

Multiclass Classification and Prioritisation of Static Analysis Warnings Using Developer-Labelled Industrial Data*

BENEDIKT FEIN, University of Passau, Germany

VIBHASH KUMAR SINGH, University of Passau, Germany and CQSE GmbH, Germany

GORDON FRASER, University of Passau, Germany

Static analysis tools can identify code quality issues like vulnerabilities or performance problems. However, the high number of irrelevant warnings needs to be addressed by developing models that pre-filter and rank warnings before showing them to the developer. Existing research heuristically constructs model training data from unlabelled open-source data by assigning clearly separated binary warning labels like *fixed* and *irrelevant*. However, such datasets cannot capture distinctions between *relevant, but not yet fixed* and *not fixed, because irrelevant* warnings. Conversely, developers at CQSE enforce a development process for their *Teamscale* product where *all* warnings in their codebase are meticulously labelled as *accepted*, *tolerated*, or *false-positive*. This raises the question of how models trained on higher-quality data perform compared to those based on incomplete open-source data. Since warnings receive labels distinguishing between actual false positives and uncritical true positives, we adapt previously proposed models for a new multiclass classification task to evaluate their warning classification and prioritisation ability in this new setting. Our experiments show that the subtle difference between categories of unresolved warnings is more challenging for the models compared to a binary prediction. Nevertheless, training the models for the multiclass task results in a statistically significant improvement in warning prioritisation.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; *Maintaining software*; • **Computing methodologies** → **Neural networks**.

Additional Key Words and Phrases: static code analysis warning prioritisation, software quality

1 Introduction

Static code analysis tools are widely adopted as part of the software development process [42, 1, 7]. However, since static analysis warnings may be false positives or lack relevance [19], between 35 % and 91 % of all warnings are being ignored by developers [15, 14, 24]. Various machine learning techniques have therefore been proposed to selectively filter out non-actionable warnings [38, 25, 16, 20] or to prioritise the relevant warnings [32, 29], thus making it easier for developers to find the actually relevant information.

A *false positive* warning would be, for example, if a code analysis tool reports an unused variable which is in fact used. There are no productive changes a developer could implement on the code to address such a warning, whereas any *true positive* warning allows the developer to make improvements to the code. A central challenge for creating predictive models that can distinguish between such warnings lies in acquiring adequate training data that includes corresponding labels for static analysis warnings. Since these datasets are difficult to acquire, existing research heuristically constructs training data from unlabelled open-source data by assigning these binary labels to clearly identifiable cases. For example, if developers resolve static analysis warnings that are true positives they may include this information in corresponding commit messages. Labels for false positives, on the other hand, are more difficult to derive. One approach explored in research is to assume that everything that developers chose to ignore represents false positives [38], which is likely a gross overapproximation [20]. An alternative approach lies in considering only cases where

*Supersedes the earlier preprint version at <https://doi.org/10.21203/rs.3.rs-6469376/v1>.

developers used annotations in the code that inform static analysis tools to ignore certain warnings and treat these as false positives [16], which, however, leaves a large gray area of unresolved warnings with no information or labels.

Both of these approaches consequently ignore an important distinction: Warnings may in fact be true positives, but are *tolerated* by the developer since their impact is minimal or fixing them is infeasible. For example, a report ‘Calling *contains* on lists can be inefficient’ is valid, but can be ignored in cases where the developer knows that the list is always small enough to have negligible performance impact. Since these cases are neither real false positives nor desirable true positives, filtering or prioritisation approaches would need to take this into account in order to provide more value to developers. Similarly, static analysis warnings may be important true positives that developers intend to fix but have not yet come around to do. Unfortunately, existing approaches for generating training datasets would either teach models explicitly to treat such warnings as false positives, thus erroneously de-prioritising potentially important warnings, or they would simply have no information leading to unclear prediction results. In either case the result would be a suboptimal prediction, negatively affecting the effectiveness of either filtering or ranking.

Arguably, these challenges are a result of the heuristic approach to reverse-engineering labelled training data, since there typically is no other source of information for open-source projects. However, things may be different in industry, where professional software developers often have to adhere to rigorous rules and guidelines that are imposed in order to ensure consistent software quality. For example, the *CQSE* company enforces the strict rule in the development process of their *Teamscale* static analysis tool that *all* static analysis warnings reported for the 173 different warning types [18] have to be tagged as either *accepted*, *tolerated*, or *false-positive*. Thus, such an industrial dataset provides complete and clean data that for open-source projects could never be heuristically reverse engineered at the same level.

The data collected by the *Teamscale* team over the course of the development history of their software product provides a unique opportunity to re-visit the problem of predictive filtering and ranking of static analysis warnings. Various types of models to achieve these tasks have been proposed over time: Classic machine-learning models using hand-crafted features extracted from the warning and the surrounding code are still among the state-of-the-art approaches in this field [38, 20]. However, since the classification of static analysis warnings might be very specific to local code elements and the code context of the warning, more recently deep learning techniques have also been proposed. In particular, state-of-the-art models using neural networks with *word2vec*-embeddings [16] as well as Transformer models [23] have shown competitive performance. All prior evaluations of these models, however, were based on datasets constructed using heuristic approaches based on open-source software with its resulting narrow focus on binary classification trained on clear cases. This raises the question of how these models perform given a more complete dataset. Consequently, given the *Teamscale* dataset, the first research question we consider in this paper is:

RQ1 How do state-of-the-art models trained on fully labelled data perform at binary warning classification?

Given our exhaustive dataset, we are no longer restricted to casting the problem as a binary classification. Instead, we can treat it as a multi-class classification problem distinguishing between true positives, actual false positives, and true positives that are tolerated by developers. Consequently, the second research question addressed in this paper is:

RQ2 How do state-of-the-art models trained on fully labelled data perform at multi-class warning classification?

To save computing resources in practice, or to apply models to projects too small to be used for dedicated training, it may be interesting to pre-train the models on a larger project and then apply it to others. To enable this, the model must be capable to generalise sufficiently. We therefore need to study whether training on one project and cross-predicting static analysis warnings on another project performs sufficiently well. Our third research question therefore is:

RQ3 How well do models perform at cross-predicting static code analysis warnings on an unseen project?

While the difference between warnings that have been fixed (*accepted resolved*) and ones that are *false-positives* may have clear indicators, the difference between warnings where the developer has declared intent to fix them in the future (*accepted unresolved*) and ones that so far remained unresolved due to being *tolerated*, may be very nuanced. Prior datasets only included the former two categories, while a central novelty of our dataset is that it includes the latter labels. This raises the question of how the models perform specifically at classifying these unresolved warnings:

RQ4 What is the influence of unresolved warnings on the models' performance?

Throughout our evaluation we focus on recall as the main metric to minimise the number of *actionable* warnings being shown as *unactionable*. However, practically usable models should not frequently present *unactionable* as *actionable* either, as this would no longer achieve the main goal of reducing the number of irrelevant warnings a developer has to check. The fifth research question therefore is:

RQ5 What are common misclassifications of tolerated and false-positive warnings?

Finally, the multi-class classification provides more fine-grained information, which can be used to refine approaches not only to classify, but also to *rank* static analysis warnings. To assess whether this provides an improvement, the final research question of this paper is:

RQ6 Do developer intentions in addressing warnings align with the prioritisation assigned by the models?

