Empirical Software Engineering (2026) 31:66
https://doi.org/10.1007/s10664-026-10807-5

Automated detection of algorithm debt in deep learning frameworks: an empirical study

Emmanuel Iko-Ojo Simon[1] · **Chirath Hettiarachchi**[1] · **Alex Potanin**[1] · **Hanna Suominen**[1,2,3] · **Fatemeh Fard**[4]

Received: 26 May 2025 / Accepted: 7 January 2026 / Published online: 7 February 2026

Abstract
Expedient design choices in software development can lead to Technical Debt (TD), with development teams documenting such decisions as Self-Admitted TD (SATD). Algorithm Debt (AD) is a type of TD resulting from the suboptimal implementation of algorithms, which impacts system performance. Given the impact of AD, its automated detection is crucial in Deep Learning (DL) frameworks due to their complexity and evolution. Early detection of AD in DL frameworks can help mitigate model degradation and scalability issues. Despite previous studies on the automated detection of TD from SATD using Machine Learning (ML)/DL models, research on AD detection in DL frameworks remains underexplored. In this study, we empirically investigated the performance of ML/DL models for the automated detection of AD using a dataset of 38, 881 SATD comments from seven DL frameworks. We trained, evaluated, and tested ML/DL models, used embeddings from both DL and large language models, and explored an approach to enrich the dataset with handcrafted features based on AD-related keywords. Our findings reveal that AD is frequently misclassified as Design or Implementation Debt. Logistic Regression (an ML model) with Custom AD Features, achieved an F1-score of 54% for AD, outperforming other ML/DL models (42% to 52%), highlighting the importance of tailored feature engineering. Our research advances automated AD detection in DL frameworks by providing insights into the strengths and limitations of ML/DL models, serving as a first step to guide future tool development. This could help developers using DL frameworks to identify AD issues during development, thereby enhancing system reliability by mitigating model degradation and scalability challenges.

Keywords Technical debt · Algorithm debt · Machine learning · Deep learning · Software engineering

Communicated by: Shinpei Hayashi and Shane McIntosh.

Extended author information available on the last page of the article

1 Introduction

Conventional Machine Learning (ML) and Deep Learning (DL) technologies have advanced in their applications in various sectors, including healthcare, tourism, finance, businesses, and education (Khan 2024; Williamson 2024; Ghasemi et al. 2025). These advancements include the emergence of applications such as GPT-4, voice assistants, and computer vision technologies including self-driving cars (Bhatia et al. 2024). Just like traditional software systems, ML/DL systems are not immune to Technical Debt (TD) (i.e., the long-term cost of expedient design or implementation decisions) (Moreschini et al. 2024). Traditional software refers to software systems that follow standard programming paradigms and do not incorporate any ML/DL algorithms or components, while ML/DL systems are systems that integrate ML/DL models or algorithms with other non-ML components (Bhatia et al. 2024). These additional components introduce new sources of TD, such as model retraining, data dependency, and parameter tuning in ML/DL systems (Liu et al. 2020).

Among the various TD types, Algorithm Debt (AD) has emerged as particularly important in DL frameworks. AD arises from sub-optimal implementations of algorithms or logic that degrade system performance (Liu et al. 2020; Simon et al. 2023). Unlike other TD types which are structural or syntactic in nature, AD directly affects the computational core of ML/DL systems, leading to inconsistent results, poor scalability, and model degradation if left unaddressed (Liu et al. 2020; Vidoni 2021). For instance, an inefficient algorithm used for model training can lead to prolonged training times or limited scalability, ultimately affecting the performance of the model when deployed. The algorithmic complexity, stochastic training processes, and hardware constraints inherent in DL frameworks make AD particularly challenging to detect and optimise (Sculley et al. 2015).

Given these challenges, identifying AD through developer-written comments, known as Self-Admitted TD (SATD) offers a promising direction. SATD, where developers explicitly document trade-offs in code comments, is a valuable approach for studying TD (Potdar and Shihab 2014), providing a practical way for categorising TD types in real-world systems. However, some TD types can be miscategorised with one another due to their conceptual similarities and linguistic patterns. For instance AD can be confused with Design and Implementation Debt in SATD comments. For example, AD is reflected in comments such as: `Is there a faster way to do pooling in the channel-first case?` (from Caffe), highlighting inefficiencies in algorithmic logic. By contrast, Design Debt is captured in comments like `This kernel should be moved into Eigen and vectorized?` (from TensorFlow), which concern structural choices. While Implementation Debt is reflected in comments such as: `We temporarily support indexing a zero-dim tensor as if it were a one-dim tensor to better maintain backwards compatibility` (from PyTorch). These examples drawn from our analysed dataset (Liu et al. 2020), demonstrate that AD focuses on algorithmic inefficiencies, distinct from the structural concerns of Design Debt and the coding flaws of Implementation Debt. These distinctions are especially important in DL systems, where SATD tends to be more complex and varied than in traditional software.

Listing 1 Example of SATD signifying AD on Line 1 in an open-source system. The excerpt is from Caffe's PoolingLayer::Forward_cpu function, highlighting the inefficient loop structure

```
1  // TODO(Yangqing): Is there a faster way to do pooling in
       the channel-first case?
2  caffe_set(top_count, Dtype(-FLT_MAX), top_data);
3     // The main loop
4     for (int n = 0; n < bottom[0]->num(); ++n) {
5       for (int c = 0; c < channels_; ++c) {
6         for (int ph = 0; ph < pooled_height_; ++ph) {
7           for (int pw = 0; pw < pooled_width_; ++pw) {
8             int hstart = ph * stride_h_ - pad_h_;
9             int wstart = pw * stride_w_ - pad_w_;
10            int hend = min(hstart + kernel_h_, height_);
11            int wend = min(wstart + kernel_w_, width_);
12            hstart = max(hstart, 0);
```

Although SATD exists in both traditional and DL systems, it differs in scale and nature (OBrien et al. 2022; Bavota and Russo 2016), with the median SATD in DL systems being twice that of traditional software (Bhatia et al. 2024). The additional components and ML operations (i.e., practices and tools for managing ML model lifecycle), make SATD in DL unique (Sculley et al. 2015; Bhatia et al. 2024). For example, in Listing 1, the TD originates from an open-source DL framework (Caffe). The deferred pooling optimisation creates AD, as the current channel-first implementation may lead to inefficiency and poor scalability (Ayyagari et al. 2022). This example illustrates how AD can impact performance and scalability in DL frameworks, highlighting the need for its early detection in DL frameworks, to help DL developers identify inefficiencies and ensure systems scale effectively and prevent long-term performance degradation. This highlights the need to evaluate whether existing SATD detection approaches are effective in identifying AD in DL frameworks.

While prior studies have explored the automated detection of TD in traditional software using SATD, a gap exists in addressing AD in DL frameworks, given that ML/DL SATD differs in nature. These studies have used NLP (da Silva Maldonado et al. 2017; Sabbah and Hanani 2023), n-gram Inverse Document Frequency (IDF) (Wattanakriengkrai et al. 2019), Generative Adversarial Network (GAN)-based neural networks (Yu et al. 2023), and ML/DL detection of SATD from different sources using Random Forest (RF), Logistic Regression (LR), Support Vector Machine (SVM), Convolutional Neural Networks (CNNs), and GANs (Li et al. 2023a). This research has been extended to R programming (Sharma et al. 2022) and Blockchain (Pinna et al. 2023), with works on Large Language Models (LLMs) through prompt tuning (Yu et al. 2024) and data argumentation (Sutoyo et al. 2024; Li et al. 2025). However, these studies did not target DL frameworks or focus on AD specifically. The only study that included AD, conducted by Sharma et al. (2022) and focused on R programming, reported a low F1 score of 31% for AD when using the RoBERTa model, suggesting room for improvement. These limitations collectively contribute to an underexplored research gap in the automated detection of AD in DL frameworks.

To address this gap, the aim of our study is to empirically investigate the performance of ML/DL models with different feature extraction techniques. To achieve this aim, we focus on two Research Questions (RQs): **RQ1**: Which ML models and feature extraction methods are effective in detecting AD in DL frameworks? and **RQ2:** How do DL models perform in detecting AD from SATD in DL frameworks?

We followed the details of the Stage 1 Protocol of our Registered Report (Simon et al. 2024) to address the RQs, which provides methodological transparency and ensures reproducibility. We trained, validated, and tested different ML/DL models on a domain-specific dataset. The dataset was curated by Liu et al. (2020) from seven open-source DL frameworks: TensorFlow, Keras, DeepLearning4j (DL4J), Caffe, PyTorch, MXNet, and Microsoft Cognitive Toolkit (CNTK) hosted on GitHub. We used this dataset because it was used to uncover AD, thereby having AD-related insights and also has an inter-rater agreement of over 85%.

The ML models we trained are the SVM (Joachims 1998), LR (Hilbe 2009), RF (Breiman 2001), using different feature extraction techniques: Term Frequency-IDF (TF-IDF), Count Vectoriser, Hash Vectoriser. For the DL models, we used transformer-based models which include RoBERTa (Robustly Optimised BERT-Pretraining approach) (Liu et al. 2021b) and ALBERT (A Lite BERT) (Lan et al. 2019). We also extracted embeddings from pre-trained versions of these models that were fine-tuned on the SATD dataset. With the advancements in LLMs, we explored INSTRUCTOR (Su et al. 2023) and VoyageAI[1] to extract embeddings from the SATD comments in the dataset. These LLMs have achieved the state-of-the-art performance in tasks such as automatic code generation, bug detection, and natural language tasks (Lemieux et al. 2023; Ozkaya 2023; Liu et al. 2024; Sallou et al. 2024). In addition, we enriched the dataset by experimenting with a technique that combined LR (an ML model) with Custom AD Features (i.e., additional keywords related to AD) to allow the models to better capture the nuances of AD. We conducted statistical significance testing for all the models, to validate the differences in their performances.

Our results show that limitations exist in the detection of AD in DL frameworks using ML/DL models. The combination of LR (a ML model) and Custom AD Features achieved the highest F1-score of 54% outperforming other ML/DL models. Also, we found that AD was often misclassified with Design and Implementation Debt, further highlighting the difficulty in detecting AD in DL frameworks. The findings from this research offer insights into the effectiveness of ML/DL models in detecting AD in DL frameworks. These insights can help developers create tools to flag and address AD early while coding, facilitating AD research and mitigating issues that can hinder model scalability and degradation. Consequently, this research adds to the body of knowledge on AD detection in DL frameworks, serving as a first step for building automated tools for AD detection. The contributions of our work are as follows:

- *Empirical Evaluation of AD Detection.* We empirically evaluated the performance of ML/DL models and embeddings from DL models and LLMs for the detection of AD in DL frameworks. Our analysis provides insights into the effectiveness of these models and their limitations.
- *Investigation of Feature Engineering for AD Detection.* By using custom features specific to AD in DL frameworks, we investigated the effectiveness of tailored feature engineering for AD detection. Our findings suggest that using AD-related features improves the detection performance of ML/DL models.
- *Proposal for Future AD Detection.* Based on our empirical findings, we proposed recommendations for improving AD detection in DL frameworks. These include incorporating information from code artifacts, refining dataset labelling strategies, and use of hybrid models to enhance model performance.

[1] https://www.voyageai.com/

The remainder of this paper is structured as follows: In Sect. 2 we review related work on TD, SATD detection, TD in ML/DL systems and the use of ML/DL for detection of SATD. In Sect. 3, we provide details of the methodology used in our study and the Research Questions (RQs) that we addressed. We present the results of our empirical study in Sect. 4. In Sect. 5 we analyse the results. In Sect. 6 we discuss the implications of our findings while in Sect. 7 we discuss the threats to validity and the steps we took to mitigate them. Finally, in Sect. 8, we provide a conclusion of our work.

2 Related Work

In this Section, we describe works related to previous studies on TD, SATD detection in traditional software engineering, and TD in ML/DL systems.

2.1 Background on TD and SATD

Cunningham (1992) introduced the concept of TD as "quick and dirty" work in software design that results in code that is "not quite-right". Since then, studies have explored and expanded this concept, leading to a variety of definitions. For instance, Rios et al. (2018) defined TD as the costs incurred in software maintenance due to tasks postponed during its development. Broadly, TD symbolises the compromise between delivering fast and producing high-quality code (Avgeriou et al. 2016); however, it is considered an abstract concept with trade-offs between optimal software quality and project deadlines being difficult to measure quantitatively (Bhatia et al. 2024).

To address the abstract nature of TD, Potdar and Shihab (2014) introduced the notion of SATD, which provides a more tangible measure of TD. SATD are comments made by the development team acknowledging the presence of TD. In their seminal work, they found that SATD was widespread in large open-source projects, with up to 31% present in files containing such comments. This was followed by a work by Bavota and Russo (2016), that conducted a comprehensive study of SATD across various software projects, leading to the creation of a taxonomy for SATD in traditional software to help in the management of TD.

To explore the management and repayment of TD, Peruma et al. (2022) conducted an exploratory study on the relationship between TD and refactoring. They identified five areas where developers often need assistance with refactoring. Also, Mastropaolo et al. (2023) explored the automated repayment of TD through neural-based generative models, where they used a dataset of 5, 039 SATD removal instances from open-source projects, to perform their experiments. Similarly, Lenarduzzi et al. (2019) performed a systematic literature review on TD and proposed different approaches for prioritising TD, offering insights into strategies for managing TD in software projects.

Several studies have also been carried out to study different aspects of TD and SATD in different domains. Azuma et al. (2022) categorised SATD within Dockerfiles through manual classification and found that 3.0% of comments in Dockerfiles represent SATD. They categorised SATDs into five main classes and eleven sub-classes. In a recent work, Pinna et al. (2023) investigated SATD in open-source blockchain projects, using NLP for comment classification in ten selected projects. Their results revealed that Design Debt surpasses Requirement Debt in Blockchain projects. Ebrahimi et al. (2023) conducted an

exploratory study on SATD in smart contracts i.e., digital contracts that automatically execute and enforce terms of the agreement in blockchain technology in Blockchain using quantitative and qualitative methods to assess their prevalence. They proposed a taxonomy of SATD consisting of six major and 26 minor categories of SATD.

Studies have also investigated the prevalence of SATD in software projects. Xiao et al. (2024), conducted a large-scale study on SATD comments to investigate the prevalence of SATD clones. They observed that SATD clones are a more prevalent phenomenon in build systems than in source code. Given that the available SATD datasets for automated detection of different SATD types are unbalanced, Sutoyo and Capiluppi (2024) augmented different SATD datasets which includes SATD from different source code comments, issue trackers, pull requests, and commit messages to provide a richer source of labelled data for SATD detection. Rantala et al. (2024) investigated the relationship between keyword labelled SATD from source code comments and reports from SonarQube to find which metrics and issues are related to keyword-labelled SATD introduction and removal and whether keyword labelled SATD is related to those issues that address them.

Similar to these studies, we leveraged SATD as an indicator of TD to explore AD. While TD types: Design, Defect, and Documentation Debt have been studied, AD remains underexplored, particularly in DL frameworks. Our work thus contributes to existing research on TD by investigating AD in DL frameworks.

2.2 TD and SATD in ML/DL Systems

While TD and SATD have been studied in traditional software systems, their implications in ML/DL systems are unique, given that DL introduce additional TD risks. Now, we examine prior works addressing TD in ML/DL systems. In this research, ML/DL systems refer to systems that contain any implementation of DL algorithm or framework. Sculley et al. (2015) conducted pioneering research into TD risk factors specific to ML systems. They stated that "ML systems have a special capacity for incurring TD because they have all of the maintenance problems of traditional code plus an additional set of ML-specific issues". They introduced 20 TD types in ML systems including Data, Model, and Reproducibility Debt.

In another work, Liu et al. (2020) examined the prevalence of SATD in DL frameworks and found that Design, Defect, and Documentation Debt are the most prevalent SATD in DL frameworks. Additionally, this work uncovered two new TD types in DL frameworks: Algorithm and Compatibility Debt. As a follow-up, Liu et al. (2021a) investigated the introduction and removal of SATD in DL frameworks, and found that Design Debt is introduced the most along the development process. As for the removal of TD, they found that Requirement Debt is removed the most, with Design Debt as the fastest to be removed.

Tang et al. (2021) explored ML system refactorings and correlated them to ML-specific TD. They introduced new ML-specific refactorings and TD categories like Duplicate Model Code, Model Code Comprehension, and Custom Data Types Debt. In another study by OBrien et al. (2022), the authors investigated different types of SATD in ML software and their distributions in different components of ML pipeline stages. Recently, Bhatia et al. (2024) investigated the occurrence of SATD in ML code through an empirical study by manually analysing open-source ML projects across five domains. They found that ML pipeline components for data preprocessing and model generation logic are more susceptible to TD than other components.

