for feature extraction and categorisation. Preprocessing produces a clean, standardised text corpus for n-gram analysis. This corpus will be used to find SATD patterns and words for the machine learning model to learn from and accurately classify. These stages prepare the input data for efficient processing and analysis by our sophisticated word embedding classification model.

3.2. N-gram IDF extraction and dictionary building

Ngweight1, an n-gram IDF extraction technique, helps us generate an n-gram dictionary from the training set. This dictionary contains all legal n-gram phrases, along with useful metrics that include identities, lengths, global term frequencies, and document frequencies. After preprocessing source code comments, we extract n-grams and build an IDF dictionary. This step helps transform raw text into a structured representation that captures word sequence context. Deploying an n-gram IDF tool starts extraction [23, 24]. This tool is carefully designed to analyse the frequency of contiguous word sequences in our corpus to find and quantify TD phrases.

$$IDF\left(t\right) = log\left(\frac{N}{df(t)}\right) \tag{1}$$

where N is the total number of documents in the corpus and df(t) is the document frequency of term t, representing how many documents contain the term.

From this analysis, we compile a comprehensive n-gram dictionary. This resource provides a complex map of n-gram phrases, including their identities, lengths, global frequencies across all texts, and document frequencies—both alone and as parts of bigger structures. This extensive dataset lets us determine the proportional value of n-grams in TD. A dictionary like this is very difficult to compile. The abundance of data can make an unmanageable lexicon with unhelpful n-grams. We use a strict feature selection approach to prioritise frequent, SATD-discriminative n-grams based on their IDF scores. Our solution relies on carefully building the n-gram IDF dictionary. The resulting feature vectors, which form the classification model, represent the text and are optimised for SATD detection in code comments.

3.3. Feature filtration and vector representation

Feature filtering and vector representation follow the n-gram IDF dictionary construction. The refinement of the feature set and classification vectors that capture TD in code comments relies on this approach. Using this n-gram dictionary, we filter out useless characteristics. Using a weight one score for each n-gram phrase, we remove less important ones, usually those that occur just once throughout all publications. This pruning formula takes into account the number of documents, the document frequency of the constituent terms of the n-gram, and its global term frequency [25]. We focus on the top 10% of n-grams with the highest weight one scores to create a 6,000–7,000-term vocabulary. These selected n-grams are utilised to construct feature vectors, each representing the raw frequency of an n-gram in a comment.

To eliminate less informative n-grams, we generate a 'weight one' score using the formula:

$$Weight1\left(t\right) = log\left(\frac{|D|}{sdf(t)}\right) \times gtf\left(t\right) \tag{2}$$

where $|D|$ is the total number of documents, $sdf(t)$ is the document frequency of the set of words composing the n-gram, and $gtf(t)$ is the global term frequency. We add the top 10% of n-grams with $Weight1$ scores to our revised vocabulary. By calculating the raw frequency of each kept n-gram in the comments, feature vectors are created for each remark. Infrequent n-grams with a global word frequency of one are eliminated to filter characteristics. This stage eliminates rare n-grams that are unlikely to be relevant for categorisation. Next, we generate a 'weight one' score for each n-gram based on document and global phrase frequency to determine its relevance. In distinguishing SATD from non-SATD remarks, this rating is crucial for n-gram value. The top percentile of n-grams with 'weight one' scores is selected to narrow down our lexicon to the most useful features, reducing the dimensionality and improving the prediction capability of our model. A handpicked array of n-grams is used for vector representation. We create a vector from the raw frequency of each n-gram for each code comment. Each vector represents the comment as a numeric vector with dimensions according to the selected n-grams and values representing their frequency. Textual data is encoded into vectors for classification by the machine learning model.

This rigorous feature filtering and vector representation prepares our model for machine learning classification by identifying SATD patterns. This feature set compresses the data and highlights SATD features, which are critical for categorisation.

3.4. Instance hardness undersampling

To handle an unbalanced dataset, we use instance hardness undersampling and the imbalanced-learn package [26]. This phase is crucial since it removes non-essential samples that might hinder the classifier. The same classifier and hyperparameters are used throughout learning and undersampling, ensuring instance hardness consistency. The instance hardness undersampling phase in our suggested technique addresses the class imbalance of the training data to improve the performance of the classifier.