In summary, the main contributions of this paper therefore are as follows:

- We introduce the idea of modelling static code analysis warning classification as a more fine-granular multiclass classification problem.
- We evaluate different models using the *Teamscale* dataset. This provides multiple advantages: (1) It allows for better comparison to prior work, in which different datasets were used in different studies; (2) it uses substantially more static analysis warning types than prior work [23, 25]; and (3) the warning labels represent the actual developer intent.
- We evaluate whether prior models can be improved using newer LLM-based code models.
- We evaluate whether the warning prioritisation of the model lines up with developer intent by comparing the ground-truth warning ranking to the ones proposed by the models.

Our experiments show that the developer-labelled multiclass classification is challenging for the models and thus does not lead to a clear improvement over binary-labelled data in the classification performance. Nevertheless, training the models for the multiclass task rather than the binary one results in a statistically significant improvement in the prioritisation of the warnings.

2 Background

2.1 Automated Static Analysis Tools

Static analysis tools like *CheckStyle*, *PMD*, *Sonarlint*, or *Spotbugs* allow developers to identify potential issues early during development by running them for every commit as part of a continuous integration system [42] and integrating them into the code review process [36, 1, 30, 35]. Studies

have confirmed that such tools identify real bugs [12, 34], leading to their adoption by large companies [33, 34, 7] and open-source projects [3] alike.

In this paper, we focus on the warnings reported in the *Teamscale* [18] static analysis tool. *Teamscale* not only generates warnings based on its own analysis, but also integrates other common analysers (e.g. *SonarLint*, *SpotBugs*) and reports their warnings together with its own in a common framework.¹ Rather than only reporting warnings, it can track them over time. This allows to distinguish between recently introduced and longstanding issues. This also allows tracking the commits that introduced and resolved the warning for each individual one. Additionally, *Teamscale* allows developers to manually label warnings into three categories: *Accepted* warnings represent valid issues that need resolution. Resolved warnings are also automatically put into this category. *Tolerated* warnings mark it as intentionally unsolved since fixing it might not be important in this specific case, e.g. a warning ‘Calling “contains” on lists is inefficient’ might be generated by *Teamscale* in case the data-flow analysis cannot determine the size, but might not have a relevant impact in a specific instance since the developer knows the list can only be small. Finally, warnings marked as *false-positive* indicate that the static code analysis incorrectly marked a snippet as problematic.

2.2 Classification of Linter Warnings

Not all warnings produced by static code analysis tools need to be addressed. Since there are potentially many warnings, a classification in actionable and irrelevant warnings should help developers focus on the more important issues. Previous research proposes different machine learning models for this classification task. Initial studies use a set of hand-crafted features extracted from the source code snippet to perform this classification [38, 32]. These features cover different aspects surrounding the warning like for example the file type or age of the file the warning appeared in, the cyclomatic complexity of the surrounding code, or historic data such as the overall proportion of previously resolved warnings of this type. The features are then used to train classical machine learning models like for example Random Forest classifiers [38].

Since designing the features requires significant engineering effort, later evaluations applied deep learning techniques originally used for natural language processing (e.g. long short-term memory, *word2vec* [28]) that learn to embed the source code context into vector space. The source code surrounding the warning location is split up into tokens which are used as ‘words’ of the ‘sentences’ used to train the embedding models. To represent the whole code snippet, the individual token embeddings are aggregated into a single vector. Instead of crafted features, this vector is then used as feature vector in the input of either classical machine learning classifiers (e.g. Support Vector Machine, Random Forest), or deep-learning networks [25, 29, 16].

Finally, to abstract from the context-specific code embeddings, the use of pre-trained Transformer-based language models has been proposed [23]. Since the Transformer model already has sufficient understanding of source code in general, the model weights can be mostly frozen during the fine-tuning for the specific task so that only few model weights that are part of the classifier layers need to be trained.

Since all these approaches use own datasets based on different static code analysis tools and different labelling heuristics, it is not possible to directly compare their model performance. Therefore, there is no clear best approach and all models can be considered to be relevant state-of-the art. However, all these approaches model the classification task as a binary problem: warnings are either actionable (also called true-positives) or unactionable (false-positives). The developer annotations

¹<https://docs.teamscale.com/reference/supported-technologies/static-analysis/>, accessed 2025-06-23

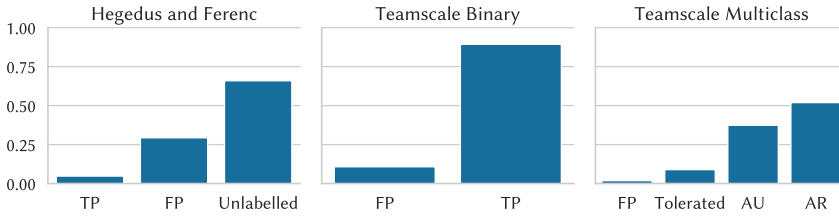


Fig. 1. Label distribution in the Hegedus and Ferenc dataset [17] differentiating between *true positive* (TP) and *false positive* (FP) warnings, and in our *Teamscale* dataset also considering *tolerated*, *accepted unresolved* (AU), and *accepted resolved* (AR) warnings.

for the *Teamscale*-generated warnings allow for a more fine-grained multiclass classification into three categories that aligns more closely with how developers interact with the warnings.

2.3 Training Data for Static Analysis Warning Classification

A common labelling approach categorises warnings into the two groups *actionable* and *unactionable*. For example, by following the approach of prior literature [15, 26, 13], Wang et al. [38] generated a binary-labelled dataset of warnings generated by the *FindBugs* tool. This dataset has also been used in later research [41, 20]. The dataset is created by running the code analysis tool on a range of historic commits from twelve different open-source projects. By comparing the set of warnings between two such states, the warnings can be labelled: *actionable* warnings were only present in the older project state but have been fixed and no longer appear in the new state while *unactionable* warnings appear in both the old and new states of the project.

This approach relies on an implicit assumption that the warnings were fixed intentionally by some change between the two compared states. However, some warnings may also be resolved automatically due to unrelated functional changes to the relevant code locations. Other warnings may be *actionable*, but have not yet fixed in the timeframe between the two commits (3–6 months [38]) due to other development priorities and are therefore misclassified as *unactionable* [20]. To solve these issues, another approach by Hegedus and Ferenc [16] collects explicitly fixed (*true positive*) and explicitly ignored (*false positive*) warnings generated by the *SonarQube*/*SonarLint* tool by mining open-source projects from *GitHub*. Explicitly ignored warnings are collected by searching for special `//NOSONAR` code comments added by developers after a useless warning appeared. This comment is recognised by the code analysis tool and instructs it to no longer generate a warning for the location. *True positive* warnings are collected by mining and inspecting commit messages that explicitly reference specific warnings (i.e. they contain the keyword `squid:` used by *SonarQube* to mark warning *IDs*). Running the code analysis on the identified commit and on the previous commit then shows whether a warning was indeed fixed by the commit.