Recently, in a work by Pepe et al. (2024), they analysed SATD in DL systems and categorised DL-specific SATD into categories including DL models, technology, and suboptimal DL processes such as model usage or configuration. Also, Ximenes (2024) identified a list of potential factors contributing to TD in the source code of ML/DL systems. They identified data processing as the most critical factor. They also identified issues related to model creation and training as issues leading to TD in ML systems. Sklavenitis and Kalles (2024), in their work to identify TD types in ML systems, conducted a survey and categorised TD types in ML systems into AD, architecture, Code and Configuration Debt. Recupito et al. (2024) further investigated ML-specific TD to address issues related to managing ML-code and Architecture Debt in ML systems through the use of a survey.

From the above, we observe the unique nature of TD in ML/DL systems. Our work is closely related to that of Liu et al. (2020), that uncovered AD in DL frameworks. While their work characterised AD in DL frameworks, our focus is to advance its automated detection using ML/DL models and SATD comments to mitigate its consequences.

2.3 ML/DL for SATD Detection

Since the first study on SATD (Potdar and Shihab 2014) various approaches have been proposed to automatically identify SATD in traditional software engineering. For example, da Silva Maldonado et al. (2017) proposed an approach to detect SATD in ten open-source projects using the NLP Max Entropy and classified SATD into five types and provided the dataset for future research. They achieved an average F1-score of 62% for Design and 43% for Requirements Debt. Wattanakriengkrai et al. (2019) used n-gram IDF techniques to categorise SATD comments into three categories: Design, Requirements, and non-SATD. They achieved an average F1-score of 64% by using the RF model. Furthermore, Zampetti et al. (2020) created a multi-level classifier using a dataset on SATD removal. Using a CNN trained on embeddings, and achieved an average precision of 55% and recall of 57%.

AlOmar et al. (2022), proposed SATDBailiff, a tool that uses an existing state-of-the-art SATD detection tool, to identify SATD in methods and track their lifespan. Sharma et al. (2022) studied how different classifiers (ME, SVM and Pretrained Language Models) perform in classifying SATD in R programming. Their classifiers achieved macro-averaged F1-scores of between 42% – 56%. Furthermore, Li et al. (2023a) proposed a method for the automated detection of SATD from issue tracking systems, pull requests, commit messages, and source code comments using LR, SVM, RF, and Text-CNN and achieved an average F1-score of 61%. Consequently, Li et al. (2023b) conducted a second study with the goal of determining correlations between SATD items from various sources, such as commit messages and code comments. They evaluated different methods for the automated detection of SATD and obtained an average F1-score of 83% and consequently provided insights into 26 types of relations.

Yu et al. (2023) proposed a DL approach that used a generative adversarial network to improve the accuracy of classifying various SATD types. Their model combined code snippets and natural language information to detect different SATD instances. Sabbah and Hanani (2023), explored different models such as SVM, RF, Naive Bayes, and CNN to identify SATD from code comments or commits. They achieved an average F1-score of

82% and 84% for RF and CNN respectively. Also, Pinna et al. (2023), investigated SATD in Blockchain, using the Stanford classifier. They obtained an F1-score of 48% and 54% for Design and Requirements Debt.

The use of LLMs has recently been proposed for SATD detection. The work by Sheikhaei et al. (2024), investigated the performance improvement obtained by using the Flan-T5 family of LLMs. Their findings indicate that for SATD identification tasks, fine-tuned LLMs outperform the best-performing non-LLM baseline, specifically the CNN model, achieving an improvement in F1-score ranging from 4.4% to 7.2%. Also, Yu et al. (2024) proposed a method of using prompt tuning to identify SATD, while Sutoyo et al. (2024) in their work used DL and data augmentation for SATD detection. In another recent work, Li et al. (2025) evaluated the effectiveness of ChatGPT for SATD classification and compared the performance to DL models (CNN, Long short-term memory (LSTM), and Transformers). They found that ChatGPT outperformed the DL models in terms of F1-score.

Although various approaches have been proposed for automated detection of SATD using ML/DL models, these approaches are mainly for traditional software using the dataset curated by da Silva Maldonado et al. (2017), in Blockchain, or R. However, in DL frameworks, AD is uniquely influenced by the inefficiencies in the implementation of the algorithm. Also, AD was not included in most of the TD types that were identified. Our work is closely related to the work by Sharma et al. (2022); while they focused on different TD types in R programming domain, our focus is on AD in DL frameworks.

3 Methodology

To address RQ1 from Sect. 1, we compared multiple ML models with different feature extraction methods to assess their effectiveness in detecting AD. Similarly, for RQ2, we evaluated the performance of DL models, their embeddings, and LLMs against the ML baselines. This design allowed us to compare approaches and highlight the role of feature representations and model architectures in the detection of AD. In this section, we describe the methodology (Fig. 1) that we followed to answer these questions.

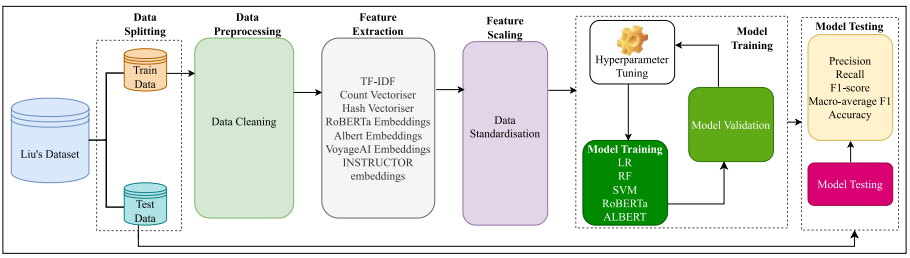

Fig. 1 The general methodology for training and testing the ML/DL models. The flowchart shows the steps we followed including data splitting, cleaning, feature extraction and scaling. It also displays the model training, validation, and testing stages

3.1 Dataset

We used an updated version of the dataset originally curated by Liu et al. (2020), which contains SATD comments from seven widely used DL frameworks. The frameworks are TensorFlow,[2] Keras,[3] DL4J,[4] Caffe,[5] PyTorch,[6] MXNet,[7] and CNTK.[8] We contacted the authors and received an updated version of the dataset that includes additional SATD instances from the 7, 159 SATD instances in the original dataset. This update provides more data (i.e., now containing a class for comments not belonging to any TD type), allowing an extensive and diverse coverage of SATD in DL frameworks.

The comments in the dataset were labelled by experts in TD and an independent PhD student in Software Engineering (Liu et al. 2020). The annotators employed a card-sorting approach in three iterations to classify the comments using the TD categories defined by Maldonado and Shihab (2015). When a comment was ambiguous, disagreements were resolved through discussion until a consensus was reached. For comments that could reasonably fit into more than one category, the annotators assigned the category reflecting the primary TD type. The reliability of the dataset is supported by a high inter-rater agreement (Kappa Cohen 1960 greater than 85%), as reported by the original authors, confirming that the labelling process was consistent. Using this dataset reduces the chances of personal bias in labelling new data since this was the research that uncovered AD.

Beyond the validity of this dataset, it was the first to uncover AD. Existing datasets either focus exclusively on general-purpose software systems or do not contain any instances of AD, making them less suitable for capturing the unique linguistic characteristics of the DL frameworks. For instance, the dataset by Sharma et al. (2022) that contains AD is specific to the R programming domain. For effective detection of AD in DL frameworks, it is important to use a dataset that aligns with this domain. Consequently, the dataset included domain-specific terms such as `RNN, LSTM, CUDNN`, and `BATCHNORM`, taking the unique characteristics of DL frameworks into perspective.

Table 1 provides an overview of the dataset distribution across TD types, highlighting the prevalence of each category. The dataset contains a total of 38, 881 SATD instances categorised into seven TD types and "WITHOUT CLASSIFICATION" (i.e., the class for the comments that do not belong to any of the seven classes). Among the TD types, AD had a total of 935 across the different DL frameworks. The WITHOUT CLASSIFICATION class had the highest number of instances (23, 156), making the dataset highly imbalanced. However, given the class imbalance, we used several mitigation strategies to avoid bias during training.

Table 2 presents examples of comments (with their associated frameworks) from the dataset with their corresponding TD types. The first column (TD Type) specifies the SATD type, while the second column (Example), presents an example of comment belonging to the SATD Type. The Comments column contains comments indicating SATD from the

[2] https://www.tensorflow.org

[3] https://keras.io

[4] https://deeplearning4j.org

[5] https://caffe.berkeleyvision.org

[6] https://pytorch.org

[7] https://mxnet.apache.org

[8] https://learn.microsoft.com/en-us/cognitive-toolkit/

Table 1 Distribution for each TD type in Liu et al. (2020)'s dataset

TD Type	Total Instances
AD	935
Compatibility	454
Defect	659
Design	10, 891
Documentation	159
Implementation	1, 967
Test	653
WITHOUT CLASSIFICATION	23, 156

Table 2 TD types and example of comments for each TD type from Liu et al. (2020)'s dataset

TD Type	Example
Algorithm	`TODO(jeff,opensource): This should really be a more interesting computation. Maybe turn this into an mnist model instead` — Tensorflow
Compatibility	`This is a workaround for recurrent layer if n is inferred dimension, we can't figure out how to repeat it in cntk now return the same x to take cntk broadcast feature to make the recurrent layer work need to be fixed in GA.` — Keras
Defect	`TODO (Patric): lrn_within_channel will cause core dump in MKLDNN backward. Need to confirm with MKLDNN team and fix later.` — MXNet
Design	`TODO: move to Variable constructor when supported in public release.` — Keras
Documentation	`TODO: add note on how we depend on TracedEval being created in here` — PyTorch
Requirement	`TODO setup for RNN.` — DL4J
Test	`TODO (fchollet): insufficiently tested.` — Tensorflow
WITHOUT CLASSIFICATION	`content=Generating artificial data:We are going to create a movie with square of size one or two by two pixels moving linearly trought time. For convenience we first create a movie with bigger width and height, and at the end we cut it to 40x40` — Keras

source code of seven popular open-source DL frameworks hosted on GitHub, while the second column contains the TD category of the comments. For replication purposes, we have made the dataset publicly available on GitHub.[9]

3.2 Data Cleaning

We followed a standard data cleaning process, drawing suggestions from previous research (Maldonado et al. 2017; Huang et al. 2018; Pinna et al. 2023). This is to help guarantee the quality of data used for consistency with established works. First, we removed stop words, i.e., common words like "the," "is," and "in", which frequently appear in text but typically do not add significant meaning or context to the data (Kaur and Buttar 2018; Feuerriegel et al. 2025). This step is important in NLP tasks with the aim of reducing the dimensionality of the data and also improve training time. We used the `Natural Language Toolkit (nltk)`,[10] a widely-used Python library for NLP tasks, to implement stop word removal. Following the methodology suggested by Huang et al. (2018), we removed punctuation from the comments, with the exception of the exclamation mark (!) and question mark (?). These specific punctuation marks were retained because they can provide context for SATD detection (Pinna et al. 2023).

To help guarantee consistency across all features, the training data was then standardised using the `Standard Scaler`[11] from the `Scikit-learn`. We applied the `Standard Scaler` only after splitting the dataset into training, validation, and test sets. This splitting helped prevent any information from the test or validation sets leaks into the testing process, preserving the integrity of the model evaluation.

3.3 Data Splitting

Following best ML practices, we first separated the dataset into training, validation, and test sets, to help guarantee the integrity of the data and avoid data leakage. We split the dataset into 80% for training and 20% for the final evaluation of the models, ensuring that test data remained unseen during training to prevent data leakage.

For a robust validation and tuning of the ML models, we implemented the k-fold CV (Yadav and Shukla 2016) on the train data. We applied the 5-fold CV, allowing the models to be evaluated across different subsets of data. We selected $k = 5$, as it provides a balance for computational efficiency (James et al. 2013).

3.4 Feature Extraction

Feature extraction is vital towards identifying informative features and reducing the dimensionality to improve model training (Guyon et al. 2008). We employed three different sets of embedding techniques: traditional vectorisation techniques, DL embeddings, and embeddings from LLMs. We chose these techniques to capture both the linguistic nuances and the contextual significance of SATD comments, which are critical for identifying instances

[9] https://github.com/ikoojos/Algorithm-Debt-Research/tree/master/dataset

[10] https://www.nltk.org/

[11] https://scikit-learn.org/dev/modules/generated/sklearn.preprocessing.StandardScaler.html

of AD in DL frameworks. These embeddings and vector representations were used to train ML models. We did not use extracted embeddings for the DL models to avoid redundancy in feature representation, as they incorporate embedding layers (Goodfellow et al. 2016).

We started with the traditional ML techniques: TF-IDF, Count Vectoriser, and Hashing Vectoriser to transform the SATD comments into numerical features. We selected these techniques due to their effectiveness to extract meaningful patterns from textual data in NLP tasks such as classification (Suryaningrum 2023; Arsyah et al. 2024). We then explored more advanced methods, such as embeddings from DL models: RoBERTa,[12] ALBERT[13] and also embeddings from LLMs: INSTRUCTOR[14] and VoyageAI,[15] which offer contextual understanding.

TF-IDF is a feature extraction technique that assess the importance of a word in a document relative to a larger collection of documents by weighting words based on their frequency (Sabbah and Hanani 2023). Intuitively, this technique determines how relevant a given word is in a particular document. We employed TF-IDF to extract important features from SATD comments, as it provides an approach for identifying significant terms in the dataset while minimising the impact of common words such as "the" and "is."

Count vectoriser transforms text data into numerical features by converting words in a document into a matrix of token counts (Danyal et al. 2024). The vectoriser, counts the occurrence of each word and assigns each word in the document a unique ID. For example, words that appear frequently in the dataset like "code," and "TODO" will have higher counts. This representation will then be used as input to the model (Deepa et al. 2019) for training.

Hashing vectorisation converts text into a numeric vector using a hash function (Roshan et al. 2023). This technique processes each word independently, offering speed and low memory usage since no dedicated dictionary is needed (Poczeta et al. 2023). Also, by using this extraction method we can define a vector of any length making it scalable for various text sizes. For instance, SATD comments can be hashed into a fixed-length vectors, enabling efficient processing regardless of the size of the dataset.

DL word embeddings One of the key breakthroughs in DL is the development of word embeddings that capture patterns within data. Learned embeddings from DL models generated by transformer-based architectures like Bidirectional Encoder Representations from Transformers (BERT) (Devlin et al. 2019), can capture the contextual meaning of words, making them useful in representing polysemous words, which can enhance classification performance (Loureiro 2023). To generate the DL embeddings, we used pre-trained DL models from the `transformers` library from `Hugging` Face. Specifically, we used embeddings generated by RoBERTa and ALBERT, to train ML models. RoBERTa and ALBERT were used in previous studies for classifying SATD and non-SATD (Sharma et al. 2022; Rajapaksha et al. 2021), making these models a good choice to explore for

[12] https://huggingface.co/docs/transformers/en/model_doc/roberta

[13] https://huggingface.co/docs/transformers/en/model_doc/albert

[14] https://huggingface.co/hkunlp/instructor-large

[15] https://docs.voyageai.com/docs/embeddings

AD detection in DL frameworks. These models were fine-tuned on our dataset using the `Trainer` class from the `transformers` library. The fine-tuning process involved 30 epochs with early stopping. The embeddings were generated from the last hidden state of the models, specifically from the [CLS] token. The embeddings had a vector size of 768 dimensions, consistent with their base model architectures.

Embeddings from LLMs In addition to the DL embeddings, we explored embeddings generated from LLMs (INSTRUCTOR Su et al. 2023 and VoyageAI) since they were among the models used for text generation on Hugging Face's leaderboard.[16] The INSTRUCTOR (Su et al. 2023) model is an fine-tuned text embedding model that is optimised for a wide range of tasks and achieved state-of-the-art results across 70 different tasks without requiring additional fine-tuning, showing its versatility. This makes the INSTRUCTOR particularly suitable for extracting meaningful insights from text data, which is essential for identifying patterns and relationships in our SATD dataset. VoyageAI (Tang and Yang 2024) is a general-purpose embedding model designed for high retrieval quality, surpassing the performance of OpenAI's V3 Large model.

To generate the embeddings for both the INSTRUCTOR and VoyageAI, we used the pretrained models available on `Hugging Face` directly without additional fine-tuning to evaluate their out-of-the-box performance. Each SATD comment was passed through the respective model to extract the embeddings. The INSTRUCTOR generates embeddings with a dimensionality of 768, making it computationally efficient for downstream ML tasks. With a context length of 3, 200 tokens, it is particularly effective for processing lengthy inputs, such as the detailed discussions present in our dataset. The VoyageAI has an embedding of 1, 024 which allows it to capture and represent patterns and relationships in the data effectively. The extracted embeddings were then used as input features for training a ML model.

3.5 Custom Feature Extraction

To improve the classification of AD, we first sought to identify domain-specific terms frequently occurring in AD-related comments. To achieve this, we grouped comments from the training set by their respective SATD classes and extracted the custom AD features from only the train set to avoid data leakage. This extraction enabled us to identify frequently occurring words associated with each TD class using word counts. Using a Python script, the first author automatically retrieved the top 30 most frequent words for each TD class.