Class imbalance occurs when incidences of distinct categories are disproportionately frequent, perhaps favouring the dominant class. Instance hardness undersampling refines our training vectors. This approach measures each instance 'hardness'—how difficult it is for the classifier to properly predict its class. Frequent misclassifications are considered 'hard' and can be removed from the training set. Hardness

is measured using the same classifier and hyperparameters as the final model to match the undersampling procedure with the classification goal.

$$\text{Hardness}(x) = 1 - P\max(y|x) \quad (3)$$

where $P\max(y|x)$ is the highest probability assigned to any class y for the instance x by the classifier. Instance hardness undersampling lowers misclassification by deliberately deleting 'hard' samples, enhancing model generalisability. This balances the training set, which is essential for training a robust classifier that can properly recognise SATD in various scenarios. Resampled training sets better depict class distributions, making them better for classifier training and laying the groundwork for accurate and reliable SATD classification in the final step of our model. Undersampling increases the classifier performance and streamlines training by reducing the number of training cases.

3.5. Machine learning classifiers

Our study aims at classifying SATD using advanced word embedding techniques. We integrate three distinct machine learning classifiers, each selected for its proven efficacy in similar applications and inherent characteristics that benefit our specific context. These classifiers are Logistic Regression, Random Forest, and XGBoost, which represent different approaches to machine learning—namely, regression, bagging, and boosting.

1) Logistic regression

We use Logistic Regression because of its success in binary classification challenges, especially in NLP for software engineering. Logistic Regression is a linear classifier that works well when the class-feature relationship is linear. The simplicity, speed, and interpretability of its output make it a useful tool for initial evaluation and application where understanding specific aspects is crucial.

2) Random forest

Random Forest, a bagging-based ensemble learning algorithm, is resilient and accurate in classification tasks. It builds numerous decision trees and votes on the most popular output class. Due to its fundamental mechanism of averaging multiple deep decision trees to reduce variance, this approach is good at handling huge datasets with high dimensionality and complicated structures without overfitting. Random Forest is reliable for detecting TD patterns in SATD, when the feature space is large and variable. The scikit-learn package provides a random forest method for categorisation. The classifier is carefully trained using resampled training data to classify source code comments as design, requirement, or non-SATD. To improve our model, we optimise a key hyperparameter—the tree count of the random forest. While leaving other parameters at their default levels, this tuning is done by experimenting with the values to achieve the best balance between accuracy and computing efficiency.

3) XGBoost

EXtreme Gradient Boosting (XGBoost) implements gradient-boosted decision trees for speed and performance. Its versatility in categorisation problems has made it a popular machine learning method. XGBoost is successful for handling complicated datasets because it controls overfitting with a more regularised model formalisation than classic boosting. Using sequential tree construction, XGBoost can focus on and improve faults in SATD identification, potentially providing more detailed insights into the subtle indications of TD.

These classifiers were chosen because of their distinct and complementary capabilities to resolve the imbalances present in SATD datasets [27, 28]. Logistic Regression provides a solid foundation for binary classification, but Random Forest and XGBoost handle complicated and unbalanced data, which improves the predicted accuracy of our model. These machine learning classifiers work together to automatically classify SATD, each with its own strengths and providing a strong, adaptable, and effective system.

3.6. Evaluative optimisation

A robust and optimised classification model will be the result of our methodological approach. The assessment portion of our research will discuss the optimisation and performance indicators. Evaluative optimisation is a vital step in developing our enhanced word embedding model for automatic SATD categorisation. This step is crucial to optimising our model for classification tasks. Each phase of our evaluative optimisation procedure improves the efficacy and efficiency of the model.

1) Hyperparameter tuning

We begin our evaluative optimisation by tweaking the hyperparameters of Logistic Regression, Random Forest, and XGBoost. Classifier parameters govern learning and can dramatically impact model performance. A key hyperparameter in Random Forest is the number of decision trees (n-estimators). Adjusting this parameter can balance learning too much or too little from the training data. We experiment with hyperparameters to find the best values. Grid search or random search is used to examine various combinations. Optimisation criteria include accuracy, precision, recall, and the F1-score, which assess the SATD detection accuracy of the model.