While this approach ensures an accurate labelling of the considered warning instances, it misses many cases where these heuristics do not apply. For example, any unresolved warning still reported by the tool for the project remain unlabelled. To estimate the proportion of such unlabelled warnings, we cloned 9 631 (of 11 248) still available repositories of the dataset by Hegedus and Ferenc [16, 17]. For each repository, we checked out the latest commit mentioned in the original dataset and executed *SonarLint* on it, which succeeded for 8 567 projects. Since the newest referenced commit is the one containing all fixed or ignored warnings considered in the original dataset for this repository, all remaining warnings reported by the analysis tool are *unlabelled*. As Fig. 1 shows, these warnings make up the majority (65.9 %) of all warnings. This clearly highlights that while

this approach solves the issue of inaccurately labelled warning instances, it results in incomplete label coverage.

3 Experimental Setup

The static analysis warnings can be classified either into binary classes (actionable or unactionable) like in previous research, or by using *Teamscale* features into the three groups ‘actionable’, ‘tolerated’, and ‘false-positive’. We therefore aim to investigate both the binary and the multiclass classification case. We mainly evaluate the models for warning classification since an accurate classifier model can also directly be used for prioritisation. The models classify the warning by assigning a value between 0 and 1 for each of the classes denoting the likelihood of the warning belonging to this class. This value therefore represents the confidence of the model in the prediction. The priority of a warning is then established by ordering the warnings classified as actionable by their descending confidence, i.e. the ones that appear first in the list should be fixed first according to the model’s suggestion.

3.1 Dataset

Our dataset was obtained from 1 200 warning-containing commits made to the default branch of the *Teamscale* codebase itself at CQSE between April 2022 and October 2024. Using *Teamscale* to analyse each commit resulted in a total of 13 419 warnings in 173 unique warning types. Each warning is stored in a database and contains a reference to the commit it was introduced in. The warning remains visible until the relevant code location has changed enough for it to be fixed or no longer be relevant. By checking the additions to and removals from the database after each commit, we can track for each warning at each point in time if it has been resolved in this commit or if it remains unresolved. All the 13 419 warnings for the *Teamscale* project were grouped by developers into the three categories *accepted*, *tolerated*, and *false-positive* (cf. Section 2.1). This labelling resulted 11 979 accepted (6 963 of them resolved, 5 016 unresolved), 1 194 tolerated, and 246 false-positive warnings (cf. Fig. 1) in our dataset. All unresolved warnings that are not explicitly marked as *tolerated* are classified as potentially relevant and therefore categorised as *accepted*. To allow for a binary classification of warnings like in prior work [20, 16], the *accepted* warnings can be seen as *actionable*, while *tolerated* and *false-positives* are unactionable.

To allow for evaluation of the model across different software projects, we also consider the Java coverage collection tool integrated into *Teamscale*—the *Teamscale Jacoco Agent*²—which is developed as a separate project, but is labelled in a similarly meticulous way as *Teamscale*. In the same timeframe as for the main project, the dataset for this project contains 101 relevant commits containing a total of 504 warnings. Of those, 372 are marked as *accepted* (269 resolved, 103 unresolved), 124 as *tolerated*, and 8 as *false-positive*. Since this dataset is too small to train a project-specific model on it, we use it purely for evaluation if our models generalise to unseen projects.

3.2 Models

3.2.1 Feature-based Classification Model. We chose the classification models *Decision Tree* and *Random Forest* from *scikit-learn*,³ *XGBoost*⁴ [6], and *Balanced Random Forest*⁵ [5] for this experiment. Since typical feature-based models use some features that are based on historical data, like for example the proportion of warnings that have so far been marked as *accepted* per warning type [38],

²<https://github.com/cqse/teamscale-jacoco-agent>, accessed 2025-06-17

³<https://scikit-learn.org/>, accessed 2025-06-17

⁴<https://github.com/dmlc/xgboost>, accessed 2025-06-17

⁵<https://imbalanced-learn.org/>, accessed 2025-06-17

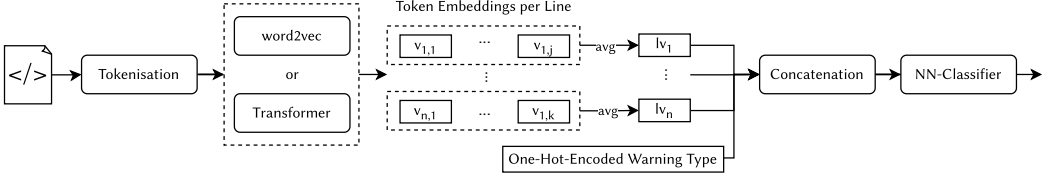


Fig. 2. Model architecture: The tokenised source code is passed into either *word2vec* or a Transformer to obtain the token embeddings $v_{i,l}$. They are averaged into line embeddings lv_i . A concatenated vector together with a one-hot encoding of the *Teamscale* warning type is fed into the neural network classifier.

a potential data leakage could occur for features that make use of historical data when randomly sampling datapoints for training and evaluation [20]. For example, in case there is a warning in the test data, but the training data contains newer commits for the same file that introduced additional warnings, the *warning density* for this file misrepresents the knowledge the model could have about unseen data. As mitigation, we sort the commits by ascending timestamp and use the first 70 % (23 months worth of data) of them as our training dataset, the next 15 % (6 months) as validation data during model development, and the remaining 15 % (5 months) as evaluation data.

Data Preprocessing. As shown in Section 3.1, our dataset exhibits a significant class imbalance regarding accepted and false-positive warnings. To prevent bias towards the majority class, we applied a combination of various techniques. By selecting a random subset of accepted warnings, i.e., applying random undersampling, we reduce the overall proportion of the dataset in this category. We apply the *SMOTE* [4] minority over-sampling technique to generate additional synthetic samples for the underrepresented classes by randomly sampling from the minority class, identifying the *k*-nearest-neighbours, and then interpolating between them. Applying *SMOTE* together with undersampling has been shown to improve classifier performance [31]. Finally, we apply *TOMEK* [2] to identify and remove borderline instances that are close to each other but reside in different classes. Such samples are likely to cause confusion for the classifier. Again, the combination of *SMOTE* with *TOMEK* has been shown to be beneficial [43].

The *Balanced Random Forest* can handle the imbalanced data internally without additional processing. For the other models in the binary classification case, we applied *SMOTE* and random undersampling to adjust the data to a 2:4 ratio. In case of the multiclass classification, we targeted a 4:8:8 ratio with the smallest proportion representing the false-positive warnings. The combination of *SMOTE* and *TOMEK* performed best for *Decision Tree* and *Random Forest*, while for *XGBoost* using *SMOTE* and random undersampling was the best choice.

Input Features. Initially, we collected 40 features inspired by the study of Wang et al. [38] plus additional ones based on the metrics provided by *Teamscale* (e.g. test coverage for the file). Using recursive feature elimination [11] to iteratively remove the least important features for the classification, we only used 14 of them in the final models for binary and 12 for multiclass classification (cf. Table 1).

3.2.2 word2vec-based Classification Model. Figure 2 shows our model architecture, which follows the architecture proposed in prior research [16] with minor changes to the *word2vec* [28] hyperparameters found by random search hyperparameter-tuning to adapt to the different dataset. The tokenisation step splits the source code into a list of tokens. To limit the vocabulary size for the *word2vec* model, we replace all identifiers (e.g. variable names) with the generic IDENTIFIER token. To train the *word2vec* model, the individual lines of the source code are used as ‘sentences’. *word2vec*

Table 1. Features used for the feature-based *binary* and *multiclass* classifier models. Features marked with (W) were proposed in the golden feature set by Wang et al. [38].