From the top 30 words, we selected five terms (*shape, input, tensor, number, matrix*), as shown in Table 3, to serve as additional features in the dataset. To reduce subjectivity, this selection was guided by a domain expert (not a co-author) with over 10 years of experience in TD research and software engineering. The expert inspected the list that was automatically generated and selected the terms based on explicit criteria: i) high specificity to AD-related comments, ii) relevance to algorithmic and software engineering contexts, and iii) exclusion of general-purpose terms such as *content, todo*. The number of custom AD features was deliberately limited to five to reduce the risk of overfitting to dataset-specific patterns and ensure better generalisability across unseen projects. Additionally, these criteria served to minimise noise and enhance the discriminative power of the features, thereby supporting

[16] https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard

Table 3 Newly engineered features for AD detection and their rationale

Feature	Expected Contribution to Classifier
Shape	Commonly used in AD-related comments to describe mismatches or errors in data structures. Helps the classifier capture AD related to data shape inconsistencies
Input	Frequently appears in comments about improper input handling in algorithmic preprocessing, such as incorrect data formats in ML pipelines or graph algorithms.
Tensor	Strongly associated with ML/DL frameworks where AD occurs in model operations. Enhances recognition of AD connected to tensor manipulation issues
Number	Often used in comments about mis-specified numeric parameters or indexing errors in algorithmic configurations, such as ML hyperparameter tuning or numerical computations
Matrix	Recurrent in AD-related comments about linear algebra operations or dimensional mismatches. Strengthens classifier ability to capture AD related to matrix operations

more accurate detection of AD. To further evaluate this selection and mitigate potential bias, we constructed feature sets by combining subsets of up to five terms from the remaining candidate words that met the above criteria. These subsets were evaluated using the same training and evaluation pipeline, and none yielded improved model performance. The final five terms were confirmed by the co-authors.

Training with custom AD features To integrate the words that we extracted as features into our data, we developed a custom ML pipeline that integrates these domain-specific features with an LR model. We only experimented with the Custom AD Features with LR since the LR had the highest F1-score. The overall process of how we implemented the pipeline is illustrated in Fig. 2.

The pipeline included a `CustomFeatureExtractor`, that was implemented as a scikit-learn-compatible transformer. This extractor was initialised with a predefined set of keywords and, during transformation, it computed binary features indicating whether each keyword was present in a given SATD. For each SATD input, the extractor created a dictionary of binary features, where each feature corresponded to the presence (1) or absence (0) of a keyword in the comment. These features were then converted into a sparse matrix format for computational efficiency. To combine these Custom AD Features with standard

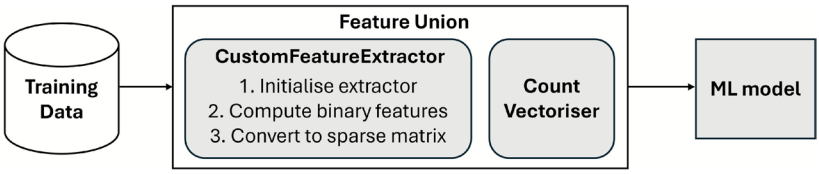

Fig. 2 Custom pipeline for extracting features, showing the steps we followed in executing the CustomFeatureExtractor

vectorisation, we used the `FeatureUnion` method. This method integrated the output of the `CustomFeatureExtractor` with the Count Vectorser, which converted the raw text data into a bag-of-words representation.

We incorporated these features into a classification pipeline, where we trained the LR model. To help guarantee a balanced performance across all classes (considering the class imbalance), we set the model's class weight parameter to "balanced" and defined a Grid Search Cross-Validation (Grid Search CV) to optimise hyperparameters. The Grid Search CV was used to evaluate different combinations of the regularisation (C), penalty (l2), and maximum iterations (max iter) using a 5-fold CV strategy. We set the scoring metric as the macro-averaged F1-score to provide a fair comparison because of the class imbalance in the dataset.

One potential limitation of the custom feature extraction approach was in its reliance on specific keywords from the dataset. By focusing on frequently occurring terms, there was a risk of overfitting to the specific dataset we used. To mitigate this, the selected keywords were carefully curated to help guarantee their relevance and applicability to AD, as discussed further in Sect. 7.

3.6 Baseline and Benchmarks

To mitigate the risk of bias and subjectivity in establishing a baseline, the results of the trained models were compared against two types of baselines: (i) reference baseline, which established minimum performance thresholds, and (ii) benchmarks from prior studies on SATD identification. These baselines helped to guarantee that the performance of the models extended meaningfully beyond random or simplistic predictions. The benchmarks were used to compare the models against established methods in SATD research.

Reference baseline We selected the Random Classifier (Ameisen 2021) as our reference baseline. A Random Classifier assigned labels to SATD instances from the dataset randomly, without considering any underlying features or patterns in the data. During the training process, it did not learn from the data; instead, it simply assigned labels to the distribution of classes present in the dataset. When making predictions, it randomly selected a class label for each instance, independent of any input features. This approach helped establish a lower bound of performance, serving as a baseline to measure the effectiveness of more sophisticated models. The Random Classifier achieved an F1-score of 0.04% on the dataset of Liu et al. (2020), which serves as the lower bound for performance evaluation.

Benchmarks based on previous works The approach by Sharma et al. (2022) was considered a benchmark noting that it is the only existing work that performed SATD classification that includes AD. RoBERTa, the best in their study, had an F1-score of 31% for AD and macro-averaged F1-score of 56%. Using this as a benchmark enabled the direct comparison of the results from our models. The approach by Li et al. (2023a), known as MT-Text-CNN, could be considered here for comparative purposes as another recent work in SATD detection; this method leverages multitask learning and CNN, to identify multiple SATD types, achieving an F1-score of 61%. For these benchmarks, the experiments were conducted on publicly available datasets.

3.7 Classification Models

To investigate the automated detection of AD in DL frameworks, we explored different ML/DLmodels, drawing inspiration from previous research on traditional software systems (Ren et al. 2019; Sharma et al. 2022; Li et al. 2023a). The ML models that we explored were: SVM (Joachims 1998), LR (Hilbe 2009), and RF (Breiman 2001). We used these models for because: i) they have been used for previous SATD classification tasks (Ren et al. 2019; Sharma et al. 2022), and ii) they performed well in text classification related tasks (Wang et al. 2019).

SVM is a supervised ML technique, that is based on statistical learning theory (Yang et al. 2015) and can be used for classification tasks. They are effective in tasks where the objective is to classify data into distinct classes (Cortes and Vapnik 1995). SVMs classify data by identifying a hyperplane that separates different classes (Rejani and Selvi 2009). The advantages of SVM are that they are particularly effective when dealing with high-dimensional data, making it suitable for applications like text classification from the SATD dataset. For instance, it can extract patterns from complex and high-dimensional representations of text, making it suited for our study on AD detection.

LR is a supervised learning algorithm (Musa 2013) that has previously demonstrated strong performance for data with a large number of features, making it suitable for experimenting AD detection (Hilbe 2009). LR is based on the logistic function and evaluates the statistical significance of independent variables in relation to probabilities (Shah et al. 2020). In text classification, the LR model recognises a vector containing variables after which it evaluates the coefficients for each input variable to predict the class of text in the form of a word vector. The LR model can excel in scenarios with a large number of features due to its simplicity and efficiency, particularly when processing sparse textual data (Bertsimas and King 2017). However, its assumption of linear relationships between features may limit its ability to capture non-linear patterns inherent in AD-related data (Sarker 2021).

RF is an ensemble ML technique (Breiman 2001; Biau and Scornet 2016) that creates a collection of decision trees during training. These trees operate on the divide-and-conquer principle, where each tree contributes to the final prediction. During classification, each tree in the RF votes for the most likely class, and the final prediction is determined by aggregating these votes. RF's ability to aggregate decisions from multiple trees ensures its robustness in classification tasks. Other advantages of the RF are its relatively few parameters to tune, along with its capability to handle small sample sizes and high-dimensional feature spaces (Khoshgoftaar et al. 2007; Breiman 2001).

RoBERTa stands for Robustly optimised BERT pre-training approach (Liu et al. 2021b), and builds on the foundational BERT architecture (Devlin et al. 2019) with different enhancements aimed at improving its performance on NLP tasks. These include training on larger datasets, removing the next sentence prediction objective, using longer text sequences, and employing dynamic masking during pre-training. These advancements have made RoBERTa particularly effective for classification tasks such as SATD detection

(Nunes et al. 2024). The RoBERTa model employs a transformer architecture with self-attention mechanisms, which enable it to capture complex dependencies within textual data such as SATD (Vaswani et al. 2017). Additionally, RoBERTa was used in a previous study on AD (Sharma et al. 2022).

ALBERT is a variant of BERT designed to reduce computational overhead and memory requirements while maintaining high performance (Lan et al. 2019). It achieves this through parameter-sharing across layers and factorised embedding parameterisation, which reduces the number of parameters compared to traditional BERT models. These optimisations make ALBERT a lightweight and suited where computational resources are constrained. ALBERT's efficiency allows for rapid processing of large datasets and its streamlined architecture is particularly useful in our study, where computational resources are needed. This combination of efficiency and performance positions ALBERT as a good option to explore for detecting AD. Additionally, ALBERT was used in a previous study on AD (Sharma et al. 2022), making it an ideal choice for us to explore.

3.8 Experimental Setup

Hyperparameter tuning Hyperparameter tuning plays an important role in optimising the performance of ML/DL models. To systematically refine hyperparameters and optimise the classification outcomes, we used the Grid Search CV (Bergstra et al. 2011). This approach allowed for the selection of optimal hyperparameters by evaluating different combinations across multiple iterations, ensuring optimal performance of our models by selecting the best combinations.

We performed hyperparameter tuning for the ML models (i.e., SVM, LR, and RF). We conducted experiments to explore different kernel functions for SVM, regularisation techniques for LR, and tree-based hyperparameters for the RF. The optimal configurations were then automatically selected based on F1-score. We have provided the details on the hyperparameters for each ML model in Appendix A. We did not perform multiple experiments for the DL models to exhaustively search for hyperparameters. This is because DL models have a vast number of hyperparameters, and exhaustive hyperparameter tuning can become computationally expensive (Joulin et al. 2017). Instead, we used standard values based on prior research, ensuring a reasonable balance between model performance and computational efficiency.

Handling class imbalance Training ML models from imbalanced datasets can be difficult and could bias the majority class, leading to poor performance on the minority classes (Batista et al. 2004). The dataset we described in Sect. 3.1 exhibited class imbalance, particularly with the WITHOUT CLASSIFICATION category having a much higher number of occurrences than the other classes. To mitigate the issue of class imbalance in our study, we set the class weight parameter to "balanced" (Li and Mani 2021). This value assigns weights inversely proportional to the frequency of each class in our training data and gives higher weights to minority classes such as AD and lower weights to the majority class, encouraging the model to treat all classes more equally during training.

3.9 Model Evaluation

Accuracy is appropriate when the classes are balanced (i.e., if each class is represented by a similar number of samples within the dataset). However, the dataset we used is imbalanced, indicating that evaluating our models based on accuracy alone would lead to biased results favoring the majority class (Batista et al. 2004). We evaluated our ML/DL models using metrics such as precision, recall, F1-score, and macro-averaged F1-scores, consistent with prior studies on SATD detection (Ren et al. 2019; Santos et al. 2020; Sharma et al. 2022; Li et al. 2023a).

Precision is a metric used to evaluate the performance of classification models by measuring the accuracy of the positive predictions (Huang et al. 2018). A high precision indicates that most of the AD instances it identified as positive are indeed correct. A focus on precision alone may overlook false negatives, which is why it is often analysed in conjunction with recall to provide a more comprehensive evaluation of a model's performance. In the context of AD detection, a true positive is an instance where the model correctly identifies a comment as AD, while a false positive is an instance where the model incorrectly identifies an SATD that is not AD as AD. Mathematically precision is computed as follows:

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives}.$$

Recall measures the proportion of true positive instances identified by a model relative to the actual total number of positive instances present (Foody 2023). A high recall for AD suggests that the model is efficient at identifying the majority of AD cases, thereby minimising false negatives. A false negative is an instance where a model identifies a comment as a TD type other than AD when it is actually AD. Mathematically recall is computed as follows:

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives}.$$

F1-score Given the trade-off in using precision or recall alone for evaluating the performance of a model (Mining 2006), we also compared the performance of our ML/DL models using the F1-score following the metrics used in software engineering papers (Arisholm et al. 2007; Sharma et al. 2022; Li et al. 2023a). The F1-score combines precision and recall to provide a balanced measure of a model's performance, particularly when dealing with unbalanced class distribution. The F1-score is computed using the harmonic mean of precision and recall. This means that a low score in either will significantly lower the overall F1-score. Mathematically F1-score is computed as follows:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}.$$

Macro-averaged F1-score is the F1-score for each class calculated independently and then averaged (Suominen et al. 2008). This metric treats all classes equally, regardless of their frequency in the dataset. It is important in our study for addressing class imbalance, as

it ensures that the minority classes (e.g., AD) are given equal weight during evaluation. However, a major limitation of the macro-averaged F1-score is that it may not provide a true reflection of the overall accuracy or a model's performance on the majority classes. Mathematically the macro-averaged F1-Score is computed as follows:

$$\text{Macro-averaged F1} = \frac{1}{N}\sum_{k=1}^{N} \text{F1}_k$$

where N is the total number of classes and k represents the index of each individual class. We reported these evaluation metrics as percentages for clarity and consistency in Sect. 4.

3.10 Statistic Test

To verify whether the improvements in the F1-score for AD using the LR with Custom AD Features were statistically significant compared to the results of the other techniques, we conducted a Wilcoxon signed-rank test (Wilcoxon et al. 1970) at a significance level (p-value) of 0.05, following the approach used in previous studies (Li et al. 2023a; Huang et al. 2018). The Wilcoxon signed-rank test is a non-parametric test designed for paired comparisons, making it suitable for comparing the performance of our models on the same dataset. Hence, we made a pairwise comparison between the F1-score of LR with Custom AD Features and all the other methods, where we obtained a p-value for each comparison.

We also calculated the Cliff's Delta $|\delta|$ to quantify the magnitude of the differences in the performance of the models (Vargha and Delaney 2000). The Cliff's Delta measures the effect size, complementing the p-values (Macbeth et al. 2011; Meissel and Yao 2024). To calculate the Cliff's Delta, we compared the F1-score of LR with Custom AD Features and all the other methods pairwise. The amount of difference between model performances is considered negligible $|\delta| < 0.11$, small ($0.11 \leq |\delta| < 0.28$), medium ($0.28 \leq |\delta| < 0.43$), or large ($|\delta| \geq 0.43$) (Vargha and Delaney 2000).

4 Results

Among the ML techniques, LR with the selected five Custom AD Features achieved the highest F1-score of 54% for AD, demonstrating the effectiveness of domain-specific feature engineering in enhancing AD detection in Table 4. To assess potential bias from the selection of these terms, we also evaluated subsets of the remaining top-30 words. These subsets achieved F1-scores ranging from 50% – 53% for AD, slightly lower than the chosen five terms. These results indicate that the selected five terms were not only more effective but also essential in reducing noise introduced by less informative terms.

This result was closely followed by LR with Count Vectoriser (52%) and RF with Count Vectoriser (51%), indicating that Count Vectoriser-based features were generally effective for AD classification. We observed that in terms of F1-score, the Count Vectoriser techniques outperformed TF-IDF and Hashing Vectorisers. For example, the Count Vectoriser outperformed LR with TF-IDF (48%) and LR with Hash Vectoriser (48%), suggesting that simple frequency-based representations are more effective than weighted term frequency

Table 4 Comparison of ML/DL model performance (in percentages) across different TD types. The table presents F1-scores for AD, COM – Compatibility, DEF – Defect, DES – Design, DOC – Documentation, IMP – Implementation, TES – Test, and W/C - WITHOUT CLASSIFICATION. The table also presents the overall macro-averaged F1-scores – M/F1 and macro-averaged accuracy – M/AC. The best results are highlighted in **bold**. Abbreviations: CV – Count Vectoriser

Technique	AD-F1	COM-F1	DEF-F1	DES-F1	DOC-F1	IMP-F1	TES-F1	W/C-F1	M/F1	M/AC
SVM + TF-IDF	44	47	40	78	51	57	49	93	57	84
SVM + CV	41	35	35	79	32	60	68	95	56	85
SVM + Hash Vectoriser	43	45	37	79	38	58	51	93	55	84
LR + TF-IDF	48	53	45	80	45	59	53	94	60	85
LR + CV	52	49	49	82	39	63	64	96	62	87
LR + Hash Vectoriser	48	51	43	81	47	60	55	94	60	85
RF + TF-IDF	42	51	41	84	40	65	63	96	60	**89**
RF + CV	51	**57**	**54**	**85**	59	69	70	96	68	**89**
RF + Hash Vectoriser	46	52	43	84	45	65	58	95	61	88
RoBERTa Embeddings	44	50	43	84	16	69	76	96	60	88
ALBERT Embeddings	25	25	25	80	05	61	67	96	48	83
VoyageAI Embeddings	30	19	21	76	34	20	21	92	53	53
INSTRUCTOR Embeddings	29	31	31	69	35	51	57	93	51	77
RoBERTa	43	50	52	85	54	68	**80**	97	66	**89**
ALBERT	21	04	26	83	00	64	67	97	45	87
Random Classifier	04	01	03	17	00	06	02	20	07	12
LR + Custom AD Features	**54**	54	53	84	**59**	**69**	73	**96**	68	88
RoBERTa + Custom AD Features	49	56	57	85	53	69	78	97	68	89

(TF-IDF) and hashed features for AD detection in DL frameworks. Similarly, the SVM models performed the least, with F1-scores ranging from 41% – 43%. The low F1-score of the SVM models could be likely due to data sparsity as a result of the feature extraction techniques resulting in high dimensional sparse vectors where many values are zero.