We use k-fold cross-validation to test the flexibility and generalisability of the model. This approach repeatedly uses one subset for testing and the others for training from k data subsets. This method reduces overfitting and improves model performance estimates on unseen data. To choose the best predictive features after hyperparameter adjustment, we revisit feature selection. This stage may require further feature engineering or data transformations, using model training and testing findings. We improve the classification accuracy of the model by refining the input feature set. Evaluative optimisation iterates at each round of testing, which may cause changes in the preprocessing, feature engineering, or model setup [29]. This iterative loop continues until model performance is sufficient or the adjustment shows some improvement. We use these thorough processes in our evaluative optimisation process to fine-tune our classification system for high accuracy and resilience, making it a dependable tool for recognising SATD in software engineering.

4. Results and Discussions

We used n-gram IDF for feature extraction in our research on automated SATD classification using advanced word embedding techniques to compare it to TF-IDF and Bag-of-Words. Traditional methods are robust, but they restrict contextual subtleties of textual data in source code comments. Our approach used n-gram IDF to capture more granular textual patterns, which is necessary to identify SATD classes. We used instance hardness undersampling and a random forest classifier in all feature engineering settings to evaluate everything. We also tested a random forest–based model against an Support Vector Machine (SVM)–based model. Although SVM performs well in text categorisation, its limitations in multi-class settings prompted us to adopt a One vs. One strategy to accommodate the multi-class nature

of our SATD classification problem. Our study also re-implements and compares methods from previous works, such as those by da Silva Maldonado et al. [1], which focused on binary classification using NLP techniques and a maximum entropy classifier. To align our multi-class classification framework with their binary classification setup for effective comparison, we have specifically curated two new datasets for this purpose—each isolating design and requirement SATD from non-SATD comments.

The evaluation of our model is meticulously structured around dividing the dataset into training, validation, and testing sets, distributed across ten different projects, ensuring that each project serves as a one-time testing set and a one-time validation set in a series of ten iterations. This approach not only enhances the robustness of our model by exposing it to various data scenarios but also fine-tunes the hyperparameters to optimise performance. This comparative analysis and evaluation aims to prove the efficacy of n-gram IDF in multi-class SATD classification [30, 31], providing a brief understanding of its capabilities related to proposed methods and revealing the complexities of TD classification software development.

1) Dataset details

Table 2 provides the details of the summarised dataset used in this study.

Our work on the automated classification of SATD using sophisticated word embedding used a publicly accessible dataset from da Silva Maldonado et al. [1]. This dataset contains 62,000 Java source code comments from eleven open-source projects: Ant, ArgoUML, Columbia, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby, and SQuirrel SQL. Design, requirement, defect, test, and documentation SATD comments are included here. This dataset was adjusted to concentrate on design, requirement, and non-SATD comments to meet our study goals. Our empirical study employed a fresh dataset of 60,000 comments from this reorganisation. The imbalanced restructured dataset primarily contains non-SATD comments, with only a few containing design and requirement SATD comments. This mismatch makes SATD comment categorisation difficult and requires particular methods to avoid bias against the more common non-SATD group. This uneven nature impacts our modelling strategy by affecting undersampling and classifier tuning to avoid majority class bias. We use instance hardness undersampling and classifier adjustment to manage the skewed class distribution and achieve accurate and unbiased classification performance.

Table 2
Data on non-SATD, requirement SATD, and design SATD percentages are distributed across various projects

Project	Non-SATD	Requirement SATD	Design SATD
JEdit	98.00%	0.10%	1.90%
Hibernate	85.60%	2.20%	12.20%
JFreeChart	95.50%	0.30%	4.20%
Columbia	97.30%	0.70%	2.00%
ArgoUML	86.90%	4.40%	8.70%
JMeter	95.80%	0.30%	3.90%
JRuby	90.40%	2.30%	7.30%
Ant	97.30%	0.30%	2.30%
EMF	97.80%	0.40%	1.80%
SQuirrel	96.40%	0.70%	2.90%