Feature	Category	Description	Model	
			b	m
number of added files	commit	New files added in the commit that introduced the warning.	✓	
number of added warnings	commit	Number of new warnings introduced in the commit.	✓	
number of modified files	commit	Number of changed files in the commit.	✓	✓
comment code ratio (W)	file	Number of commented lines divided by total source lines in the file.	✓	✓
file size	file	File size assessment by <i>Teamscale</i> : Green, Yellow, Red.		✓
number of attributes	file	Number of attributes of the class declared in the file.	✓	
number of comments (W)	file	Total number of comments in the file.		✓
number of methods (W)	file	Number of methods in the file.	✓	
SLOC	file	Source lines of code.		✓
test coverage	file	Proportion of lines covered by tests.	✓	✓
warning density (W)	file	Number of warnings per 1 000 lines of code.	✓	✓
defect likelihood (W)	historic	Proportion of unactionable warnings of the same warning type.	✓	✓
defect likelihood variance (W)	historic	Variance of the defect likelihood of the warning type within the warning group.	✓	✓
false positive likelihood	historic	Proportion of false-positive warnings of the same warning type.		✓
number of edits	historic	Number of commits modifying the file in which the warning occurred.	✓	
toleration likelihood	historic	Proportion of tolerated warnings of the same warning type.		✓
warning context warning type (W)	historic	Normalised difference of actionable and unactionable warnings for the warning type.	✓	✓
warning group	warning	Warning group of the warning type.	✓	
warning type (W)	warning	The concrete warning type out of the 173 types.	✓	

in skip-gram configuration with a window size of 3 and embedding size 128 yielded the best results for binary classification, while a window size of 1 with the same embedding size performed best in the multiclass classification.

To generate the inputs for the classifier, we limit the code context to three lines above and below the warning location. Then, the *word2vec* model generates the individual token embeddings. We average these embeddings per line to obtain a list of line embeddings. To include warning

information, we also generate a one-hot-encoded vector of the warning type (173 classes). The input to the neural network classifier is a concatenation of the seven line embeddings and this warning type encoding. The neural network (NN) classifier has ReLU-activated hidden linear layers of variable sizes with dropout layers ($p = 0.2$) in between. For binary classification, we add a final linear sigmoid-activated layer and use binary cross-entropy loss during training. The multiclass classifier uses the softmax-activation and categorical cross-entropy loss instead. Both variants are trained using the Adam optimizer with a learning rate of 1×10^{-4} [16]. To account for the imbalanced dataset and avoid bias towards the majority class, we compute class weights and apply them during the training process. The layer sizes of the NN were determined by grid search for layer counts between one and four, and individual layer sizes in [100, 200, 400, ..., 6400].

Since this model does not use historical features, random sampling to split into training and evaluation data is now possible. However, like for the feature-based model, we use the newest 15 % commits as evaluation data ensure comparability of results. The remaining 85 % of the data are used during training in a stratified 5-fold cross-validation. As best overall model configuration we selected the one that had the highest mean recall across the five folds.

By stacking the token or line embedding vectors to arrange them into a matrix, a convolutional neural network could be used instead of the NN-classifier [25]. Since this approach resulted in considerably worse metrics (−8 %-point average recall) even after hyperparameter-tuning in our preliminary experiments, we do not consider it further in the following.

3.2.3 Transformer-based Classification Model. In the model architecture (see Fig. 2), the *word2vec* model maps the source code tokens into embeddings. A Transformer model receives the same type of inputs and outputs and can therefore act as a direct replacement within our architecture. Replacing all identifiers with a fixed token in the tokenisation step is no longer necessary, since the Transformer can handle larger vocabularies. Therefore, we can omit specific preprocessing and treat the full source code extract like regular text when passing it to the model-specific tokeniser.

Due to their size, the Transformer models require a large corpus of training data to model the programming language syntax. Since our model architecture only uses the Transformer to generate the token embeddings without training it further, it does not require specific knowledge about the warning context. We therefore employ models that have been trained on large corpora of source code. The models used for our experiments are *CodeBERT* [8], *GraphCodeBERT* [10], *UniXcoder* [9], *CodeT5* [40], and *CodeT5+* [39].⁶ The *BERT* models represent encoder-only Transformers while the other three use an encoder-decoder architecture. With approximately 125×10^6 parameters for (*Graph*)*CodeBERT* and *UniXcoder*, and 223×10^6 parameters for the *T5* variants, these models are comparatively small compared to other state-of-the-art code models (e.g. *StarCoder 2* [27] with $3\text{--}15 \times 10^9$ or *Devstral*⁷ with 24×10^9 parameters). Since our code snippets are relatively short and only consist of a single programming language, we do not require the larger model’s ability to generalise in such scenarios, but instead benefit from the reduced computational effort.

All Transformer-generated token embeddings have a fixed vector length of 768. To adapt the remainder of our model architecture (see Fig. 2) to the new embedding generation model, the size of $v_{i,l}$, lv_i is changed accordingly. During hyperparameter-tuning, we changed the layer sizes of the classifier with the same range for layer count and sizes as for the *word2vec*-based model and allowed $n \in \{2, \dots, 7\}$ lines of code context surrounding the issue. Otherwise, approach and training parameters (cf. Section 3.2.2) remain unchanged.

⁶Model implementations available on [HuggingFace](https://huggingface.co/microsoft/codebert-base) as [microsoft/codebert-base](https://huggingface.co/microsoft/codebert-base), [microsoft/graphcodebert-base](https://huggingface.co/microsoft/graphcodebert-base), [microsoft/unixcoder-base](https://huggingface.co/microsoft/unixcoder-base), [Salesforce/codet5-base](https://huggingface.co/Salesforce/codet5-base), and [Salesforce/codet5p-220m](https://huggingface.co/Salesforce/codet5p-220m).

⁷<https://mistral.ai/news/devstral>, accessed 2025-06-17

Table 2. Best model configurations on the binary classification dataset by Hegedus and Ferenc [16]. All models use $n = 2$ lines of code context surrounding the issue location.

Embedding Model	NN Configuration	Acc.	F1	Prec.	Recall
<i>word2vec</i> (original)	800, 800, 800	0.902	0.803	0.831	0.778
<i>word2vec</i>	400, 400, 400	0.972	0.958	0.947	0.971
<i>CodeBERT</i>	2400, 1200, 800	0.998	0.997	0.997	0.998
<i>GraphCodeBERT</i>	200, 100	0.998	0.997	0.996	0.998
<i>UniXcoder</i>	800, 800, 800	0.989	0.984	0.981	0.987
<i>CodeT5</i>	400, 400, 400	0.998	0.997	0.996	0.998
<i>CodeT5+</i>	2400, 2400, 2400	0.999	0.998	0.998	0.999

3.3 Open-Source Baseline

Ideally, we would like to compare the performance of our models on open-source data, in particular since our dataset is smaller than datasets used in prior work. We considered the publicly available replication package by Hegedus and Ferenc containing 224 484 binary labelled samples [16, 17]. Following the original experiment, we randomly selected 10 % of the dataset using stratified sampling based on the warning type for evaluation and used five-fold cross-validation on the remaining data during training.