Although the F1-scores for the models were generally low for AD, the models achieved higher scores in classifying SATD in classes such as Design (ranging between 78% – 85%) and WITHOUT CLASSIFICATION (ranging between 93% – 96%). Specifically, RF with Count Vectoriser, achieved the highest F1-score (85%) for Design Debt, while both RF with TF-IDF and RF with Hash Vectoriser achieved an F1-score of 84%. The LR with Cus-

tom AD Features also had high scores in TD classes for Implementation and WITHOUT CLASSIFICATION with an F1-score of 69% and 96% respectively. The success in the ML models detecting Design Debt and the WITHOUT CLASSIFICATION classes could likely be attributed to the structured nature of design-related issues, which makes it easier for the models to identify patterns in the data. Another reason could be the large number of their instances within the dataset.

In terms of macro-averaged F1-score, the combination of LR with Custom AD Features and RF with Count Vectoriser achieved the highest score of 68%, indicating that these models provided a more balanced performance across different TD types. This score is followed by LR with Hash Vectoriser and RF with TF-IDF with a score of 60% both. The SVM with Hash Vectoriser had the lowest macro-averaged F1-score, highlighting its difficulty in maintaining a strong balance between precision and recall across the TD types. The performance of LR with Custom AD Features suggests that domain-specific feature engineering improves model effectiveness by enhancing its ability to capture AD-related patterns across different contexts. Similarly, RF with Count Vectoriser's high score can be attributed to the ensemble learning of RF, which benefits from multiple decision trees voting on predictions, improving generalisation. However, the lower performance of SVM with Hash Vectoriser suggests that SVM struggles to generalise well across multiple TD types, likely due to its sensitivity to sparse feature representations and difficulty in handling class imbalances.

The accuracy results indicated that RF with TF-IDF and Count Vectoriser achieved the highest accuracy of 89%, suggesting their effectiveness in the overall classification of the different TD types. LR with Custom AD features achieved an accuracy of 88%, while LR with Count Vectoriser followed closely with 87% accuracy. The SVM models (TF-IDF, Count Vectoriser, and Hash Vectoriser) had the lowest accuracy ranging between 84% and 85%. Although the F1-scores for the models were lower for AD and TD types such as Defect, and Documentation, their overall performance demonstrates their potential in detecting different TD types in DL frameworks. However, given that accuracy can be misleading in imbalanced datasets, relying solely on this metric may overestimate the effectiveness of the models, especially for underrepresented TD types like AD.

Among the ML models, the RF achieved the highest precision in Table 5. Specifically, the combination of RF and TF-IDF achieved, a precision of 96%, Count Vectoriser 92% and Hash Vectoriser 91%. This high precision could likely be due to the ensemble nature of the RF, where the trees vote before providing the final output. This is followed by the LR models having a precision between 53% and 54%. SVM with the Count Vectoriser had the lowest performance of 37% in terms of precision. The high precision of the RF model indicates its strong ability to correctly identify AD instances with minimal false positives. This makes RF particularly suitable for scenarios where avoiding false positives is critical, such as in systems where false alarms can lead to unnecessary resource allocation.

For recall, LR with Custom AD Features achieved the highest recall of 57%. This suggested that leveraging domain-specific knowledge can enhance AD detection by capturing subtle patterns missed by generic feature extraction methods. The higher recall indicated that LR with Custom AD Features could capture a broader range of AD instances, making it particularly useful in scenarios where identifying a high number of AD instances is critical, such as during early-stage debugging. This was followed by the LR with Count Vectoriser (46%) and SVM with Count Vectoriser. This further highlighted the effectiveness

Table 5 Performance Metrics (in percentages) across different experiments showing the precision and recall for AD, with the best results highlighted in **bold**

Model and Feature Extraction	Precision	Recall
SVM + TF-IDF	54	38
SVM + CV	37	46
SVM + Hash Vectoriser	49	38
LR + TF-IDF	54	44
LR + CV	53	50
LR + Hash Vectoriser	53	44
RF + TF-IDF	**96**	27
RF + CV	92	35
RF + Hash Vectoriser	91	30
RoBERTa Embeddings	49	39
ALBERT Embeddings	23	28
VoyageAI Embeddings	67	32
INSTRUCTOR Embeddings	20	52
RoBERTa	55	36
ALBERT	29	16
LR + Custom AD Features	52	**57**
RoBERTa + Custom AD Features	61	41

of the Count Vectoriser in identifying AD instances. On the other hand, the RF models achieved low recall ranging from 27% for the TF-IDF to 35% for Count Vectoriser, likely due to their prioritisation of precision (i.e., prioritising correctly identified cases).

> **RQ1 Summary** For the ML models, LR with Custom AD Features achieved the highest F1-score (54%) for AD detection, outperforming other methods. RF models, while achieving the highest precision (96%), struggled with recall (as low as 27%), limiting their overall effectiveness for AD detection. For recall, the LR with Custom AD features had the highest score of 57% for AD. In terms of macro-averaged F1-score, LR with Custom AD Features and RF with Count Vectoriser achieved the highest score of 68%. For accuracy, the RF with Count Vectoriser and TF-IDF achieved the highest score of 89%.

Among the DL embeddings, RoBERTa achieved the highest F1-score for AD, scoring 44% in Table 4. This was followed by embeddings from VoyageAI and INSTRUCTOR, which achieved 30% and 29% respectively. ALBERT embeddings achieved the lowest F1-score of 25%. When considering recall, the INSTRUCTOR achieved the highest score of 52% for AD. This was followed by RoBERTa and VoyageAI embeddings with scores of 39% and 32%. The ALBERT embeddings had the lowest recall of 28%.

For the macro-averaged F1-score, the RoBERTa embeddings achieved the highest score of 60%, which was followed by the VoyageAI and INSTRUCTOR embeddings, which achieved 53% and 51% respectively. ALBERT embeddings achieved the lowest macro-averaged F1-score of 45%. For precision, the VoyageAI embeddings achieved the highest precision of 67% for AD. This was followed by RoBERTa embeddings with a score of 49%. ALBERT and INSTRUCTOR embeddings achieved a precision of 23% and 20% respectively. These low performance scores might be due to the limitations of general-purpose embeddings in effectively representing AD characteristics, potentially resulting from insufficient domain-specific training or the broad, generalised nature of these embeddings.

Beyond DL embeddings, we evaluated stand-alone DL models, and RoBERTa achieved the highest F1-score for AD detection at 43%. In contrast, the ALBERT model achieved an F1-score of 21%. To investigate whether domain-specific features could further enhance the performance of DL models, we concatenated a binary Custom AD Features vector to the RoBERTa [CLS] embedding prior to the final classification layer. This model achieved an F1-score of 49% for AD, a 14% improvement over the RoBERTa. For the precision of AD, RoBERTa achieved a score of 55%, while ALBERT scored 29%. Also, the recall score for ALBERT remained at 16% for AD, while RoBERTa achieved a higher score of 36%, which is still low (less than a random guess). The macro-averaged F1-score followed the same trend, where RoBERTa achieved a score of 66%, while ALBERT scored 45%. The overall consistent low performance of ALBERT could be likely due to its lightweight architecture, which may prioritise efficiency at the expense of detailed feature representation.

Beyond AD, the performance of the DL models was low for Documentation and Defect Debt, as indicated by the low F1-scores. Notably, ALBERT model performed poorly and was unable to detect Documentation Debt. Embedding techniques for ALBERT and RoBERTa had F1-scores of 5% and 16%, respectively for Documentation Debt. For Defect Debt, the F1-scores ranged between 21% (VoyageAI embeddings) to 52% (RoBERTa). Despite the low F1-scores achieved by our DL models, they performed well for other TD types. For instance, RoBERTa excelled in detecting WITHOUT CLASSIFICATION, Design, and Test Debt, achieving F1-scores of 97%, 85%, and 80% respectively. Likewise, ALBERT had a high F1-score of 97% for WITHOUT CLASSIFICATION, 83% for Design, and 67% for Test Debt. The high performance of the DL models for these specific TD types could be due to the high number of samples in these classes, providing the models with sufficient training data to learn their characteristics effectively.

For the accuracy metric, RoBERTa achieved the highest accuracy of 89%, while ALBERT had an accuracy of 87%. The RoBERTa embeddings achieved an accuracy of 83%, while INSTRUCTOR achieved a score of 77%. The VoyageAI embeddings had the least accuracy of 53%. The variation in accuracy across models and embeddings highlights their differing effectiveness in TD classification.

Given that DL models require a large amount of training data, the limited number of samples for TD types like AD, Compatibility, Defect, and Documentation Debt could have contributed to the low F1-scores for these TD types, as the available data might not have been sufficient for effective training. Hence, the overall performance for the DL models was lower than for the ML models.

> **RQ2 Summary 2:** RoBERTa outperformed the other DL models for AD detection in terms of F1-score, achieving score of 43%. The Instructor embeddings achieved the highest recall of 52%, while VoyageAI achieved the highest precision of 67%. ALBERT had the lowest scores for precision, recall, F1-scores, and macro-averaged F1-scores, with values of 29%, 16%, 21%, and 45%, respectively, limiting their overall effectiveness for AD detection in DL frameworks. Despite their low performance for AD detection, in terms of accuracy, Deep Leaning Models achieved a higher score. RoBERTa and its embeddings achieved the highest accuracy of 89% and 88%, respectively, followed by ALBERT with an accuracy of 87%, suggesting its effectiveness in broader TD classification tasks.

Table 6 Pairwise comparison of statistical significance (p-value) and effect sizes (Cliff's Delta) for F1-scores in AD detection, comparing LR with Custom AD features against other ML/DL models. Each row represents a comparison between LR with custom AD Features and the model in the model column. A p-value < 0.05 indicates a statistically significant difference, while a higher Cliff's delta represents a larger effect size

Model	p-value	Cliff's Delta
SVM + TF-IDF	0.0158	0.5000
SVM + CV	0.0058	0.3473
SVM + Hash Vectoriser	0.0058	0.3430
LR + CV	0.0309	0.3281
LR + Hash Vectoriser	0.0482	0.3750
RF + TF-IDF	0.0168	0.3125
RF + CV	≥ 0.05	−0.0156
RF + Hash Vectoriser	0.0140	0.3593
RoBERTa Embeddings	≥ 0.05	0.2345
ALBERT Embeddings	0.0190	0.3281
VoyageAI Embeddings	0.0022	0.5937
INSTRUCTOR Embeddings	0.0002	0.6406
RoBERTa	≥ 0.05	0.1562
ALBERT	0.0294	0.3125
Random Classifier	0.0000	1.0000

Overall, out of the models we evaluated, the LR with Custom AD Features achieved statistically significant improvements ($p < 0.05$) over other models (Table 6). Specifically, the models SVM + TF-IDF, SVM + Count Vectoriser, SVM + Hash Vectoriser, LR + Count Vectoriser, LR + Hash Vectoriser, RF + TF-IDF, RF + Hash Vectoriser, ALBERT Embeddings, VoyageAI Embeddings, INSTRUCTOR Embeddings, and the Random Classifier demonstrated statistically significant differences in performance. Conversely, RF + Count Vectoriser, RoBERTa, and RoBERTa Embeddings showed no significant difference in performance.

These statistically significant differences varied in magnitude. Among the ML/DL models that we compared with the LR with Custom AD Features, the SVM + TF-IDF, Random Classifier, INSTRUCTOR Embeddings, and VoyageAI Embeddings exhibited the large effect sizes with Cliff's delta values greater than 0.43. The effect sizes between LR with Custom AD Features and SVM + CV, SVM + Hash Vectoriser, LR + CV, LR + Hash Vectoriser, RF + TF-IDF, RF + Hash Vectoriser, ALBERT Embeddings, and ALBERT, were considered medium effect sizes (Cliff's Delta between 0.280 and 0.43). We observed small effect sizes (Cliff's Delta between 0.11 and 0.28) with RoBERTa Embeddings and RoBERTa. The effect size for RF + CV was negligible with a negative Cliff's delta of −0.0156.

These test underscored the value of custom AD features in enhancing the performance of ML/DL models. The statistically significant improvement, particularly in those models with medium to large effect sizes, suggested that incorporating AD features can lead to more effective detection of AD in DL frameworks. The small effect sizes suggested that while statistically significant, the practical impact of the LR model's improvements over these specific models may be less pronounced. The negligible effect size for RF + CV reinforces the lack of practical difference between this model and the LR model. We have provided a more detailed classification report for all the experiments in Appendix B.

5 Discussion

In this study, we investigated the detection of AD in DL frameworks, by evaluating the strengths and limitations of ML/DL models. Our findings reveal that while these models demonstrate potential for AD detection, their performance varies, suggesting differing abilities to handle the inherent complexities of AD.

To validate our findings, we conducted a statistical significance test, confirming that the performance differences we observed between LR and Custom AD Features and the other ML/DL models were statistically significant. Additionally, we compared the models against a Random Classifier baseline to ensure that their predictions exceeded chance levels. As expected, the Random Classifier showed low performance across all metrics, reinforcing the effectiveness of our ML/DL models in detecting AD (Table 4). We now examine the specific strengths and weaknesses of these ML/DL models in AD detection.

Performance of ML/DL models Among the ML/DL models, our proposed approach of using Custom AD Features with LR achieved the highest F1-score for AD (54%). While we did not test all 30 words that were extracted for the AD class, evaluating a subset of the remaining top-30 terms yielded slightly lower F1-scores (50–53%), suggesting that the manually guided selection captures the most discriminative domain-specific terms. By involving an independent domain expert in the selection process, we mitigated potential bias and improved the relevance of the features. These findings underscore the value of combining automatic frequency-based extraction with expert validation to enhance performance in AD detection. This also suggests that further investigation of custom feature engineering could lead to even greater improvements.

Among the DL models, RoBERTa demonstrated superior performance, achieving an F1-score of 43% for AD and the highest macro-averaged F1-score (66%) across all experiments, which can be attributed to its attention mechanism, which captures contextual nuances (Vaswani et al. 2017). While the 43% F1-score for AD may seem modest, it reflects the inherent complexity of AD within DL frameworks and the challenges posed by limited labelled data. The relatively low F1-scores of DL models for AD and TD types like Compatibility, Defect, and Documentation Debt further underscores the impact of data scarcity, suggesting that specialised training approaches are needed for AD these TD types.

While the F1-scores and macro-averaged F1-scores provided valuable insights into model performance, we acknowledge the impact of class imbalance in our dataset. The WITHOUT CLASSIFICATION class constituted a significantly larger proportion of the data, which could potentially bias the models. Future work could explore techniques like oversampling, undersampling, to mitigate the effects of class imbalance and potentially improve performance, particularly for the under-represented classes like AD, Compatibility Debt, Defect, and Documentation Debt.

A key question arising from our initial experiments was whether our custom AD features could enhance large-scale transformer models for AD detection. To investigate this, we conducted an additional experiment, integrating our domain-specific features into the RoBERTa

model. Specifically, we integrated our Custom AD Features into the RoBERTa architecture by concatenating the binary feature vector with the RoBERTa [CLS] token embedding before the final classification layer. This hybrid model achieved an F1-score of 49% for AD, a 14% improvement over the RoBERTa. These findings demonstrate that domain-specific features provide complementary domain-specific information that strengthens RoBERTa's predictive capability. However, the current feature set alone does not fully close the gap to effective AD detection, suggesting that additional strategies will be needed to fully leverage these custom features.