2) Performance measures

To test our automated classification model for SATD utilising sophisticated word embedding, we use a rigorous assessment methodology. Our study uses Precision, Recall, and F1-score to assess SATD detection in source code comments. Using a confusion matrix, these metrics divide classification results into SATD and non-SATD. A formula that incorporates the odds of comments being design or requirement SATD against non-SATD facilitates this binary classification:

$$C_i = \{^{SATD \quad if P(i,Des)+P(i,Reg)>P(i,Non)}_{non-SATD \; Otherwise} \tag{4}$$

where C_i represents the binary classification of the source code comment i and $P(i, Des)$, $P(i, Req)$, and $P(i, Non)$ are the probabilities of the comment i being predicted as design, requirement, and non-SATD, respectively. To further refine our evaluation, we incorporate the Macro-Averaged Mean Cost-Error (MMCE), a metric that measures the accuracy of class probability predictions against actual classes:

$$MMCE = \frac{1}{3}\sum_{j=1}^{k}\frac{1}{n_j}\sum_{i=1}^{n}\left|\widetilde{y_i} - y_i\right| \tag{5}$$

Here $\widetilde{y_i}$ is the predicted probability of class k for sample i, y_i is the actual class, k is the total number of classes (three in our study: non-SATD, design SATD, requirement SATD), n is the total number of samples, and n_j is the number of samples for class k. To compare the performance of two classification models, we employ statistical tests such as the Wilcoxon Signed-Rank Test to assess the significance of differences in cost errors between models. We also utilise the Correlated Samples Case of the Vargha and Delaney $A\hat{}xy$ statistics to measure the effect size, which indicates the performance superiority between two models:

$$\widetilde{Axy} = \frac{\#(\mathrm{X<Y}) + 0.5 \times \#(\mathrm{X=Y})}{n} \tag{6}$$

where #(X < Y) is the number of instances where the cost error of model X is less than model Y, #(X = Y) is the number where the cost error is equal, and n is the total number of comparisons. The hyperparameter tuning, particularly the number of trees in our Random Forest classifier, is meticulously managed to avoid overfitting while enhancing model performance. Based on literature and empirical evidence, we adjust the number of trees from 50 to 350, observing the impact on the F1-score to determine the optimal setting. Our validation process leverages projects with the highest ratios of positive classes, ensuring a balanced evaluation of the model capabilities across varied datasets.

Table 3 organises and displays performance metrics for three text feature extraction techniques—TF-IDF, Bag-of-Words, and N-gram IDF—across various software projects, such as JEdit, JRuby, EMF, Ant, ArgoUML, JMeter, Hibernate, JFreeChart, Columbia, and SQuirrel. The effectiveness of each technique is evaluated based on four key metrics: Precision, Recall, F1-score, and MMCE. Precision measures the accuracy of predicted positive outcomes, Recall assesses how well the model identifies all relevant instances, the F1-score provides a balance between precision and recall, and MMCE evaluates the average cost errors across classifications, with lower values indicating better performance. The arrangement in Table 3 allows for a comparative analysis of the performance of each method in identifying SATD in Java source code comments across different projects, highlighting the nuances in performance that can guide further refinement and selection of feature extraction methods for specific contexts or needs.

Table 3
Metrics for precision, recall, F1, and MMCE across three techniques (TF-IDF, Bag-of-Words, n-gram IDF) for various projects

Project	TF-IDF MMCE	TF-IDF F1	TF-IDF Recall	TF-IDF Precision	Bag-of-Words MMCE	Bag-of-Words F1	Bag-of-Words Recall	Bag-of-Words Precision	N-gram IDF MMCE	N-gram IDF F1	N-gram IDF Recall	N-gram IDF Precision
JEdit	0.495	0.424	0.3	0.724	0.506	0.356	0.257	0.581	0.535	0.315	0.2	0.737
JRuby	0.437	0.829	0.795	0.867	0.417	0.833	0.815	0.852	0.388	0.831	0.81	0.853
EMF	0.519	0.481	0.404	0.594	0.557	0.276	0.213	0.392	0.493	0.45	0.309	0.829
Ant	0.531	0.352	0.287	0.456	0.506	0.311	0.231	0.472	0.518	0.317	0.204	0.71
ArgoUML	0.4	0.862	0.925	0.807	0.387	0.874	0.919	0.834	0.348	0.874	0.922	0.831
JMeter	0.461	0.754	0.736	0.773	0.463	0.703	0.662	0.751	0.434	0.771	0.73	0.817
Hibernate	0.404	0.821	0.773	0.876	0.408	0.817	0.735	0.919	0.332	0.821	0.723	0.95
JFreeChart	0.461	0.533	0.427	0.708	0.432	0.542	0.452	0.677	0.414	0.548	0.462	0.672
Columbia	0.373	0.813	0.864	0.768	0.371	0.815	0.793	0.838	0.347	0.849	0.846	0.851
SQuirrel	0.428	0.663	0.691	0.637	0.637	0.637	0.637	0.637	0.637	0.698	0.698	0.714