As Table 2 shows, the Transformer-based model achieved a dubious recall of 0.998 which is far above the originally reported one of 0.778. This performance is unlikely unless a data leak contaminated the training data. We also applied our reimplementations of the *word2vec*+NN approach (see Section 4.2) to this dataset. It also exhibits a vastly superior performance (recall 0.971) compared to the original. However, it is no longer a near-perfect prediction, thus making a direct data leak unlikely. Instead, we assume our reimplementations are either missing an important data processing step that was not described in the original paper, or differs somewhere in the actual implementation. To train this model, we transformed the data from the original format into the one we used for our dataset and then applied the same model training and evaluation script as used for our experiments. Therefore, an implementation error in our scripts is unlikely, since we would have experienced a similarly high performance in the evaluation on our own dataset. Regardless, this makes our results incomparable to the original ones.

In an attempt to further eliminate a data leak as possible explanation for the results, we evaluated this exceptionally performing Transformer model on the 850 manually labelled datapoints in the replication package of [20, 21]. However, the model does not produce useful predictions with an F1-score of only 0.47 and an accuracy of 0.77 that matches the proportion of actionable warnings in this dataset (660/850), i.e. a ‘dumb’ baseline model that ignores the input and always predicts *actionable* would perform equally well. It is possible that the model primarily learned to predict the class based on the one-hot-encoded warning type. In this case such a result would be expected, since this vector is always zero due to the different warning types when applied to this test dataset. To analyse if this issue of the transition between domains can be overcome, future research could evaluate whether the combination of the code embedding vector combined with hand-crafted features still exhibits the same behaviour.

In summary, based on the evaluation on those replication packages, we cannot determine whether the Transformer model would perform better on a larger dataset. Most likely this is neither due to a faulty implementation of our models, nor due to our models failing to generalise to new data, but is caused by some non-replicable data processing steps for the larger binary-labelled dataset.

3.4 Experiment Procedure

RQ1 and RQ2: Binary and Multiclass Classification. As main evaluation metric for RQ1 and RQ2 we use the average, non-class-weighted recall (i.e. the macro recall). Compared to the weighted variant, this gives a bigger weight to the underrepresented classes to ensure the models perform similarly well for all classes rather than only for the majority class. Like prior work [23], we focus on recall as main evaluation metric to minimise the number of false-negative predictions, i.e. we want to avoid classifying actionable warnings as unactionable. Additionally, we report the model performance according to the accuracy, F1, and precision metrics commonly used for machine learning experiments.

RQ3: Cross-Project Prediction. To answer in RQ3 how well the models generalise to unseen projects, we use the best-performing variants of models determined in RQ1 and RQ2 for the binary and multiclass tasks trained on the *Teamscale* dataset. We evaluate the performance of these models on the smaller *Teamscale Jacoco Agent* dataset without further fine-tuning.

RQ4: Influence of Unresolved Warnings. RQs 1–3 evaluated the model performance according to overall metrics like recall. However, these overall metrics do not indicate the model performance in distinguishing the nuanced differences between *accepted unresolved*, *accepted resolved*, and *tolerated* warnings. In RQ4 we therefore again use the best-performing model variants of RQ2 and investigate the kinds of misclassifications these models make for *accepted unresolved* and *accepted resolved* warnings. Since this distinction is only possible in the multiclass case, we do not consider the binary classification for this research question.

RQ5: Misclassifications of Tolerated and False-Positive Warnings. Similarly, in RQ5 we investigate the misclassifications of *tolerated* and *false-positive* warnings to investigate how often models misclassify *unactionable* warnings as *actionable*. We consider both the multiclass case by using the labels as-is and the binary case by considering *tolerated* and *false-positive* warnings to be *unactionable* and the other labels as *actionable*.

RQ6: Prioritisation Alignment with Developer Intentions. To answer RQ6 we need to produce a ranking based on the predictions. The models classify the warning by assigning a probability for each of the classes denoting the likelihood of the warning belonging to this class. This value therefore represents the confidence of the model in the prediction. The priority is then established by ordering the warnings classified as actionable by their descending confidence, i.e. the ones that appear first in the list should be fixed first according to the model’s suggestion.

In general, we assume that *accepted* warnings that have been resolved already are the most important ones with *accepted* but unresolved warnings following next. Then *tolerated* warnings could be relevant next, and the clearly irrelevant *false-positive* warnings should be ranked last. We rank all 2013 warnings from the test set of the *Teamscale* dataset. For the *Jacoco Agent*, all 504 data points are used for evaluation. We grouped these ground-truth data points accordingly. Since this does not provide a ranking within the group, we introduce as second ordering criterion the warning time-to-live (TTL, i.e. the time between it being introduced and being fixed) to rank warnings that have been fixed quickly higher, and use the randomly generated warning identifier (a UUID) as tiebreaker.

To deduce the ranking from the binary model prediction, we rank *actionable* warnings before *unactionable* ones. The multiclass model does not discern between resolved and unresolved labels, but the *accepted*, *tolerated*, *false-positive* order can be applied. For both variants, the ranking within groups is given by the confidence.

To evaluate whether the rankings proposed by the models match the ground-truth ones, we calculate the Kendall rank correlation coefficient [22] between the two lists. For a perfect model, the coefficient τ would be 1. To compare between the rankings of the binary and multiclass variants of the respective models, we also apply the Kendall test to these two generated rankings.

3.5 Threats to Validity

Threats to internal validity. A threat to internal validity arises from the significant data imbalance within the training dataset, leading to poorer model performance in those areas. While we apply multiple techniques to mitigate this imbalance, some of it remains in the final dataset. Furthermore, the different members of the *Teamscale* development team may have subjectively labelled similar warnings as *tolerated* or *false-positive* depending on their experience and individual judgement. Such inconsistencies can affect the model’s ability to generalise. Since warnings are shown to developer and reviewers for each change request, differences in judgement can be discussed there to agree on a label and therefore achieve a more consistent labelling. A further possible source of inconsistencies emerges due to the active *Teamscale* development. Especially the structure of *false-positive* warnings may change over time due to improvements in *Teamscale* that fix commonly appearing false-positive-patterns.

For RQ6 some additional threats to validity arise. The construction of the ground-truth ranking poses a threat. The original labels only allow for the grouping into four categories. They are developer labelled and therefore directly represent the developer intent. However, we cannot gather such information within the groups. We therefore use the warning time-to-live (i.e. the time between the commit that introduced it and the one where it was resolved) as an approximation of the intent under the assumption that warnings which were quickly fixed are more likely to be important than ones that have remained unresolved for longer. Not only important, but also easily fixable warnings might be resolved quickly. Our ground-truth construction of ranking them higher also makes sense in this case, since recommending fixes for these warnings represents code improvements that can be implemented with little effort.

Threats to external validity. The external validity is primarily threatened by the specific characteristics of the dataset. The findings for this industrial project may not generalise to other datasets. With *Teamscale*, we also use a different static code analysis tool compared to prior research. To allow for a comparison, we also evaluate a reduction of the classification problem to a binary one like for the detection of false-positive (i.e. unactionable) warnings.