Challenges with LLM embeddings Standalone LLM embeddings such as those from VoyageAI and INSTRUCTOR appeared to be less effective for AD detection, achieving notably lower F1-scores compared to other ML/DL models. This limitation may be due to lack of task-specific pre-training or the inherent complexity of AD, which requires contextual understanding beyond generic embeddings. Despite their successes in other NLP tasks (e.g., text classification Su et al. 2023), the embeddings from these LLMs underperformed for AD detection, highlighting important directions for future research. Future work could address this limitation by exploring richer LLM-based feature representations. For example, fine-tuning embeddings on AD-specific corpora or leveraging prompt-engineered contextual embeddings to better capture the linguistic cues relevant to the detection of AD.

Effectiveness of count vectoriser For the vectorisation techniques, the Count Vectoriser achieved the highest F1-scores across all the ML models, indicating its effectiveness in capturing the textual features relevant to AD detection. This performance might be attributed to the nature of AD-related terminology, which often features task-specific keywords. Unlike TF-IDF, which penalises frequently occurring terms, the Count Vectoriser emphasises these repetitions, enhancing its ability to identify AD patterns. Future work could investigate this hypothesis by comparing the term importance assigned by Count Vectoriser, TF-IDF, and Hash Vectorisers, and evaluate their effectiveness in the detection of AD.

Variability in AD comments The generally low F1-scores for AD that was observed across the different experiments could be likely due to the variability in the length of AD comments. These comments ranged from concise statements to lengthy, detailed descriptions. For example short comments like `"content=TODO: add an epsilon [CR comment by Nikos]"` lacks sufficient keywords, making detection harder for the ML/DL models. While traditional feature extraction methods like the Count Vectoriser are effective for term frequency analysis, they struggled to capture the semantic nuances of AD, especially in shorter comments with limited keywords, due to issues with synonyms and polysemy (Joachims 1998).

Misclassifaction of AD To gain further insights into the performance of the models, we analysed the confusion matrix (Fig. 3) of our proposed technique. This analysis revealed frequent misclassifications of AD into Design Debt or Implementation Debt, suggesting conceptual similarities between AD and these TD types. The overlap with Design Debt may arise from the shared focus on architectural and structural concerns. While AD often involves algorithmic inefficiencies or limitations, such issues may be rooted in or result

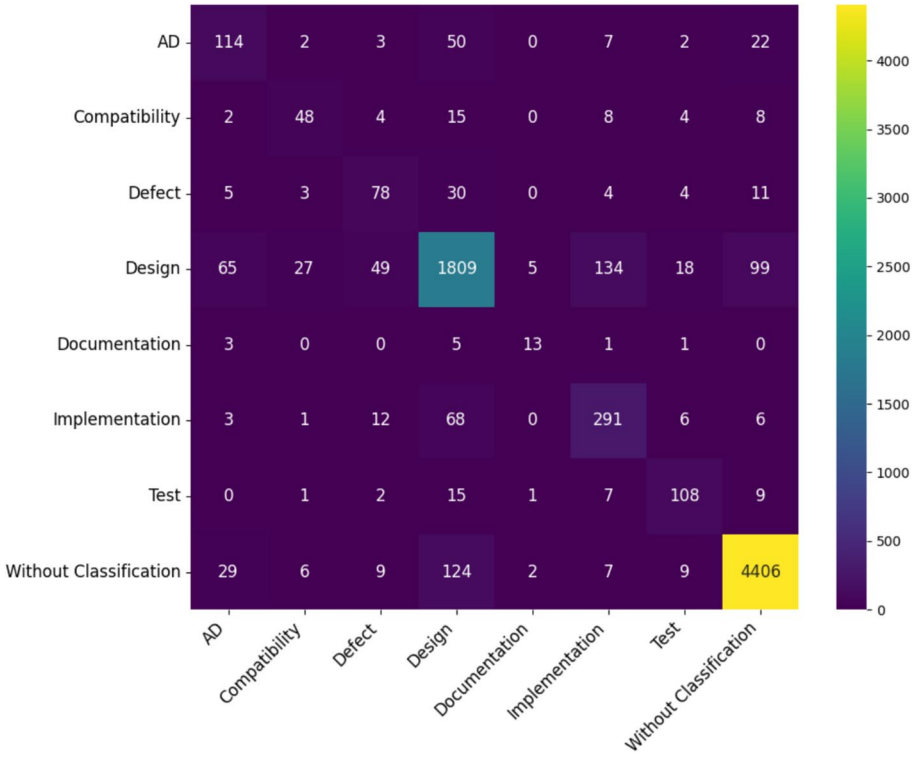

Fig. 3 Confusion Matrix of the LR with Custom AD Features

from poor design decisions. For example, tasks like improving algorithm performance may implicitly touch on design aspects, leading to ambiguity. For example,

- `todo: this seems to be very slow`
- `todo(misard) if we see graphs built with a different structure, relax this constraint. leaving it here for now to avoid writing unnecessary complex code since we believe graphs generated by front ends all have the back edge as the second input to the merge node.`

The first comment reflects a performance concern which is characteristic of AD. However, the lack of specificity and context in the comment may cause the model to interpret the comment as Design Debt, where the algorithm's slowness could stem from poor architectural choices rather than the algorithm itself. The second comment represents AD because it highlights algorithmic limitations based on specific assumptions about graph structures (e.g., the back edge as the second input to the merge node). The decision to avoid "unnecessary complex code" reflects a trade-off prioritising simplicity over robustness, deferring potential issues to the future. The comment was misclassified as Design Debt possibly due to the language in the comment, such as references to "graphs built with a different structure" and "merge nodes", which suggest architectural or structural concerns.

The overlap with Implementation Debt may be due to their shared emphasis on practical execution and optimisation challenges. Implementation Debt typically refers to issues in the code-level execution of a task, which may be closely aligned with the implementation of an algorithm. For example, these two comments were classified as Implementation Debt although the ground truth was AD.

- `todo: update w.r.t. time offset`
- `for each device, add the tensors on that device first, then gather the partial sums from multiple devices.`

The first comment suggested modifying an algorithm to account for time offsets, which is characteristic of AD. The reference to implementation-related specifics such as "update" might have caused the misclassification as Implementation Debt. The second comment pertained to the implementation of a computationally distributed algorithm. While the underlying issue relates to the algorithm's functionality, the focus on practical execution ("gather the partial sums") could have made the model classify it as Implementation Debt.

These findings highlight that AD is not an isolated phenomenon but is interconnected with broader software engineering practices, intersecting with other TD types such as Design and Implementation Debt. This calls for the need for developers to provide clear and detailed comments, enabling the distinction of algorithmic issues from design and implementation concerns in SATD documentation. Future efforts could focus on creating more granular definitions of these TD categories and designing domain-specific features to capture these nuanced differences. For example, distinguishing AD from Design Debt could use semantic analysis to identify context-specific usage of terms like 'algorithm' or 'structure'.

Additionally, a multi-step model could be created to help resolve these misclassifications. For instance, similar to the work by Venkatkrishna et al. (2024), a first model could classify AD and another general model could be used to detect other TD types.

Comparison with previous works Our findings regarding the challenges of AD detection aligned with previous research by Sharma et al. (2022). They classified 11 TD types in R programming, and reported low F1-scores for AD, consistent with our observation that AD achieved some of the lowest F1-scores (ranging from 30% to 54% across four of our 16 experiments). However, our proposed approach using LR with custom AD features achieved a 54% F1-score, a 74% improvement over their reported 31%, highlighting the importance of domain-specific features for AD detection in DL frameworks. This suggests that tailored strategies and custom feature extraction could help mitigate the unique challenges of AD detection.

Our study also aligned with Li et al. (2023a), who noted challenges in classifying certain TD types, including Documentation Debt. Their study reported F1-scores of 62% for Documentation Debt and 44% for Test Debt, though our best-performing model surpassed this score for Test Debt. This reinforced the persistent difficulties in SATD classification, particularly for "hard to find" categories like Documentation Debt (Yu et al. 2020). These challenges might be attributed to several factors, including potential inaccuracies in manual classification. Future work should further investigate the underlying reasons for these persistent challenges in detecting AD and these specific TD types.

TD Type	Definition
Algorithm Debt	Suboptimal algorithm logic affecting performance (e.g., TODO: Optimise kCostPerUnit).
Compatibility Debt	Compatibility Debt refers to dependencies on immature or temporary solutions.
Design Debt	Design Debt refers to issues like misplaced code, lack of abstraction, or temporary workarounds.
Defect Debt	Defect Debt refers to code that behaves incorrectly or requires fixes but is delayed.
Documentation Debt	Documentation Debt refers to Missing, inadequate, or incomplete documentation and areas which violate good software design practices, causing poor flexibility to evolving business needs
Requirement Debt	Requirement Debt refers to Features or methods that are incomplete or not meeting planned requirements. implementations are left unfinished or in need of future feature support. In the non-functional case, the corresponding code does not meet the requirement standards (speed, memory usage, security).
Test Debt	Refers to deficiencies in test coverage or quality.

Input to Model: Given the following comment: {comment}

Tasks:

1. Identify the type of Technical Debt (e.g., Algorithm, Design, etc.) or state "WITHOUT CLASSIFICATION" if it does not belong to any Technical Debt.
2. If a type is identified, provide a brief explanation of why this comment corresponds to the identified Technical Debt type.

Fig. 4 Prompt design for identifying TD in the GPT-4 experiment. The figure provides definitions TD types, along with tasks assigned to the model for processing the SATD comments

Exploring prompt engineering using GPT-4 While our ML/DL experiments revealed limitations in accurately detecting AD given the low F1-scores, recent advancements in LLMs have shown promise in software engineering tasks, including TD detection (Sheikhaei et al. 2024). Given this, we conducted an exploratory experiment to assess whether prompt engineering with LLMs could AD detection. This experiment was not part of our main evaluation but aimed to provide preliminary insights into the feasibility of LLMs for this task.

Using GPT-4, we designed a custom prompt (Fig. 4) and evaluated its effectiveness using a subset of stratified 100 samples from the testing set that we described in Sect. 3.1. The stratification of the samples ensured a proportional representation of the different TD types, including less frequent AD and Compatibility Debt. This stratified sampling helped to ensure that the evaluation was unbiased and representative. In the prompt, we provided definitions for each TD type, an option for "WITHOUT CLASSIFICATION", and two structured tasks: (i) identifying the TD type and (ii) explaining the rationale for the classification. These components were designed to guide GPT-4 in the classification process.

By analysing the output we observed that for all the AD instances identified by GPT-4, they were correctly classified as AD in the original dataset (i.e., the ground truth), but failed to identify all instances of AD. This indicates perfect precision but a lower recall. An example of a correctly classified SATD as AD is as follows:

> **Comment**: `TODO: Depending on the circumstance, it may be more efficient to parallelise over rows.`
> Ground Truth: ALGORITHM
> LLM Response: 1. The type of Technical Debt in this comment is Algorithm Debt.
> 2. This comment corresponds to Algorithm Debt because it suggests that the current method of parallelisation in the code may not be the most efficient. The comment indicates that depending on the circumstances, it might be more efficient to parallelise over rows, implying that the current algorithm could be optimised for better performance.

The high precision and low recall could likely be due to two reasons i) the number of test samples that we used or ii) because GPT-4 only labels an SATD as AD when it has high confidence, thereby minimising false positives but missing some true instances of AD. While high precision is important, high recall is important to avoid missing instances of AD in DL frameworks. In practice, missing instances of AD could have consequences such as model degradation and poor scalability. Therefore, we suggest that recall should be prioritised as well as precision when designing automated tools for detecting AD. In a practical deployment scenario, this trade-off could be managed by using a high-precision model like GPT-4 as a secondary filter. For example, a high-recall model trained on domain specific features, could first flag a broad set of potential AD instances from code comments. GPT-4 with its high precision, could then be used to validate these instances. This pipeline could be integrated into code review tools or Continuous Integration/Continuous Deployment workflows, allowing developers to efficiently address AD without being overwhelmed by false positives. This would help ensure that AD instances are missed minimally, while maintaining actionable instances for developers.

Similar to the results of the ML/DL experiments, the outputs of the misclassifications by the GPT-4, showed that AD was frequently misclassifed as Design Debt. Though the ground truth was AD, GPT-4 predicted Design Debt, possibly due to the description in the comment focusing on structural and design-related challenges, often associated with Design Debt (Alves et al. 2014). An example of this misclassification is as follows:

> **Comment**: `TODO: We should be able to move instead of copy but it currently isn't straight forward due to redU and redVT being slices`
> Ground Truth: ALGORITHM
> LLM Response: 1. The type of Technical Debt in this comment is Design Debt.
> 2. This comment corresponds to Design Debt because it indicates a design issue in the code where the current structure or design does not allow for a more efficient operation (moving instead of copying). The comment suggests that the current design of handling 'redU' and 'redVT' as slices is causing complications, which is a typical characteristic of Design Debt.

These results from the LLMs, though preliminary, suggested that tailored prompts could enhance their effectiveness for AD detection, similar to previous findings on their use in SATD detection (Sheikhaei et al. 2024). The experiment highlighted i) the potential of GPT-4 in AD detection and ii) the need to improve recall through refined prompt engineering. Future work could explore iterative prompt refinement and expanding the sample size and testing across different datasets for a more comprehensive evaluation of the performance of the GPT-4 for AD detection in DL frameworks.

We compared our approach (LR with Custom AD Features) against GPT-4, using 15 identical SATD comments. Both approaches correctly identified unambiguous SATD comments (e.g., `TODO(nsilberman): Documentation)` for Documentation Debt and also for COMPATIBILITY Debt. A qualitative assessment of the outputs indicates that by and large, the predictions are similar though divergences are observed for AD and implementation Debt. GPT-4 misclassified AD as Design Debt, likely because of structural terms (e.g., "move", "nodes") overlap with design-related language (`TODO: We should be able to move instead of copy but it currently isn't straightforward due to redU and redVT being slices`). Overall, this evaluation highlights that the two approaches produce consistent results but also identifies specific cases where linguistic overlap leads to misclassification. These findings suggest future work to refine feature engineering to better distinguish AD from Implementation and Design Debt. Details of this comparison are in Table 24 of Appendix C.

The overall insights from our preliminary experiments suggested that current ML/DL and LLM approaches have limitations in AD detection in DL frameworks, hindering their real-world efficiency. However, hybrid models integrating domain knowledge and the use of LLMs could offer improvements in performance.

5.1 Limitations

Our study used a dataset of SATD comments drawn from established and, in some cases, historically significant DL frameworks (e.g., Caffe). While this provides valuable insights for foundational analysis, the rapid evolution of the ML/DL ecosystem, including the emergence of more recent frameworks (e.g., HuggingFace Transformers), limits the extent to which our findings can be directly applied to current development practices. In addition, since our evaluation was restricted to a single curated dataset, it remains uncertain how well the custom features would generalise across unseen software projects with different linguistic and structural characteristics. Future work should therefore extend the data collection to include more recent SATD instances from updated versions of widely used frameworks (e.g., TensorFlow 2.x) as well as newly emerging DL frameworks and libraries (e.g., Mamba, DeepSpeed). Such efforts would enable more robust evaluation of the scalability and generalisability of the proposed features.

We note that the programming language of the source code in the dataset we used may bias the findings of this study. Certain keywords, coding patterns, and terminology are language-specific, which could affect the relevance of selected features and the performance of the classifier. Consequently, our results may be more representative of code written in the languages present in our dataset. Evaluating our approach on datasets from additional programming languages constitutes an important direction for future work to assess the generalisability and robustness of our approach.

While AD can exist in various types of software systems beyond DL frameworks, our current focus on AD within DL frameworks is intended to address the gap in the literature on the limited studies on automatic SATD in DL frameworks. Moreover, the study that uncovered AD was in this domain, and the dataset on AD is available, which helps in mitigating the bias of labeling a new dataset. However, we recognise the importance of understanding AD in a broader context. Future research should extend this research to include other software systems to provide a more comprehensive analysis of AD across different domains.

While INSTRUCTOR and VoyageAI were state-of-the-art at the time of initial experimentation, the rapid advancement of language models means newer, potentially more powerful, or code-specific embeddings (e.g., from more recent foundational models, or models specifically pre-trained on code and software engineering text) might offer superior performance. The 3, 200 token context window also represents a constraint given the larger context capabilities of modern LLMs.

Finally, using standard hyperparameter values for the DL models (RoBERTa and ALBERT) could have limited their optimal performance. A more systematic and exhaustive hyperparameter optimisation strategy (e.g., Bayesian optimisation, advanced grid search) would be crucial in future studies to ensure a fairer comparison and maximise model performance.

6 Implications and Future Work

Building on our findings as a first step to AD detection in DL frameworks, this study has implications for researchers, developers, and educators.

For researchers The findings suggest that integrating words related to AD into the dataset and training ML models with this data could improve the performance of AD detection in DL frameworks. However, one potential limitation of the custom feature extraction approach lies in its reliance on domain-specific keywords. By focusing on frequently occurring AD-related terms, there is a risk of overfitting. To mitigate the risk of overfitting, future research should expand the keyword selection process to include datasets from diverse DL projects and also explore AD keywords from different software artefacts such as commits and pull requests. This broader analysis could help validate the generalisability of custom AD features and their effectiveness across diverse DL frameworks.