Table 4 provides a comprehensive evaluation of four machine learning classifiers XGBoost, Logistic Regression, Support Vector Machine (SVM), and Random Forest (RF)—in various software development projects. Each classifier is assessed based on three key performance metrics: Precision, Recall, and F1-Score. These metrics are essential for understanding how effectively each classifier identifies and categorises SATD within Java source code comments. Precision assesses the ability of the classifier to minimise false positives by finding true positives among all positive identifications. In contrast, recall evaluates the capacity of the classifier to detect all true positives and minimise false negatives. The F1-score balances precision and recall to quantify the ability of the classifier to identify true positives without misclassifying negatives. Table 4 illustrates the classifier performance based on these criteria for each project, from JRuby to SQuirrel. This allows us to compare classifier performance across projects and classifiers, revealing how project type and SATD variables affect classifier performance. This extensive research helps determine which classifiers work better under different settings and helps choose models for particular software applications.

Our work on automatic SATD categorisation using sophisticated word embedding faced internal and external constraints that affected our approach and results. Due to the lack of implementation details, da Silva Maldonado et al.'s [1] categorisation model had to be reimplemented internally. We minimised this threat by following their publication methods and utilising the same dataset, which led to similar findings as the original work. This reimplementation, while rigorous, carries the inherent risk that our model may not exactly replicate every aspect of the original model. Another internal consideration was that we used the n-gram extraction tool Ngweight to build our n-gram dictionary. The sensitivity of the tool to parameter changes means that any variation in its configuration can significantly alter the output dictionary, potentially affecting the classification results. Additionally, our application of instance hardness undersampling can also lead to variations in the outcomes, as different algorithms or threshold adjustments during the undersampling process can result in the selection of different data subsets for training.

From an external perspective, the generalisability of our findings may be limited. The dataset obtained from da Silva Maldonado et al. [1] comprises Java source code comments from ten open-source projects from different domains. This specificity raises concerns about how well our model will perform on projects from other domains, projects written in other programming languages, or projects not written in English. Furthermore, projects with little or no source code comments may not be suitable for our approach, as it relies on textual analysis of comments

Table 4
Performance of each classifier across all metrics for each project

Metric	JRuby	JEdit	Hibernate	EMF	Ant	JMeter	Columbia	ArgoUML	JFreeChart	SQuirrel
XG Precision	0.89	0.95	0.92	0.91	0.89	0.92	0.88	0.86	0.95	0.9
XG Recall	0.79	0.1	0.78	0.39	0.18	0.55	0.72	0.9	0.49	0.63
XG F1	0.84	0.15	0.85	0.48	0.25	0.69	0.79	0.88	0.56	0.68
LR Precision	0.88	0.91	0.86	0.89	0.9	0.89	0.83	0.83	0.9	0.85
LR Recall	0.77	0.15	0.73	0.32	0.15	0.6	0.69	0.85	0.43	0.58
LR F1	0.82	0.2	0.79	0.4	0.22	0.72	0.75	0.84	0.5	0.64
SVM Precision	0.89	0.91	0.93	0.87	0.5	0.87	0.91	0.83	0.98	0.94
SVM Recall	0.44	0.05	0.71	0.21	0.13	0.51	0.66	0.87	0.26	0.5
SVM F1	0.59	0.09	0.8	0.34	0.2	0.65	0.76	0.85	0.41	0.66
RF Precision	0.85	0.74	0.9	0.83	0.71	0.82	0.85	0.83	0.67	0.71
RF Recall	0.81	0.2	0.72	0.31	0.2	0.73	0.85	0.92	0.46	0.68
RF F1	0.83	0.32	0.82	0.45	0.32	0.77	0.85	0.87	0.55	0.7