Threats to construct validity. A threat to construct validity arises from the assumption that unresolved warnings are placed in the *accepted* category, i.e. they are actionable but have not yet been worked on by developers. Some of them might be *tolerated* upon further inspection. However, the project guidelines recommend to proactively mark *tolerated* and *false-positive* warnings as such. From our experience, this suggestion is followed by the developers, thus reducing the number of the unresolved warnings that fall into either of those two categories. A further threat to construct validity arises from the choice of models. As mitigation, we adapted existing architectures that have successfully been applied to the binary classification task in prior research and evaluate a wide range of different model designs.

4 Results

4.1 RQ1: Binary Classification

In this research question we compare the performance of the feature-based, the *word2vec*-based, and the Transformer-based classification models on the binary classification task as baseline. To

Table 3. Best model configurations for the binary classification task. n denotes the number of code context lines used as model input. The classifier neural network (NN) is defined by the size of its layers.

Model	n	NN Configuration	Acc.	F1	Prec.	Recall
<i>Decision Tree</i>			0.368	0.343	0.547	0.611
<i>XGBoost</i>			0.757	0.556	0.556	0.616
<i>Random Forest</i>			0.653	0.503	0.539	0.603
<i>Balanced Random Forest</i>			0.375	0.350	0.555	0.631
<i>word2vec</i>	2	2400, 2400, 2400	0.725	0.547	0.557	0.632
<i>CodeBERT</i>	4	800, 800, 800	0.637	0.498	0.541	0.613
<i>GraphCodeBERT</i>	3	6400, 3200, 1600, 800	0.61	0.488	0.544	0.623
<i>UniXcoder</i>	2	6400, 3200, 1600	0.641	0.502	0.544	0.620
<i>CodeT5</i>	3	800, 400, 200	0.490	0.420	0.536	0.602
<i>CodeT5+</i>	2	200, 200	0.736	0.564	0.569	0.661

establish this baseline for binary classification on our dataset, we first reduce the dataset labels to *actionable* (accepted warnings) and *unactionable* (tolerated and false-positive warnings).

4.1.1 Feature-based Models. Our first approach uses a classification model similar to prior work [38, 20] using hand-crafted features extracted from the warning and the surrounding code, and serves as a baseline the neural-network-based models can be compared against. As Table 3 shows, none of the feature-based models clearly outperforms the other ones. While *XGBoost* is clearly better considering accuracy (0.757) and F1-score (0.556), its recall falls behind the best model. The high accuracy can be attributed to its tendency to classify most warnings into the majority class. The *Balanced Random Forest* on the other hand has a precision matching *XGBoost*, but can achieve the best recall (0.631).

After feature reduction, the most relevant features are the warning type, the defect likelihood (i.e. the proportion of warnings of this type that are actionable in the historic data), and the total number of new warnings for the commit. As shown in Table 1, the other relevant features cover a wide range, ranging from commit-related features (e.g. number of added/modified files), over ones relating to the file the warning was generated for (e.g. comment to code ratio, number of methods), to ones specific to the warning instance (warning type and group). The file test coverage being a relevant feature might indicate that thoroughly tested code is less likely to contain warnings.

While Kang et al. performed their experiments on a different dataset containing warnings generated by a different static code analysis tool [20], we achieve similar metrics with our model (F1: 0.38 vs. 0.350, recall: 0.57 vs. 0.631), thus highlighting that this kind of model is adaptable to different scenarios by choosing a suitable feature set. For our model, seven of the 14 used features—including the two most important features for our model—stem from the ‘golden feature set’ used for prior evaluations [38, 20]. Our evaluation therefore confirms that historic data like the defect likelihood are important features [38], even though a careful implementation is required to avoid a potential data leak [20].

4.1.2 word2vec-based Models. The feature-based model captures broad information about the code. However, the generated warnings might be very specific to local code elements, e.g. modifying a list while iterating over it. We therefore evaluate *word2vec* embedding models that capture information about individual tokens in the code instead, which in the field of ranking static analysis warnings is still considered a state-of-the-art approach. Table 3 shows that the *word2vec* model recall (0.632) is

on-par with the feature-based model (0.631). The higher accuracy (0.725) compared to the feature-based model (0.375) combined with the similar recall indicates that the *word2vec* model tends to classify more frequently towards the majority class.

In practice, often small details in the code structure can decide between a warning being valid or not (e.g. the use of `<` over `<=` can decide if there is an off-by-one error). The tendency of *word2vec* to classify towards *actionable/accepted* may indicate that it cannot always capture such details. Therefore, even though the recall is similar to the feature-based model, this approach may present more potentially *actionable* warnings to the developer.

Our hyperparameter-tuning resulted in a NN that is considerably larger (layer sizes 2 400) than a comparable model established in prior research (layer sizes 800) [16]. Considering the smaller dataset compared to the previous research, we would have expected a small model to perform best. Otherwise, there might not be enough information contained in the training data to sufficiently train all model parameters. We observed however, that the large binary model could still be consistently trained to a similar performance level in all five training folds (recall standard deviation 0.017 8), thus indicating that no underfitting due to too few training samples occurs.

4.1.3 Transformer-based Models. Since the proposal of *word2vec* [28], more powerful language models based on the Transformer architecture [37] have emerged. We therefore want to evaluate if such models can improve the warning classification model. Table 3 shows the best configuration for the Transformer embeddings. The *CodeT5+* model emerges as the best one with a recall of 0.661, thus outperforming all others on all considered metrics.

As Table 3 shows, a wide range of NN configurations can achieve similar results depending on the Transformer embedding model. However, for all models, except for *CodeBERT*, a small code context size of two or three lines yields the best results. Additionally, the best performing model configurations use very small NN layers in sizes 200 or lower. When not looking at the best configuration per model type but the five best overall model configurations found during hyperparameter-tuning according to the mean recall over the five training folds, four of them are variants of a *CodeT5* embedding combined with layer sizes of 800 or less. Only the *GraphCodeBERT* variant uses larger layers (cf. Table 3).

This might indicate that the 9 122 warnings used during training do not provide enough information to fully train a larger NN-classifier with many parameters, which however would be required to achieve good recall across all training folds without depending on a good random parameter initialisation. This is supported by the differences in recall scores between folds: The *CodeT5+*-based model has the lowest standard deviation of $\sigma = 0.029$, whereas it ranges between 0.043 and 0.069 for the other models. Decreasing the layer size reduces the number of trainable parameters the available information has to be distributed across. However, due to the fixed embedding size of 768, the input size to the classifier has a minimum size of 4 013 ($768 \cdot 5$ lines of code context plus 173 for the one-hot-encoded warning type). Therefore, the NN-classifier always requires a trainable matrix of size $4\,013 \times k$ when having a first layer with output size k . This means that for the best models in Table 3 already $4\,013 \cdot 200 = 802\,600$ parameters are required just to ‘compress’ the information from the embeddings into the size required for the following layers.

4.1.4 Model Comparison. Overall, the best model according to the recall metric is the NN-classifier using the *CodeT5+* embeddings. The recall score of 0.661 is only slightly higher than the scores of 0.632 and 0.631 achieved by the *word2vec*+NN-classifier and *Balanced Random Forest* models, respectively. Also considering the other metrics, the Transformer-based model performs only slightly better than the *word2vec*-based one. However, there is a clear improvement over the feature-based model for both. Therefore, the embedding-based models should be preferred overall over the feature-based ones even though the difference in recall is small.