To improve the generalisation of ML/DL models in AD detection, this research can be expanded by collecting larger and more diverse data across various sources beyond comments and include more DL frameworks. Specifically, incorporating source code alongside SATD comments could improve model performance by providing the models with more context. We hypothesise that this richer context, derived from both comments and code, could improve the performance of the ML/DL models due to the additional information contained in the expanded dataset.

The relatively low F1-scores for AD across the models reflect the limitations in accurately detecting AD. This limitation underscores the need for tailored strategies to enhance AD detection in DL frameworks. Future research could explore a multi-step technique for

AD detection. For example, an initial model identifies AD in a binary fashion, and then subsequent models could target other TD types. This could be in the form of a multiclassifer system of models composed of ML/DL models, where the attention mechanism of DL models can be leveraged for improved performance.

We noted that the low F1-score in both our ML/DL and the GPT-4 experiments was partly due to the conceptual similarity between AD, Design, and Implementation Debt, which highlights the difficulty of accurately distinguishing these TD types. To address this, future efforts could explore multi-stage classification pipelines, where an initial classifier flags general AD in a binary fashion, followed by specialised classifiers to distinguish between TD types such as Design and Implementation Debt. Additionally, hybrid models that integrate rule-based heuristics with ML/DL to incorporate domain knowledge (e.g., AD-related keywords) which are common in AD.

Building on the promising results from GPT-4, future work should be done to explore the effectiveness of LLMs such as GPT-4, LLaMa, and DeepSeek. Researchers should explore iterative prompt refinement to improve techniques on how to improve the recall using these LLMs. One potential direction is leveraging techniques such as chain-of-thought prompting. This approach encouraging step-by-step reasoning, could enhance GPT-4's ability to analyse whether a comment describes an algorithmic inefficiency or a general design issue. By guiding GPT-4 to sequentially assess optimisation-related keywords, performance concerns, and computational complexity indicators, this approach could potentially reduce false negatives in AD detection.

Additionally, few-shot learning strategies could be investigated to better adapt GPT-4 to AD-specific linguistic patterns by providing it with labelled AD examples before classification in low resource datasets. Beyond prompt engineering, integrating hybrid approaches such as combining LLMs with traditional ML models may offer a way to mitigate misclassification patterns and improve overall detection performance for AD in DL systems.

For developers Our findings further indicate that the way developers write AD comments affects automated detection. Comments that are very brief or lack explicit rationale are harder for ML/DL models to classify, highlighting a challenge in accurately detecting AD. An implication of this observation is that ML/DL practitioners should develop and adopt guidelines for reporting AD when using DL frameworks, encouraging more descriptive and structured comments. Educating developers to include key contextual and rationale information in their comments could improve both the performance of automated detection models and the clarity of communication within development teams. Standardising these practices across projects would further enhance consistency and facilitate more efficient detection of AD.

Developers should consider the trade-offs between model simplicity and performance for AD detection. ML models such as LR with domain-specific features, may be suitable for scenarios with limited computational resources or training data. However, the results indicate that such models, while promising with a higher recall (compared to the other models) by leveraging domain understanding, may still produce predictions close to random guessing.

For educators Finally, our findings offer insights for educators teaching ML and software engineering, particularly in the context of DL development. The limitations we encountered in detecting AD underscore the important role of domain understanding and careful feature engineering. Educators can use our results to demonstrate the impact of integrating domain-specific features with even relatively simple models, such as LR, to achieve performance gains. This can be illustrated through hands-on exercises where students are challenged to develop custom features for AD detection and analyse the impact of different feature sets on model performance. By emphasising the importance of understanding both the dataset and the specific characteristics of the problem domain, educators can better prepare students to build effective and tailored models for real-world software engineering tasks, including the critical task of managing AD.

7 Threats to Validity

While designing our study, we considered potential threats to the validity of our research. We took measures to address these threats or mitigated them where possible to ensure the credibility of our findings.

To address concerns regarding **internal validity**, we used an already labelled dataset from previous research (Liu et al. 2020) to train, validate, and test the ML/DL models. This dataset was labelled by the authors who uncovered AD, and demonstrated a high level of inter-rater agreement, (with a Cohen's kappa of over 85%) showing it is of high quality. Using this dataset mitigated any threat that we would have encountered if we performed the labelling ourselves, hence we avoided any potential misclassification of AD. However, we acknowledge that potential biases in the original annotation process could still impact our results.

From Sect. 3.1, the dataset is highly imbalanced, particularly the WITHOUT CLASSIFICATION class, containing a significantly larger number of SATD instances, which could bias model training. To mitigate this issue, we employed a class-weighting strategy to balance the dataset during the training process. Specifically, we set the class weight to 'balanced', assigning higher weights to classes with fewer samples like AD and Compatibility Debt. This balanced approach aimed to improve training robustness and prevent models from favouring the majority class.

We addressed potential overfitting using different approaches. Following ML best practices, we split our data into training, evaluation, and test sets. We then implemented a dedicated training pipeline, applied solely to the training set, to prevent data leakage during training. These combined measures helped to ensure the integrity of our evaluation and minimised the risk of overfitting. During the training of the ML/DL models, we used Grid Search CV to automatically fine-tune hyperparameters and optimise their performance. We used Grid Search CV due to its exhaustive evaluation of hyperparameter evaluation and computational feasibility.

The **external validity** of our study is limited by the dataset we used, which focuses exclusively on DL frameworks. As a result, the generalisability of the findings concerning AD is limited to the domain of DL frameworks and may not extend to traditional software systems that do not incorporate DL components. However, the presence of DL-specific

terminologies in our dataset is expected to enhance model performance for datasets with similar linguistic characteristics.

A potential limitation of the LR with Custom AD Features lies in the reliance on the custom AD keywords. By focusing on frequently occurring AD-related terms from the dataset, the model's performance may degrade when applied to datasets with different characteristics. To mitigate this, we took a systematic approach: a list of keywords was first automatically generated by the first author using frequency-based text analysis techniques, and the top five words were then selected by a TD expert. This list was subsequently discussed among the authors.

To address threats to **construct validity** related to our imbalanced dataset, we used appropriate evaluation metrics when comparing ML/DL model performance. Acknowledging that accuracy can be misleading in such cases favouring the majority class (Batista et al. 2004), we evaluated our models using precision, recall, F1-score, and macro-averaged F1-score, consistent with prior studies on SATD detection (Ren et al. 2019; Santos et al. 2020; Sharma et al. 2022). These metrics are particularly relevant as they provide a more accurate assessment of the performance of minority classes such as AD.

To address the threat to **conclusion validity** due to potentially statistically insignificant results when comparing ML/DL model performance, we used statistical tests. We applied the Wilcoxon Signed-Rank test and Cliff's Delta to evaluate whether the differences we observed in the F1-scores between the LR with custom AD features and the other ML/DL models are statistically significant, rather than due to random variations in the dataset. By using these tests, we mitigated the possibility of over-interpreting the differences we observed and strengthened the statistical basis of our conclusions.

8 Conclusion

In this study, we investigated the performance of ML/DL models for the automated detection of AD in DL frameworks. We demonstrated the potential of incorporating custom features to enhance the performance of ML/DL models. The improved performance of LR (an ML model) with custom features highlighted the value of tailored feature engineering for AD detection. However, our best F1-score of 54%, while an improvement over baselines, remains insufficient for practical adoption; the misclassifications between AD, Design and Implementation Debt suggests that finer-grained approaches such as hierarchical classification or subcategory annotation may be required. Furthermore, our exploratory experiment with GPT-4 revealed the potential of LLMs in AD detection, although further refinement is needed. These results suggest that both custom features and LLMs offer promising avenues for automated AD detection in DL frameworks. Future research should develop more generalisable AD-specific features applicable across diverse DL frameworks and include source code in the dataset. Researchers should also explore prompt engineering techniques to improve AD recall in LLMs. Finally, future work should investigate the conceptual overlaps between AD and other TD types to develop more accurate automated detection tools to help developers using DL frameworks to flag AD in real time. Addressing these challenges will pave the way for more effective TD management, leading to faster development cycles and more reliable DL systems.

A Appendix A: Hyperparameter Tuning

A.1 Hyperparameter Tuning

Hyperparameters play an important role in the performance of ML/DL models. To optimise these hyperparameters and enhance classification performance, we used the Grid Search CV, a technique that systematically fine-tunes hyperparameters to improve the outcomes of models (Bergstra et al. 2011).

For the **SVM**, we conducted experiments with different kernels, including *RBF, linear, and polynomial*, to determine which kernel function provides the best decision boundaries for the SATD data. These SVM kernels offer a diverse range of capabilities: the *RBF* kernel is effective for capturing non-linear patterns. It is particularly effective for datasets where the separation between classes is not linear or polynomial. The *linear* kernel serves as a baseline for linearly separable data, where no explicit mapping to a higher-dimensional space is performed, and the *polynomial* kernel allows for the modeling of curved decision boundaries by exploration of polynomial relationships. From our experiments, the linear kernel performed better than other kernels in terms of F1-scores.

For the **LR** model, we conducted experiments using the following hyperparameters: *C*, *penalty*, *Max_iter*. *C* controls the regularisation strength of the model, by playing a role in balancing underfitting and overfitting. Regularisation helps prevent overfitting by penalising larger coefficients, which can make the model more generalised to unseen data. Smaller values of *C* indicate stronger regularisation, while larger values allow the model more flexibility in fitting the data. The range values we used were 0.1, 1, and 10, allowing us to explore a range of regularisations. These values were chosen to find the right balance between model complexity and generalisation performance.

The *penalty* hyperparameter specifies the type of regularisation to be applied to the LR model. We used the L2 regularisation, which penalises the sum of the squared coefficients. L2 regularisation is commonly used in LR classification tasks as it tends to reduce over-reliance on any single feature while retaining all features in the model. This helps the model generalise better and reduces the risk of overfitting. Consequently we selected L2 due to its effectiveness in prior studies on similar classification tasks and its computational efficiency compared to alternatives like L1.

The *max_iter* hyperparameter controls the maximum number of iterations the solver can take to converge. LR uses iterative solvers to find the optimal weights for the model, and sometimes these solvers require different iterations to converge, particularly with complex data. The range that we selected were values of 1, 10, 100, and 200, to enable convergence while balancing computation time. These values were chosen to cover minimal iterations as well as extended runs for complex data patterns across various configurations.

Similarly, for the **RF** classifier, Grid Search CV experiments were conducted with different hyperparameters: *n_estimators*, *max_depth*, *min samples split*, *min_samples_leaf*. We explored a range of experimental values for these hyperparameters. The *n_estimators*, defines the number of trees in the RF model. More trees generally reduce the variance of the model, making it more stable and accurate. However, too many trees can increase computational time and memory usage. The grid included values of 100 and 200 to strike a balance between having enough trees for accurate predictions and maintaining computational efficiency. We used these values to explore whether increasing the number of trees provides performance gains.

The *max_depth* controls how deep each tree in the forest can grow. Deeper trees can capture more complex patterns but may also lead to overfitting. On the other hand, shallow trees might underfit and miss important patterns. To strike a balance for computational efficiency, the range of values we selected include: None, 10, 20, and 30, ensuring that trees could capture complex patterns without excessive computational cost or overfitting.

The *min_samples_split* controls the minimum number of samples required to split an internal node. Setting a high value prevents the tree from splitting too frequently, which can reduce overfitting by limiting the complexity of the tree. A smaller value allows for more frequent splits, creating more complex trees that can capture intricate patterns but may lead to overfitting. The grid included None, 2, 5, and 10 to explore how varying the minimum samples for splitting affects model performance.

The *min_samples_leaf* sets the minimum number of samples required to be in a leaf node. This prevents the model from creating overly specific leaves that could overfit the training data. Smaller values, such as 1, allow leaves to be created from very few samples, potentially increasing model complexity. Increasing this value to 2 helps avoid overfitting by requiring more samples to form a leaf. The grid included values 1 and 2 to explore whether smaller or slightly larger leaves would lead to better model generalisation. These values provide a controlled way to help guarantee the model does not overfit by limiting the complexity of the tree branches.

After running the Grid Search CV across these configurations, the optimal hyperparameters for each ML classifier are summarised in Table 7. It is important to note that unlike ML models, multiple experiments were not conducted for the DL models. This is because DL models have a vast number of hyperparameters, and exhaustive hyperparameter tuning can become computationally expensive (Joulin et al. 2017).

Table 7 Final hyperparameter configuration for for the ML models (LR, SVM, and RF)

LR	SVM	RF
C = 10	RBF	Max depth = 'None'
Max Iter = 100	Linear	Min Samples split = 2
Penalty = l2	Polynomial	Min samples leaf = 1
		n_ estimators = 200

B Appendix B: Model Classification Reports and Confusion Matrix

This appendix contains the classification report for the models we evaluated models in our study. The report includes metrics: precision, recall, F1-scores, macro-averaged F1-scores, and accuracy for each model in percentage. These results provide an overview of the performance of the models across the different TD types to support the findings discussed in the main text (Tables 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22 and 23).

Table 8 Classification report for LR with Custom AD features

TD Type	Precision	Recall	F1-score	Support
ALGORITHM	51	56	54	200
COMPATIBILITY	56	55	55	89
DEFECT	50	58	53	135
DESIGN	86	82	84	2,206
DOCUMENTATION	62	57	59	23
IMPLEMENTATION	63	75	69	387
TEST	71	76	74	143
WITHOUT_CLASSIFICATION	97	96	96	4,592
Accuracy	88			7,775
Macro avg	67	69	68	7,775
Weighted avg	89	88	88	7,775

Table 9 Classification report for SVM with TF-IDF

TD Type	Precision	Recall	F1-score	Support
ALGORITHM	54	38	44	200
COMPATIBILITY	57	39	47	89
DEFECT	45	36	40	135
DESIGN	77	79	78	2,206
DOCUMENTATION	62	43	51	23
IMPLEMENTATION	61	53	57	387
TEST	75	36	49	143
WITHOUT_CLASSIFICATION	91	95	93	4,592
Accuracy	84			7,775
Macro avg	66	53	57	7,775
Weighted avg	83	84	83	7,775

Table 10 Classification report for SVM + hash vectoriser

TD Type	Precision	Recall	F1-score	Support
ALGORITHM	49	38	43	200
COMPATIBILITY	62	35	45	89
DEFECT	37	37	37	135
DESIGN	78	80	79	2,206
DOCUMENTATION	50	30	38	23
IMPLEMENTATION	59	56	58	387
TEST	72	40	51	143
WITHOUT_CLAS-SIFICATION	92	95	93	4,592
Accuracy	84			7,775
Macro avg	62	51	55	7,775
Weighted avg	84	84	84	7,775

Table 11 Classification report for SVM + count vectoriser

TD Type	Precision	Recall	F1-score	Support
ALGORITHM	37	46	41	200
COMPATIBILITY	31	42	35	89
DEFECT	29	43	35	135
DESIGN	83	76	79	2,206
DOCUMENTATION	27	39	32	23
IMPLEMENTATION	60	60	60	387
TEST	62	74	68	143
WITHOUT_CLAS-SIFICATION	95	95	95	4,592
Accuracy	85			7,775
Macro avg	53	59	56	7,775
Weighted avg	86	85	85	7,775

Table 12 Classification report for RF +TFIDF

TD Type	Precision	Recall	F1-score	Support
ALGORITHM	96	27	42	200
COMPATIBILITY	97	35	51	89
DEFECT	88	27	41	135
DESIGN	79	90	84	2,206
DOCUMENTATION	86	26	40	23
IMPLEMENTATION	88	51	65	387
TEST	90	48	63	143
WITHOUT_CLAS-SIFICATION	93	98	96	4,592
Accuracy	89			7,775
Macro avg	90	50	60	7,775
Weighted avg	89	89	87	7,775

Table 13 Classification report for RF + hash

TD Type	Precision	Recall	F1-score	Support
ALGORITHM	91	30	46	200
COMPATIBILITY	91	36	52	89
DEFECT	81	29	43	135
DESIGN	78	90	84	2,206
DOCUMENTATION	88	30	45	23
IMPLEMENTATION	83	54	65	387
TEST	91	43	58	143
WITHOUT_CLASSIFICATION	93	97	95	4,592
Accuracy	88			7,775
Macro avg	87	51	61	7,775
Weighted avg	88	88	87	7,775

Table 14 Classification report for RF + count vectoriser

TD Type	Precision	Recall	F1-score	Support
ALGORITHM	92	35	51	200
COMPATIBILITY	86	43	57	89
DEFECT	75	42	54	135
DESIGN	81	89	85	2,206
DOCUMENTATION	79	48	59	23
IMPLEMENTATION	71	66	69	387
TEST	78	64	70	143
WITHOUT_CLASSIFICATION	95	97	96	4,592
Accuracy	89			7,775
Macro avg	82	60	68	7,775
Weighted avg	89	89	89	7,775