to detect SATD. Our research also examined whether undersampling is beneficial for unbalanced datasets. We tested our classification model with and without undersampling to address this concern. Our empirical studies divided the dataset into training, validation, and testing sets across numerous rounds and showed that undersampling can improve model performance. Undersampling improved the F1-score and MMCE findings, suggesting it helps manage the skewed class distribution of SATD data.

5. Conclusion and Future Work

Finally, our work on automated SATD categorisation utilising advanced word embedding methods showed encouraging results, especially with n-gram IDF in feature dictionaries for machine learning models. We implemented and compared the classification algorithms Random Forest, SVM, Logistic Regression, and XGBoost, and found that Random Forest and XGBoost performed well in terms of accuracy, recall, and F1-score. Our tests showed that instance hardness undersampling can solve the SATD classification problems for unbalanced datasets.

Several intriguing options exist for further work. To improve model generalisability, the dataset might be expanded to include more projects from other programming languages and fields. This extension can further test the stability of the model across different coding styles and projects. Alternative machine learning and ensemble approaches may increase classification accuracy. Further research can also explore hybrid models that combine the strengths of various classification techniques, as recent works have begun exploring such directions using generative AI and large language models [32–34]. Furthermore, addressing external validity concerns by applying the model to commercial software projects and projects written in languages other than Java could provide insights into its practical utility in a wider array of development environments. Finally, the continuous refinement of the undersampling techniques and parameter tuning based on newly emerging data characteristics can further enhance the effectiveness and efficiency of the model in real-world applications.

Ethical Statement

This study does not contain any studies with human or animal subjects performed by any of the authors.

Conflicts of Interest

The authors declare that they have no conflicts of interest to this work.

Data Availability Statement

Data are available on request from the corresponding author upon reasonable request.

Author Contribution Statement

Satya Mohan Chowdary Gorripati: Methodology, Software, Formal analysis, Investigation, Resources, Data curation, Writing – original draft. **Ali Altalbe:** Writing – review & editing. **Prasanna Kumar Rangarajan:** Conceptualization, Validation, Visualization, Supervision, Project administration.

References

[1] da Silva Maldonado, E., Shihab, E., & Tsantalis, N. (2017). Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, *43*(11), 1044-1062. https://doi.org/10.1109/TSE.2017.2654244

[2] Potdar, A., & Shihab, E. (2014). An exploratory study on self-admitted technical debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*, 91-100. https://doi.org/10.1109/ICSME.2014.31

[3] Huang, Q., Shihab, E., Xia, X., Lo, D., & Li, S. (2018). Identifying self-admitted technical debt in open source projects using text mining. Empirical Software Engineering, 23(1), 418-451. https://doi.org/10.1007/s10664-017-9522-4

[4] Yu, D., Wang, L., Chen, X., & Chen, J. (2021). Using BiLSTM with attention mechanism to automatically detect self-admitted technical debt. *Frontiers of Computer Science*, *15*(4), 154208. https://doi.org/10.1007/s11704-020-9281-z

[5] Ren, X., Xing, Z., Xia, X., Lo, D., Wang, X., & Grundy, J. (2019). Neural network-based detection of self-admitted technical debt: From performance to explainability. *ACM Transactions on Software Engineering and Methodology, 28*(3), 15. https://doi.org/10.1145/3324916

[6] Wang, X., Liu, J., Li, L., Chen, X., Liu, X., & Wu, H. (2021). Detecting and explaining self-admitted technical debts with attention-based neural networks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 871-882. https://doi.org/10.1145/3324884.3416583

[7] Chen, X., Yu, D., Fan, X., Wang, L., & Chen, J. (2022). Multiclass classification for self-admitted technical debt based on XGBoost. *IEEE Transactions on Reliability, 71*(3), 1309-1324. https://doi.org/10.1109/TR.2021.3087864