Table 4. Best model configurations for the multiclass classification task. n denotes the number of code context lines used as model input. The classifier neural network (NN) is defined by the size of its layers.

Model	n	NN Configuration	Acc.	F1	Prec.	Recall
<i>Decision Tree</i>			0.854	0.428	0.411	0.469
<i>XGBoost</i>			0.559	0.372	0.403	0.558
<i>Random Forest</i>			0.847	0.487	0.464	0.531
<i>Balanced Random Forest</i>			0.454	0.339	0.407	0.643
<i>word2vec</i>	2	200, 100	0.681	0.409	0.401	0.626
<i>CodeBERT</i>	4	800, 800, 800	0.475	0.304	0.374	0.499
<i>GraphCodeBERT</i>	2	2400, 2400, 2400	0.755	0.405	0.393	0.482
<i>UniXcoder</i>	2	200, 200, 200	0.783	0.438	0.435	0.463
<i>CodeT5</i>	2	6400, 3200, 1600	0.497	0.315	0.373	0.563
<i>CodeT5+</i>	2	200, 100	0.698	0.422	0.411	0.573

Our results indicate that while the crafted features can capture relevant information, the code context of the warning location is important for the classification. At the same time, the assumed additional expressiveness of the embedding of the pre-trained Transformers does not have a big impact. The embedding size of the Transformers is 768, while for *word2vec* the vector has size 128. Due to the comparatively small dataset, it is possible that the required larger NN-classifier for the Transformer-based model cannot be trained as well as the smaller *word2vec*-based one. Therefore, not all the potentially more nuanced features of the large vector can be extracted fully in the NN. Instead, the classifier might focus only on more important features also contained in the smaller *word2vec*-embedding. An alternative explanation for the similar performance of the embedding-based models might arise from the construction of the embeddings: Since the *word2vec*-model has been trained specifically on the project, it might capture project-specific patterns as part of the embedding, while the Transformer has no such information but can still achieve a similar performance by instead recalling more general code patterns learned in the pre-training steps.

Summary RQ1: All models classify warnings with similar performance considering recall. However, code-embedding-based models achieve a more balanced prediction performance compared to the feature-based models when taking other metrics into account. This highlights the importance of the code context for the classification.

4.2 RQ2: Multiclass Classification

Using the binary classification in RQ1 as baseline, we now evaluate the models on the multiclass classification task. Establishing such a baseline is especially important for the multiclass classification since no prior work explored this task. Therefore, we apply a multiclass classifier on the original labels *accepted*, *tolerated*, and *false-positive* of our dataset.

4.2.1 Feature-based Models. In Section 4.1.1 we established a feature-based model on the binary classification task as a baseline. As Table 4 shows, also in the multiclass classification task, none of the models clearly outperform the other ones. The *Random Forest* outperforms the other models on F1 and precision, while the *Decision Tree* model achieves the best accuracy. The *Balanced Random Forest* can achieve a better result only for the recall metric (0.643) with a clear margin over the next-best model (*XGBoost*, 0.558). This suggests that the *Balanced Random Forest* is more effective in identifying relevant instances across all classes. In contrast, the considerably lower recall for all

other models indicates a more conservative approach that results in missed relevant instances. As in the binary case, the models that achieve a high accuracy do not perform best overall since they may be overeager to classify instances into the majority class (*accepted*).

The *Balanced Random Forest* is the only model that can improve in the recall metric for this task compared to the binary classification. For all other models, even for the similar *Random Forest* model, a clear drop in performance can be observed (cf. Table 3). This might indicate that, compared to the other methods outlined in the experiment setup, the internal data balancing employed by the *Balanced Random Forest* model is more effective at handling the data imbalance.

The top three features for this task denote the historic likelihood of the warning type belonging to one of the three classes. Unlike the binary classification, where all kinds of other features were relevant, this model mostly focusses on historic features (cf. Table 1) and ones based on the file (e.g. source lines of code, test coverage, file size). The only commit-related feature is the number of modified files. No features are specific to the warning instance. This indicates that the historic data is important to detect the difference between the minority classes *tolerated* and *false-positive* warnings. This matches the experience that, for example, bugs in the static code analysis tool tend to affect only specific warning types and thus increase the likelihood of *false-positives* for this type in the historic data.

4.2.2 word2vec-based Models. As Tables 3 and 4 show, the *word2vec*+NN approach performs similarly on both the binary and the multiclass classification with recalls of 0.632 and 0.626, respectively. However, the multiclass classification model is outperformed by the *Balanced Random Forest* model (0.643). Similarly to the binary classification, we observe higher accuracy (0.681) compared to the feature-based model (0.454) with similar recall; consequently the multiclass *word2vec* model also tends to classify more frequently towards the majority class.

The hyperparameter-tuning resulted in a very different configuration for the multiclass classification tasks compared to the binary classification task, even though both neural networks use the same input size of $5 \cdot 128 + 173 = 813$ (five lines of code context embeddings and the one-hot-encoded warning type). While for the binary case the NN was considerably larger than a comparable model established in previous research (layer sizes 800) [16], for the multiclass classification a very small NN with only two layers of size 200 and 100 performs best. This aligns with our expectations considering the smaller dataset compared to the previous research.

4.2.3 Transformer-based Models. Table 4 also shows the best configuration for each of the Transformer embeddings. For the multiclass classification the choice of the best model is not as clear as it was for the binary classification task. *UniXcoder* performs better considering accuracy, F1, and precision, while *CodeT5+* performs worse according to those metrics but achieves the best recall score (0.573). However, as outlined in Section 3.4, we consider this to be the most important metric to reduce the number of false-negative predictions. Due to the big difference in recall score between *CodeT5+* and *UniXcoder* (0.11), we therefore also choose *CodeT5+* as the best model for the multiclass use-case. Only the other *CodeT5* variant achieves a similar recall score, but performs considerably worse on the other metrics. Finally, the best performing model configurations for the multiclass classification continues to use very small NN layers in sizes 200 or lower like in the binary case.

4.2.4 Model Comparison. Like in the binary case, the *Balanced Random Forest* is still the best of the feature-based models and the best Transformer-based model uses the *CodeT5+* embeddings. However, the Transformer-based model no longer performs best in the multiclass case, but instead the feature-based model achieves the highest recall, followed by the *word2vec* model, and the Transformer-based model coming last.

Table 5. Comparison of the best models.

Dataset	Model	Binary Classification				Multiclass Classification			
		Acc.	F1	Prec.	Recall	Acc.	F1	Prec.	Recall
Teamscale	BRF	0.375	0.350	0.555	0.631	0.454	0.339	0.407	0.643
	<i>word2vec</i>	0.725	0.547	0.557	0.632	0.681	0.409	0.401	0.626
	Transformer	0.736	0.564	0.569	0.661	0.698	0.422	0.411	0.573
Jacoco Agent	BRF	0.625	0.619	0.688	0.731	0.601	0.487	0.503	0.580
	<i>word2vec</i>	0.742	0.656	0.662	0.652	0.738	0.592	0.555	0.675
	Transformer	0.718	0.628	0.631	0.626	0.692	0.455	0.451	0.461

While the feature- and *word2vec*-based model achieve similar recall scores as for the binary classification, there is a considerable drop (0.661 to 0.573) in performance for the Transformer-based model. Compared to the binary classification, we observe a less balanced prediction with the precision score now being consistently lower for all models (binary: between 0.536 and 0.569, multiclass: 0.373–0.464). This indicates that even though a similar recall performance can be achieved by most models, the multiclass classification is nevertheless more challenging compared to the binary one.