Table 15 Classification report for LR + TF-IDF

TD Type	Precision	Recall	F1-score	Support
ALGORITHM	54	44	48	200
COMPATIBILITY	60	48	53	89
DEFECT	47	44	45	135
DESIGN	80	79	80	2,206
DOCUMENTATION	40	52	45	23
IMPLEMENTATION	59	59	59	387
TEST	59	48	53	143
WITHOUT_CLASSIFICATION	93	95	94	4,592
Accuracy	85			7,775
Macro avg	61	59	60	7,775
Weighted avg	84	85	85	7,775

Table 16 Classification report for LR + hash vectoriser

TD Type	Precision	Recall	F1-score	Support
ALGORITHM	53	44	48	200
COMPATIBILITY	56	46	51	89
DEFECT	44	43	43	135
DESIGN	81	80	81	2,206
DOCUMENTATION	43	52	47	23
IMPLEMENTATION	59	61	60	387
TEST	60	51	55	143
WITHOUT_CLASSIFICATION	93	94	94	4,592
Accuracy	85			7,775
Macro avg	61	59	60	7,775
Weighted avg	85	85	85	7,775

Table 17 Classification report for LR + count vectoriser

TD Type	Precision	Recall	F1-score	Support
ALGORITHM	53	50	52	200
COMPATIBILITY	48	49	49	89
DEFECT	45	53	49	135
DESIGN	84	80	82	2,206
DOCUMENTATION	36	43	39	23
IMPLEMENTATION	59	69	63	387
TEST	62	66	64	143
WITHOUT_CLASSIFICATION	96	96	96	4,592
Accuracy	87			7,775
Macro avg	60	63	62	7,775
Weighted avg	87	87	87	7,775

Table 18 Classification report for RoBERTa

TD Type	Precision	Recall	F1-Score	Support
ALGORITHM	55	36	43	200
COMPATIBILITY	59	44	50	89
DEFECT	58	47	52	135
DESIGN	82	89	85	2,206
DOCUMENTATION	71	43	54	23
IMPLEMENTATION	78	60	68	387
TEST	82	78	80	143
WITHOUT_CLASSIFICATION	96	97	97	4,592
Accuracy	89			7,775
Macro Avg	73	62	66	7,775
Weighted Avg	89	89	89	7,775

Table 19 Classification report for RoBERTa embeddings

TD Type	Precision	Recall	F1-score	Support
ALGORITHM	49	39	44	200
COMPATIBILITY	47	53	50	89
DEFECT	35	56	43	135
DESIGN	85	84	84	2,206
DOCUMENTATION	10	48	16	23
IMPLEMENTATION	75	64	69	387
TEST	72	80	76	143
WITHOUT_CLASSIFICATION	97	96	96	4,592
Accuracy	88			7,775
Macro Avg	59	65	60	7,775
Weighted Avg	89	88	88	7,775

Table 20 Classification report for ALBERT

TD Type	Precision	Recall	F1-Score	Support
ALGORITHM	29	16	21	187
COMPATIBILITY	40	02	04	91
DEFECT	31	22	26	132
DESIGN	79	87	83	2,178
DOCUMENTATION	00	00	00	32
IMPLEMENTATION	63	64	64	393
TEST	72	63	67	131
WITHOUT_CLASSIFICATION	96	97	97	4,631
Accuracy	87			7,775
Macro Avg	51	44	45	7,775
Weighted Avg	86	87	86	7,775

Table 21 Classification report for ALBERT Embeddings

TD Type	Precision	Recall	F1-Score	Support
ALGORITHM	23	28	25	187
COMPATIBILITY	29	22	25	91
DEFECT	18	45	25	132
DESIGN	85	76	80	2,178
DOCUMENTATION	03	28	05	32
IMPLEMENTATION	75	51	61	393
TEST	71	64	67	131
WITHOUT_CLASSIFICATION	98	95	96	4,631
Accuracy	83			7,775
Macro Avg	50	51	48	7,775
Weighted Avg	88	83	85	7,775

Table 22 Classification report for INSTRUCTOR embeddings

TD Type	Precision	Recall	F1-Score	Support
ALGORITHM	21	51	29	200
COMPATIBILITY	22	52	31	89
DEFECT	24	45	31	135
DESIGN	79	62	69	2,206
DOCUMENTATION	25	61	36	23
IMPLEMENTATION	42	65	51	387
TEST	50	64	56	143
WITHOUT_CLAS-SIFICATION	96	89	93	4,592
Accuracy	78			7,775
Macro Avg	45	61	50	7,775
Weighted Avg	84	78	80	7,775

Table 23 Classification report for RoBERTa and AD features

TD Type	Precision	Recall	F1-Score	Support
ALGORITHM	0.61	0.41	0.49	200
COMPATIBILITY	0.56	0.55	0.56	89
DEFECT	0.61	0.53	0.57	135
DESIGN	0.82	0.89	0.85	2,206
DOCUMENTATION	0.82	0.39	0.53	23
IMPLEMENTATION	0.74	0.64	0.69	387
TEST	0.76	0.79	0.78	143
WITHOUT_CLASSIFI-CATION	0.97	0.96	0.97	4,592
Accuracy	0.89			7,775
Macro Avg	0.74	0.65	0.68	7,775
Weighted Avg	0.89	0.89	0.89	7,775

C Appendix C: Comparison of LLM and LR with Custom AD Features

Table 24 presents a side-by-side comparison of TD for 15 selected SATD comments. We compared our proposed approach (LR with Custom AD Features) against GPT-4 using the same set of comments. The table shows the *Ground Truth*, the LLM predictions, and our LR with Custom AD Features. This comparison highlights areas of agreement and divergence between the approaches. Both methods correctly identify unambiguous TD comments (e.g., for COMPATIBILITY, DESIGN, and DOCUMENTATION). Divergences primarily occur for AD and IMPLEMENTATION Debt, where GPT-4 sometimes misclassifies AD as Design Debt, likely due to terms such as “move”.

Table 24 Comparison of TD classification: ground truth, LLM Prediction, and LR with custom features predictions

Comment	Ground Truth	LLM Prediction	LR + Custom Features Prediction
Adjust learning per minibatches at very beginning of training process this could be used to tackle the unstableness of ASGD	ALGORITHM	ALGORITHM	ALGORITHM
TODO: We should be able to move instead of copy but it currently isn't straightforward due to redU and redVT being slices	ALGORITHM	DESIGN	ALGORITHM
Hack for the tracer that allows us to represent RNNs as singlenodes and export them to ONNX in this form	ALGORITHM	DESIGN	ALGORITHM
Linux gcc barfs on this 'us = (double)((std::wstring)arg). size();' due to some ambiguity error	COMPATIBILITY	COMPATIBILITY	COMPATIBILITY
TODO: fix libname for OSX / Windows TODO: just load 5.1, not 5.1.3 TODO: dynamic version checks via cudnnGetVersion	COMPATIBILITY	COMPATIBILITY	COMPATIBILITY
/* TODO: remove the extra copies of the input. These are only * used for debugging purposes during development and testing. */	DESIGN	DESIGN	DESIGN
/*TODO: merge with call site*/ void BackpropToLeftS(Matrix ElemType & input1FunctionValues...	DESIGN	DESIGN	DESIGN
TODO vectorize mixed product	DESIGN	ALGORITHM	IMPLEMENTATION
TODO(b/73448937): Move all update damping code to a separate class/function	DESIGN	DESIGN	DESIGN
This really should be done in an external debugging tool	DESIGN	DESIGN	DESIGN
ReshapeNode – reshape input matrix TODO: Why is this in NonlinearityNodes.h? Should be linear algebra no?	DESIGN	DESIGN	DESIGN
TODO(nsilberman): Documentation	DOCUMENTATION	DOCUMENTATION	DOCUMENTATION
`/*l*/) todo: add assertion`	IMPLEMENTATION	TEST	IMPLEMENTATION
TODO: add loading from checkpoint	IMPLEMENTATION	IMPLEMENTATION	IMPLEMENTATION
TODO(satok): Implement all possible cases	IMPLEMENTATION	IMPLEMENTATION	IMPLEMENTATION

Author Contributions Emmanuel Iko-Ojo Simon designed the study, conducted the experiments, verified the results, analysed and interpreted the data, and wrote both the initial draft and the final manuscript. Chirath Hettiarachchi contributed to the study design, verified the results, reviewed and edited the final manuscript, and provided supervisory guidance. Alex Potanin contributed to the study design, reviewed and edited the final manuscript, and provided supervisory guidance. Hanna Suominen contributed to the study design, reviewed and edited both the draft and final manuscript, and provided supervisory guidance. Fatemeh Fard contributed to the study design, verified the experiments, and reviewed and edited both the draft and final manuscript, and provided supervisory guidance. All authors read and approved the final manuscript.

Funding Open Access funding enabled and organized by CAUL and its Member Institutions. This work is supported by the Australian National University (ANU) through the ANU PhD scholarship within the ANU Research School of Computing.

Data Availability To ensure the replicability of our work and to adhere to ACM's open access policy, we have created an online repository on GitHub (https://github.com/ikoojos/Algorithm-Debt-Research/tree/master), that contains the models and the dataset we used for training the ML/DL models.

Declarations

Conflict of interest The authors declare that they have no Conflict of interest.

Ethical Approval Not applicable.

Informed Consent Not applicable.

Clinical Trial Number Clinical trial number: Not applicable.

References

AlOmar EA, Christians B, Busho M, AlKhalid AH, Ouni A, Newman C (2022) Satdbailiff-mining and tracking self-admitted technical debt. Sci Comput Program 213:102693

Alves N, Ribeiro LF, Caires V, Mendes TS, Spínola RO (2014) Towards an ontology of terms on technical debt. Int. Workshop on Managing Technical Debt, IEEE, pp 1–7

Ameisen E (2021) Building Machine Learning Powered Applications: Going from Idea to Product. O'Reilly Media

Arisholm E, Briand LC, Fuglerud M (2007) Data mining techniques for building fault-proneness models in telecom java software. In: 18th IEEE International Symposium on Software Reliability, IEEE, pp 215–224

Arsyah UI, Pratiwi M, Muhammad A (2024) Twitter sentiment analysis of public space opinions using svm and tf-idf methods. Indonesian Journal of Computer Science 13(1)

Avgeriou P, Kruchten P, Ozkaya I, Seaman C (2016) Managing technical debt in software engineering (dagstuhl seminar 16162). In: Dagstuhl Reports, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, vol 6

Ayyagari M, Bhatt VD, Pandey A (2022) Efficient implementation of pooling operation for ai accelerators. In: 2022 IEEE International Conference for Women in Innovation, Technology & Entrepreneurship (ICWITE), IEEE, pp 1–5

Azuma H, Matsumoto S, Kamei Y, Kusumoto S (2022) An empirical study on self-admitted technical debt in dockerfiles. Empir Softw Eng 27(2):49
Batista GE, Prati RC, Monard MC (2004) A study of the behavior of several methods for balancing machine learning training data. ACM SIGKDD Explorations Newsl 6(1):20–29
Bavota G, Russo B (2016) A large-scale empirical study on self-admitted technical debt. In: Proceedings of the 13th international conference on mining software repositories, pp 315–326
Bergstra J, Bardenet R, Bengio Y, Kégl B (2011) Algorithms for hyper-parameter optimization. Advances in Neural Information Processing Systems 24
Bertsimas D, King A (2017) Logistic regression: From art to science. Statistical Science pp 367–384
Bhatia A, Khomh F, Adams B, Hassan AE (2024) An empirical study of self-admitted technical debt in machine learning software. ACM Transaction on Software Engineering Methodologies 1(1)
Biau G, Scornet E (2016) A random forest guided tour. TEST 25:197–227
Breiman L (2001) Random forests Machine learning 45:5–32
Cohen J (1960) A coefficient of agreement for nominal scales. Educ Psychol Measur 20(1):37–46
Cortes C, Vapnik V (1995) Support-vector networks Machine learning 20:273–297
Cunningham W (1992) The wycash portfolio mgt system. ACM SIGPLAN OOPS 4:29–30
da Silva ME, Shihab E, Tsantalis N (2017) Using natural language processing to automatically detect satd. IEEE Trans Software Eng 43(11):1044–1062
Danyal MM, Khan SS, Khan M, Ullah S, Ghaffar MB, Khan W (2024) Sentiment analysis of movie reviews based on nb approaches using tf-idf and count vectorizer. Soc Netw Anal Min 14(1):1–15
Deepa D, Tamilarasi A et al (2019) Sentiment analysis using feature extraction and dictionary-based approaches. 2019 Third International conference on I-SMAC (IoT in Social. Mobile, Analytics and Cloud)(I-SMAC), IEEE, pp 786–790
Devlin J, Chang MW, Lee K, Toutanova K (2019) BERT: Pre-training of deep bidirectional transformers for language understanding. In: Burstein J, Doran C, Solorio T (eds) Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), Association for Computational Linguistics, Minneapolis, Minnesota, pp 4171–4186, https://doi.org/10.18653/v1/N19-1423, https://aclanthology.org/N19-1423/
Ebrahimi AM, Oliva GA, Hassan AE (2023) Self-admitted technical debt in ethereum smart contracts: A large-scale exploratory study. IEEE Trans Softw Eng
Feuerriegel S, Maarouf A, Bär D, Geissler D, Schweisthal J, Pröllochs N, Robertson CE, Rathje S, Hartmann J, Mohammad SM, et al (2025) Using natural language processing to analyse text data in behavioural science. Nat Rev Psychol 1–16
Foody GM (2023) Challenges in the real world use of classification accuracy metrics: From recall and precision to the matthews correlation coefficient. PLoS One 18(10):e0291908
Ghasemi N, Justo JA, Celesti M, Despoisse L, Nieke J (2025) Onboard processing of hyperspectral imagery: Deep learning advancements, methodologies, challenges, and emerging trends. IEEE J Sel Top Appl Earth Observ Remote Sens
Goodfellow I, Bengio Y, Courville A (2016) Deep Learning, 1st edn. MIT Press, Cambridge, MA. https://www.deeplearningbook.org/
Guyon I, Gunn S, Nikravesh M, Zadeh LA (2008) Feature extraction: foundations and applications, vol 207. Springer
Hilbe JM (2009) Logistic regression models. Chapman and hall/CRC
Huang Q, Shihab E, Xia X, Lo D, Li S (2018) Identifying self-admitted technical debt in open source projects using text mining. Empir Softw Eng 23:418–451
James G, Witten D, Hastie T, Tibshirani R et al (2013) An introduction to statistical learning, vol 112. Springer
Joachims T (1998) Text categorization with support vector machines: Learning with many relevant features. In: European Conference on Machine Learning, Springer, pp 137–142
Joulin A, Grave E, Bojanowski P, Mikolov T (2017) Bag of tricks for efficient text classification. In: Lapata M, Blunsom P, Koller A (eds) Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers, Association for Computational Linguistics, Valencia, Spain, pp 427–431. https://aclanthology.org/E17-2068/
Kaur J, Buttar PK (2018) A systematic review on stopword removal algorithms. Int J Future Revol Comput Sci Commun Eng 4(4):207–210
Khan AK (2024) Ai in finance disruptive technologies and emerging opportunities. J Artif Intell Gen Sci (JAIGS) 3(1):155–170
Khoshgoftaar TM, Golawala M, Van Hulse J (2007) An empirical study of learning from imbalanced data using random forest. In: 19th IEEE international conference on tools with artificial intelligence (ICTAI 2007), IEEE, vol 2, pp 310–317

Lan Z, Chen M, Goodman S, Gimpel K, Sharma P, Soricut R (2019) Albert: A lite bert for self-supervised learning of language representations. In: 2020 International conference on learning representations, ICLR '20

Lemieux C, Inala JP, Lahiri, Sen S (2023) Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In: 2023 IEEE/ACM 45th international conference on software engineering (ICSE), IEEE, pp 919–931

Lenarduzzi V, Saarimäki N, Taibi D (2019) The technical debt dataset. In: Proceedings of the fifteenth international conference on predictive models and data analytics in software engineering, pp 2–11

Li Y, Soliman M, Avgeriou P (2023) Automatic identification of self-admitted technical debt from four different sources. Empir Softw Eng 28(3):1–38

Li J, Li L, Liu J, Yu X, Liu X, Keung JW (2025) Large language model chatgpt versus small deep learning models for self-admitted technical debt detection: Why not together? Softw Pract Exper 55(1):3–28

Li J, Mani G (2021) Machine learning application on prediction of male breast cancer with plco dataset. J Student Res 10(3). https://doi.org/10.47611/jsrhs.v10i3.2199

Li Y, Soliman M, Avgeriou P (2023b) Automatically identifying relations between self-admitted technical debt across different sources. In: 2023 ACM/IEEE international conference on technical debt (TechDebt), pp 11–21. https://doi.org/10.1109/TechDebt59074.2023.00008