[8] Thabtah, F., Hammoud, S., Kamalov, F., & Gonsalves, A. (2020). Data imbalance in classification: *Experimental evaluation. Information Sciences*, *513*, 429-441. https://doi.org/10.1016/j.ins.2019.11.004

[9] Guo, Z., Liu, S., Liu, J., Li, Y., Chen, L., Lu, H., & Zhou, Y. (2021). How far have we progressed in identifying self-admitted technical debts? A comprehensive empirical study. *ACM Transactions on Software Engineering and Methodology*, *30*(4), 45. https://doi.org/10.1145/3447247

[10] Behutiye, W. N., Rodríguez, P., Oivo, M., & Tosun, A. (2017). Analyzing the concept of technical debt in the context of agile software development: A systematic literature review. *Information and Software Technology*, *82*, 139-158. https://doi.org/10.1016/j.infsof.2016.10.004

[11] Freire, S., Rios, N., Gutierrez, B., Torres, D., Mendonça, M., Izurieta, C., ..., & Spínola, R. O. (2020). Surveying software practitioners on technical debt payment practices and reasons for not paying off debt items. In *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, 210-219. https://doi.org/10.1145/3383219.3383241

[12] de Freitas Farias, M. A., de Mendonça Neto, M. G., Kalinowski, M., & Spínola, R. O. (2020). Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary. Information and Software Technology, 121, 106270. https://doi.org/10.1016/j.infsof.2020.106270

[13] Yu, Z., Fahid, F. M., Tu, H., & Menzies, T. (2022). Identifying self-admitted technical debts with jitterbug: A two-step approach. *IEEE Transactions on Software Engineering, 48*(5), 1676-1691. https://doi.org/10.1109/TSE.2020.3031401

[14] Wattanakriengkrai, S., Maipradit, R., Hata, H., Choetkiertikul, M., Sunetnanta, T., & Matsumoto, K. (2018). Identifying design and requirement self-admitted technical debt using n-gram idf. In *2018 9th International Workshop on Empirical Software Engineering in Practice*, 7-12. https://doi.org/10.1109/IWESEP.2018.00010

[15] Zhao, K., Xu, Z., Zhang, T., Tang, Y., & Yan, M. (2021). Simplified deep forest model based just-in-time defect prediction for android mobile apps. *IEEE Transactions on Reliability, 70*(2), 848-859. https://doi.org/10.1109/TR.2021.3060937

[16] Murillo, M. I., López, G., Spínola, R., Guzmán, J., Rios, N., & Pacheco, A. (2023). Identification and management of technical debt: A systematic mapping study update. *Journal of Software Engineering Research and Development, 11*(1), 8. https://doi.org/10.5753/jserd.2023.2671

[17] Flisar, J., & Podgorelec, V. (2018). Enhanced feature selection using word embeddings for self-admitted technical debt identification. In *2018 44*th *Euromicro Conference on Software Engineering and Advanced Applications*, 230-233. https://doi.org/10.1109/SEAA.2018.00045

[18] Yu, J., Zhao, K., Liu, J., Liu, X., Xu, Z., & Wang, X. (2022). Exploiting gated graph neural network for detecting and explaining self-admitted technical debts. *Journal of Systems and Software*, 187, 111219. https://doi.org/10.1016/j.jss.2022.111219

[19] Sliger, M., & Broderick, S. (2008). *The software project manager's bridge to agility* (1st Ed.). USA: Addison-Wesley Professional. https://dl.acm.org/doi/abs/10.5555/1406474

[20] Page, A. M. (2019). *Technical debt: The cost of doing nothing*. Master's Thesis, Massachusetts Institute of Technology.

[21] Li, Y., Soliman, M., Avgeriou, P., & Somers, L. (2023). Self-admitted technical debt in the embedded systems industry: An exploratory case study. *IEEE Transactions on Software Engineering, 49*(4), 2545-2565. https://doi.org/10.1109/TSE.2022.3224378

[22] Yang, Y., Zhang, B., Guo, D., Du, H., Xiong, Z., Niyato, D., & Han, Z. (2024). Generative AI for secure and privacy-preserving mobile crowdsensing. *IEEE Wireless Communications, 31*(6), 29-38. https://doi.org/10.1109/MWC.004.2400017

[23] Yang, Z., Keung, J., Kabir, M. A., Yu, X., Tang, Y., Zhang, M., & Feng, S. (2021). AComNN: Attention enhanced Compound Neural Network for financial time-series forecasting with cross-regional features. Applied Soft Computing, 111, 107649. https://doi.org/10.1016/j.asoc.2021.107649

[24] Terdchanakul, P., Hata, H., Phannachitta, P., & Matsumoto, K. (2017). Bug or not? Bug report classification using N-gram IDF. In *2017 IEEE International Conference on Software Maintenance and Evolution*, 534-538. https://doi.org/10.1109/ICSME.2017.14

[25] Maipradit, R., Hata, H., & Matsumoto, K. (2019). Sentiment classification using N-gram inverse document frequency and automated machine learning. *IEEE Software*, *36*(5), 65-70. https://doi.org/10.1109/MS.2019.2919573

[26] Sabbah, A. F., & Hanani, A. A. (2023). Self-admitted technical debt classification using natural language processing word embeddings. *International Journal of Electrical and Computer Engineering, 13*(2), 2142-2155. http://doi.org/10.11591/ijece.v13i2.pp2142-2155

[27] Li, Y., Soliman, M., & Avgeriou, P. (2023). Automatic identification of self-admitted technical debt from four different sources. *Empirical Software Engineering*, *28*(3), 65. https://doi.org/10.1007/s10664-023-10297-9

[28] Aiken, W., Mvula, P. K., Branco, P., Jourdan, G. V., Sabetzadeh, M., & Viktor, H. (2023). Measuring improvement of F1-scores in detection of self-admitted technical debt. In *2023 ACM/IEEE International Conference on Technical Debt*, 37-41. https://doi.org/10.1109/TechDebt59074.2023.00011

[29] Satya Mohan Chowdary, G., & Prasanna Kumar, R. (2023). Automated identification and prioritization of self-admitted technical debt using NLP word embeddings. In *Proceedings of the IEEE International Conference on Software Systems and Applications*, 963–971. https://doi.org/10.1109/ICSSAS57918.2023.10331839

[30] Mastropaolo, A., Di Penta, M., & Bavota, G. (2023). Towards automatically addressing self-admitted technical debt: How far are we? In *2023 38th IEEE/ACM International Conference on Automated Software Engineering*, 585-597. https://doi.org/10.1109/ASE56229.2023.00103

[31] Yang, Y., Du, H., Sun, G., Xiong, Z., Niyato, D., & Han, Z. (2025). *Exploring equilibrium strategies in network games with generative AI. IEEE Network*, *39*(5), 191-200. https://doi.org/10.1109/MNET.2024.3521887

[32] Sheikhaei, M. S., Tian, Y., Wang, S., & Xu, B. (2025). Understanding the effectiveness of LLMs in automated self-admitted technical debt repayment. *arXiv Preprint*:2501.09888. https://doi.org/10.48550/arXiv.2501.09888

[33] Gurusamy, B. M., Rangarajan, P. K., & Altalbe, A. (2024). Whale-optimized LSTM networks for enhanced automatic text summarization. *Frontiers in Artificial Intelligence*, *7*, Article 1399168. https://doi.org/10.3389/frai.2024.1399168

[34] Bharathi Mohan, G., Prasanna Kumar, R., Parathasarathy, S., Aravind, S., Hanish, K. B., & Pavithria, G. (2023). Text Summarization for Big Data Analytics: A Comprehensive Review of GPT 2 and BERT Approaches. In A. J. Sanyal, S. K. Das, & D. U. K. M. Kumar (Eds.), *Internet of Things Data Analytics for Internet of Things Infrastructure* (pp. 247–264). Springer. https://doi.org/10.1007/978-3-031-33808-3_14

How to Cite: Gorripati, S. M. C., Altalbe, A., & Rangarajan, P. K. (2025). Automated Classification of Self-Admitted Technical Debt Using Advanced Word Embedding Techniques. *Journal of Computational and Cognitive Engineering*. https://doi.org/10.47852/bonviewJCCE52025976