Liu J, Huang Q, Xia X, Shihab E, Lo D, Li S (2021) An exploratory study on the introduction and removal of different types of technical debt in deep learning frameworks. Empir Softw Eng 26:1–36

Liu M, Wang J, Lin T, Ma Q, Fang Z, Wu Y (2024) An empirical study of the code generation of safety-critical software using llms. Appl Sci 14(3):1046

Liu J, Huang Q, Xia X, Shihab E, Lo D, Li S (2020) Is using deep learning frameworks free? characterizing technical debt in deep learning frameworks. In: ACM/IEEE 42nd int conf on software engineering, pp 1–10

Liu Z, Lin W, Shi Y, Zhao J (2021b) A robustly optimized bert pre-training approach with post-training. In: Chinese Computational Linguistics: 20th China National Conference, CCL 2021, Hohhot, China, August 13–15, 2021, Proceedings, Springer-Verlag, Berlin, Heidelberg, pp 471–484. https://doi.org/10.1007/978-3-030-84186-7_31https://doi.org/10.1007/978-3-030-84186-7_31

Loureiro D (2023) Learning word sense representations from neural language models. PhD thesis, Universidade do Porto (Portugal)

Macbeth G, Razumiejczyk E, Ledesma RD (2011) Cliff's delta calculator: A non-parametric effect size program for two groups of observations. Universitas Psychologica 10(2):545–555

Maldonado EdS, Abdalkareem R, Shihab E, Serebrenik A (2017) An empirical study on the removal of self-admitted technical debt. In: International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 238–248

Maldonado EdS, Shihab E (2015) Detecting and quantifying different types of self-admitted technical debt. In: 7th International Workshop on Managing Technical Debt, IEEE, pp 9–15

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 (ASE), IEEE, pp 585–597

Meissel K, Yao ES (2024) Using cliff's delta as a non-parametric effect size measure: an accessible web app and r tutorial. Practical Assessment, Research, and Evaluation 29(1)

Mining WID (2006) Data mining: Concepts and techniques. Morgan Kaufinann 10(559–569):4

Moreschini S, Lenarduzzi V, Coba L (2024) Towards a technical debt for ai-based recommender system. In: Proceedings of the 7th ACM/IEEE International Conference on Technical Debt, Association for Computing Machinery, New York, NY, USA, TechDebt '24, pp 36–39. https://doi.org/10.1145/3644384.3648574

Musa AB (2013) Comparative study on classification performance between support vector machine and logistic regression. Int J Mach Learn Cybern 4:13–24

Nunes RO, Spritzer AS, Balreira DG, Freitas CM, Carbonera JL (2024) An evaluation of large language models for geological named entity recognition. IEEE

OBrien D, Biswas S, Imtiaz S, Abdalkareem R, Shihab E, Rajan H (2022) 23 shades of self-admitted technical debt: an empirical study on machine learning software. In: Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering, pp 734–746

Ozkaya I (2023) Application of llms to software engineering tasks: Opportunities, risks, and implications. IEEE Softw 40(3):4–8

Pepe F, Zampetti F, Mastropaolo A, Bavota G, Di Penta M (2024) A taxonomy of self-admitted technical debt in deep learning systems. In: 2024 IEEE international conference on software maintenance and evolution (ICSME), pp 388–399

Peruma A, Simmons S, AlOmar EA, Newman CD, Mkaouer MW, Ouni A (2022) How do i refactor this? an empirical study on refactoring trends and topics in stack overflow. Empir Softw Eng 27(1):11. https://doi.org/10.1007/s10664-021-10045-x
Pinna A, Lunesu MI, Orrù S, Tonelli R (2023) Investigation on self-admitted technical debt in open-source blockchain projects. Future Int 15(7):232
Poczeta K, Płaza M, Michno T, Krechowicz M, Zawadzki M (2023) A multi-label text message classification method designed for applications in call/contact centre systems. Appl Soft Comput 145
Potdar A, Shihab E (2014) An exploratory study on self-admitted technical debt. In: Int conference on software maintenance and evolution, pp 91–100
Rajapaksha P, Farahbakhsh R, Crespi N (2021) Bert, xlnet or roberta: the best transfer learning model to detect clickbaits. IEEE Access 9:154704–154716
Rantala L, Lenarduzzi V, Mäntylä MV (2024) Relationship between self-admitted technical debt and code-level technical debt. an empirical evaluation. Softw Quality Jornal. https://www.researchgate.net/publication/374998443_Relationship_between_Self-Admitted_Technical_Debt_and_Code-level_Technical_Debt_An_Empirical_Evaluation
Recupito G, Pecorelli F, Catolino G, Lenarduzzi V, Taibi D, Di Nucci D, Palomba F (2024) Technical debt in ai-enabled systems: On the prevalence, severity, impact, and management strategies for code and architecture. J Syst Softw 112151
Rejani Y, Selvi ST (2009) Early detection of breast cancer using svm classifier technique. Int J Comput Eng 1(3):127–130
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 Trans Softw Eng Method (TOSEM) 28(3):1–45
Rios N, de MendonCa Neto MG, SpInola RO (2018) A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. Inf Softw Technol 102:117–145
Roshan R, Bhacho IA, Zai S (2023) Comparative analysis of tf-idf and hashing vectoriser for fake news detection in sindhi: A machine learning and deep learning approach. Eng Proc 46(1):5
Sabbah AF, Hanani AA (2023) Self-admitted technical debt classification using natural language processing word embeddings. Int J Electr Comp Engr 13(2)
Sallou J, Durieux T, Panichella A (2024) Breaking the silence: the threats of using llms in software engineering. In: Proceedings of the 2024 ACM/IEEE 44th International conference on software engineering: new ideas and emerging results, pp 102–106
Santos RM, Santos IM, Júnior MCR, de Mendonça Neto MG (2020) Long term-short memory neural networks and word2vec for self-admitted technical debt detection. In: International conference on enterprise information systems (ICEIS), pp 157–165
Sarker IH (2021) Machine learning: Algorithms, real-world applications and research directions. SN Comput Sci 2(3):160
Sculley D, Holt G, Golovin D, Davydov E, Phillips T, Ebner D, Chaudhary V, Young JF, Dennison D (2015) Hidden technical debt in machine learning systems. Adv Neural Inf Process Syst 28
Shah K, Patel H, Sanghvi D, Shah M (2020) A comparative analysis of logistic regression, random forest and knn models for the text classification. Augmented Human Res 5(1):12
Sharma R, Shahbazi R, Fard FH, Codabux Z, Vidoni M (2022) Self-admitted technical debt in r: detection and causes. Autom Softw Eng 29(2):53
Sheikhaei MS, Tian Y, Wang S, Xu B (2024) An empirical study on the effectiveness of large language models for satd identification and classification. Empir Softw Eng 29(6):159
Simon EIO, Hettiarachchi C, Potanin A, Suominen H, Fard F (2024) Automated detection of Algorithm Debt in deep learning frameworks: An empirical study. Presented at the 40th IEEE international conference on software maintenance and evolution (ICSME), Registered Reports Track. https://doi.org/10.48550/arXiv.2408.10529
Simon EIO, Vidoni M, Fard FH (2023) Algorithm debt: Challenges and future paths. In: 2023 IEEE/ACM 2nd international conference on AI engineering–software engineering for AI, IEEE, pp 90–91
Sklavenitis D, Kalles D (2024) Measuring technical debt in ai-based competition platforms. In: Proceedings of the 13th hellenic conference on artificial intelligence, pp 1–10
Suominen H, Pahikkala T, Salakoski T (2008) Critical points in assessing learning performance via cross-validation. In: Proceedings of the 2nd international and interdisciplinary conference on adaptive knowledge representation and reasoning (AKRR 2008), Helsinki University of Technology, pp 9–22
Suryaningrum KM (2023) Comparison of the tf-idf method with the count vectorizer to classify hate speech. Eng Math Comput Sci J (EMACS) 5(2):79–83

Su H, Shi W, Kasai J, Wang Y, Hu Y, Ostendorf M, Yih Wt, Smith NA, Zettlemoyer L, Yu T (2023) One embedder, any task: Instruction-finetuned text embeddings. In: Findings of the association for computational linguistics: ACL 2023

Sutoyo E, Avgeriou P, Capiluppi A (2024) Deep learning and data augmentation for detecting self-admitted technical debt. In: 31st Asia-pacific software engineering conference (APSEC 2024)

Sutoyo E, Capiluppi A (2024) Satdaug - a balanced and augmented dataset for detecting self-admitted technical debt. In: Proceedings of the 21st international conference on mining software repositories, association for computing machinery, New York, NY, USA, MSR '24, pp 289–293. https://doi.org/10.1145/3643991.3644880

Tang Y, Khatchadourian R, Bagherzadeh M, Singh R, Stewart A, Raja A (2021) An empirical study of refactorings and technical debt in machine learning systems. In: 2021 IEEE/ACM 43rd international conference on software engineering (ICSE), IEEE, pp 238–250

Tang Y, Yang Y (2024) Multihop-RAG: Benchmarking retrieval-augmented generation for multi-hop queries. In: First conference on language modeling. https://openreview.net/forum?id=t4eB3zYWBK

Vargha A, Delaney HD (2000) A critique and improvement of the cl common language effect size statistics of mcgraw and wong. J Educ Behav Stat 25(2):101–132

Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Lu, Polosukhin I (2017) Attention is all you need. In: Advances in neural information processing systems, curran associates, Inc., vol 30. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

Venkatkrishna V, Nagabushanam DS, Simon EIO, Vidoni M (2024) Multi-step automated generation of parameter docstrings in python: An exploratory study. In: Proceedings of the 2024 IEEE/ACM 46th international conference on software engineering: companion proceedings, pp 356–357

Vidoni M (2021) Self-admitted technical debt in r packages: An exploratory study. In: International Conference on Mining Software Repositories, IEEE, pp 179–189

Wang Q, Li B, Xiao T, Zhu J, Li C, Wong DF, Chao LS (2019) Learning deep transformer models for machine translation. In: Proceedings of the 57th annual meeting of the association for computational linguistics, association for computational linguistics, Florence, Italy, pp 1810–1822. https://doi.org/10.18653/v1/P19-1176https://aclanthology.org/P19-1176/

Wattanakriengkrai S, Srisermphoak N, Sintoplertchaikul S, Choetkiertikul M, Ragkhitwetsagul C, Sunetnanta T, Hata H, Matsumoto K (2019) Automatic classifying self-admitted technical debt using n-gram idf. In: 2019 26th Asia-Pacific software engineering conference (APSEC), IEEE, pp 316–322

Wilcoxon F, Katti S, Wilcox RA et al (1970) Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank test. Sel Tables Math Stat 1:171–259

Williamson B (2024) The social life of ai in education. Int J Artif Intell Educ 34(1):97–104

Xiao T, Zeng Z, Wang D, McIntosh S (2024) Quantifying and characterizing clones of satd in build systems. Empir Softw Eng 29(2):1–31

Ximenes RG (2024) Issues that lead to code technical debt in machine learning systems. Master's dissertation, Pontifical Catholic University of Rio de Janeiro (PUC-Rio)

Yadav S, Shukla S (2016) Analysis of k-fold cross-validation over hold-out validation on colossal datasets for quality classification. In: 2016 IEEE 6th international conference on advanced computing (IACC), IEEE, pp 78–83

Yang Y, Li J, Yang Y (2015) The research of the fast svm classifier method. In: 2015 12th International computer conference on wavelet active media technology and information processing (ICCWAMTIP), IEEE, pp 121–124

Yu Z, Fahid FM, Tu H, Menzies T (2020) Identifying self-admitted technical debts with jitterbug: A two-step approach. IEEE Trans Software Eng 48(5):1676–1691

Yu J, Tian H, Li R, Zuo Q, Di Y (2024) Detecting self-admitted technical debts via prompt-based method in issue-tracking systems. Electronics 13(23):4700. https://doi.org/10.3390/electronics13234700

Yu J, Zhou X, Liu X, Liu J, Xie Z, Zhao K (2023) Detecting multi-type satd with gan-based neural networks. Inf Softw Technol 158

Zampetti F, Serebrenik A, Di Penta M (2020) Automatically learning patterns for self-admitted technical debt removal. 2020 IEEE 27th International Conference on Software Analysis. Evolution and Reengineering (SANER), IEEE, pp 355–366

Emmanuel Iko-Ojo Simon is a Ph.D. researcher at the School of Computing at the Australian National University (ANU). He holds a B.Sc. and an M.Sc. in Computer Science. His research focus is on Algorithm Debt in machine learning and deep learning systems, with the aim to characterise its causes, effects, and mitigation strategies for the long-term quality of machine learning software. He is also interested in the design and evaluation of AI software tools for public health, integrating machine learning models with software engineering practices.

Chirath Hettiarachchi is a Research Fellow at the School of Computing at The Australian National University (ANU), developing algorithms in Reinforcement. Learning-based closed-loop clinical treatment strategies for glucose regulation in Type 1 Diabetes and brain stimulation for depression. He completed his PhD in Computer Science at the ANU, MSc (Research), and BSc Honours (Electronics & Telecommunication Engineering) from the University of Moratuwa.

Alex Potanin completed his PhD in 2006 on Generic Ownership, demonstrating how type polymorphism can support ownership types—a concept later popularized by the Rust programming language. He subsequently explored the deep connections between ownership and immutability through Royal Society of New Zealand Marsden Funding (2008–2011), culminating in a comprehensive book chapter on the subject. Following a sabbatical at Carnegie Mellon University's Institute for Software Research with Professor Jonathan Aldrich, Alex created the Wyvern Programming Language; designed for security and usability, this project generated a decade of student research and publications, including innovations in type members that have influenced the modern Scala programming language. Currently, Alex is focused on designing modern module systems based on capabilities, exploring combinations of abstract and algebraic effects, and developing new paradigms for fully verified and secure software.

Professor Hanna Suominen MSc (Applied Mathematics), PhD (Computer Science), Adj/Prof (a.k.a. Docent, Computer Science), MEdL (Curriculum & Pedagogy), SFHEA, is a renowned catalyst in co-creating human-centered machine/deep learning, as evidenced by her TEDx talk; method, research, teaching, and business awards; and 20 years of fuelling the demand for smart sensing and sense-making systems. She has authored over 250 papers published in the most prestigious journals with over 250 co-authors from about 20 countries. Her papers have been cited approximately 4,500 times (h-index 31), which is impressive in her area of health informatics and natural language processing. After her pioneering contributions to clinical text mining, speech recognition in hospitals, and smart sports sensors in 2005-2015, she has discovered methods to detect Parkinsonian markers on human voice that are imperceptible to a neurologist; created smartphone apps to facilitate such discoveries; and supported customer-friendly information access through data and software, evaluated and reviewed systematically as rapidly growing and long-lasting in their research impact. In 2017-24, she conceptualised and co-chaired the inaugural strategic initiative (Our Health in Our Hands) at the Australian National University that brought together a diverse team of 100. According to its 5-year external review, her skilled leadership and mentoring had a positive impact on their capacity to work together collegially across disciplines, and both PhD candidates and post-doctoral researchers were overwhelmingly positive about the benefits of transdisciplinary working. She has also successfully spun out a company, called PostAc®, and co-produced research and education with healthcare consumers and companies. Therefore, her academic practice, with a proven discovery capability, research excellence, and advocacy for continuous improvement in researcher training, has completely reshaped engagement and impact within her scholarly community, also seeing students and staff thrive under her leadership.

Dr. Fatemeh Hendijani Fard is an Assistant Professor at The University of British Columbia, Canada, where she leads the Foundational AIware Research and Development (FARD) lab. Her research interests are at the intersection of Natural Language Processing and Software Engineering, focusing on different aspects of AI including AI Safety, reasoning and explainability, computational efficiency and knowledge transfer for low-resource and domain-specific domains, with limited available data. Few-shot learning, retrieval augmented generation and agent-based systems are at the heart of her research with directions for developing more precise and smaller models. She collaborates closely with industry, advises some AI startup companies in NLP area, and has served as a program committee member and reviewer in several journals and conferences, including TSE, TOSEM, EMSE, FSE, and ASE. Dr. Fard is an ACM and senior IEEE member. She gets back to the community by mentoring females interested in AI.

Authors and Affiliations

Emmanuel Iko-Ojo Simon[1] · Chirath Hettiarachchi[1] · Alex Potanin[1] · Hanna Suominen[1,2,3] · Fatemeh Fard[4]

✉ Emmanuel Iko-Ojo Simon
emmanuel.simon@anu.edu.au

[1] The Australian National University (ANU), ANU School of Computing, Canberra, Australia

[2] ANU School of Medicine and Psychology, Canberra, Australia

[3] University of Turku, Department of Computing, Turku, Finland

[4] University of British Columbia, Department of Computer Science, Kelowna, Canada