The large drop in recall for the Transformer-based model might support the hypothesis that the NN-classifier can only extract the most important features from the large embedding vector (cf. Section 4.1.4). For the more complex multiclass classification, it might now be even more difficult to learn to extract the relevant features from the large *CodeT5+* embedding compared to extracting them from the smaller *word2vec* vector.

Summary RQ2: The feature-based model performs best for the multiclass classification task. The feature- and *word2vec*-based models can achieve similar recall scores for the binary and the multiclass classification. Nevertheless, this task is more challenging since the prediction precision drops considerably for all models.

4.3 RQ3: Cross-Project Prediction

In research questions RQ1 and RQ2 we chose a model architecture each and found the best-performing variant of it for the binary and multiclass tasks. Table 5 gives an overview over these models and their performance. It shows that the Transformer-based model has very similar metrics to the *word2vec*-based one for the binary task but still can outperform it across all evaluated metrics. Its multiclass counterpart also performs best regarding the accuracy, F1, and precision metrics. However, due to its low recall of 0.573 it is ultimately outperformed by both the *word2vec* model (recall 0.626) and the feature-based one (0.643). By prioritising the recall we aim to reduce the number of false-negative predictions, i.e. we want to avoid classifying actionable warnings as unactionable. The higher accuracy values for the *word2vec*- and Transformer-based models in both classification variants additionally indicate that these models tend to classify towards the majority class, i.e. the *actionable* label.

To save computing resources in practice, or to apply models to projects too small to be used for dedicated training, it may be interesting to pre-train the models on a larger project and then apply it to others. For that, the model must not only perform well on the original dataset but also be capable to generalise sufficiently. To evaluate this, Table 5 also shows the model performances when applied to the *Jacoco Agent* dataset (see Section 3.1) that was not seen during training. For

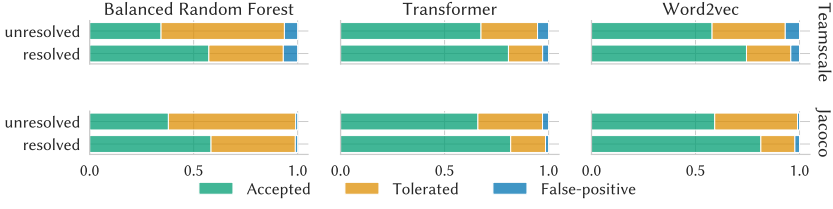


Fig. 3. Multiclass classifications for *accepted* warnings of the *Teamscale* and *Jacoco* datasets. *Accepted* but unresolved warnings are more frequently misclassified as *false-positive* compared to *accepted* warnings that have been resolved.

the binary classification case, now the feature-based model clearly outperforms the others with a recall of 0.731 compared to 0.652 for the next-best model. In the multiclass task, the *word2vec*-based model clearly performs best across all metrics with a recall of 0.675. The next-best model only achieves a recall of 0.580.

While the feature- and the *word2vec*-based model variants show improved metrics on the *Jacoco Agent* dataset, the Transformer does not. As outlined in Sections 3.3 and 4.1.3, the comparatively small training dataset likely results in an underfitted model which can retain the most relevant features of the original project to perform well on it, but cannot capture the nuances required to generalise to new projects. On the other hand, the *word2vec* model performs best across many metrics in the binary as well as the multiclass task. This generalisation ability may indicate that it can sufficiently capture and abstract from relevant re-appearing code structures.

In case of the feature-based model, the binary model performs well on the unseen project whereas the multiclass variant does not. The binary variant uses a diverse set of file- and commit-related features that may be similar across projects (see Table 1 and Section 4.1.1) while the multiclass variant instead focusses mostly on features calculated on historic data. Since the type and number of generated warnings differs considerably between projects, the information captured based on those historic features does not generalise to the new domain.

Especially at the beginning of new projects there is insufficient data available to train a dedicated model. Our results show that in practice a feature-based model could be pre-trained on an existing bigger project for the simpler binary classification task. Only when enough data could be collected for the target project, a new specifically trained multiclass model should be used.

Summary RQ3: As expected, for the feature-based model the use of project-independent features is important for generalisation. Most of our code-embedding-based models can generalise well to the other project. The binary classifiers generalise better than the multiclass ones.

4.4 RQ4: Influence of Unresolved Warnings

As Table 5 shows, the feature-based multiclass *Balanced Random Forest* model performs best considering the recall. However, as Fig. 3 highlights, it misclassifies more than 60 % percent of *accepted* warnings that have not yet been fixed (i.e. they are unresolved) as *tolerated*. While both the Transformer- and *word2vec*-based models perform worse according to the recall, they both are less likely to mark unresolved *accepted* warnings as *tolerated*, and the misclassification occurs only in 37 % and 41 % of the cases.

Figure 3 shows that the frequency with which unresolved warnings are misclassified is higher compared to the one for already resolved ones across all models and both datasets. This highlights a crucial challenge for the models due to the developer-labelled dataset: While the difference

between warnings that have been fixed (*accepted resolved*) and ones that are *false-positives* may have clear indicators, the difference between warnings where the developer has declared intent to fix them in the future (*accepted unresolved*) and ones that so far remained unresolved due to being *tolerated*, may be very nuanced. The high misclassification rate of the feature-based model shows that the hand-crafted feature-set cannot sufficiently capture these subtle differences. Using the *word2vec*-based code modelling considerably decreases the frequency of misclassifications, and the embeddings obtained from the pre-trained Transformers can decrease it further. This suggests that the difference between *accepted* and *tolerated* warnings is often observable in the code surrounding the warning rather than being inherent to the overall features of the code structure within the class or file.

Previous evaluations mined public *GitHub* repositories to search for warning-resolving commits [20, 16]. Since these datasets then do not contain ambiguous cases between *accepted unresolved* and *tolerated* warnings, it might be easier for a model to learn to differentiate between the more clearly separated *actionable* and *unactionable* groups. For example, Hegedus and Ferenc [16] explicitly mined commits that contained references to fixed warnings to collect *actionable* code locations from the state of the code before the commit. They then separately collected *unactionable* warnings from code locations that were marked by developers as to be ignored by the tool. Since we re-implemented their model as part of our evaluations, we can compare the influence of the dataset on the classification performance. The difference may of course also be influenced by other factors like for example projects considered in the experiment and the applied static code analysis tool (*SonarQube* vs. *Teamscale*), even though *Teamscale* includes many warnings reported by *SonarQube* (cf. Section 2.1). As noted in Section 3.3, we trained our model also on their replication package, but are not convinced our results for this experiment are representative. Therefore, we consider the comparison on different datasets and expect that it can still serve as an indicator for the differences caused by the additional challenges of the more nuanced labels. Hegedus and Ferenc [16] report a recall of 0.778 (precision: 0.831, accuracy: 0.902, F1-score: 0.803) on their evaluation of the *word2vec*-based model. Our model can only achieve a recall of 0.632 (precision: 0.557, accuracy: 0.725, F1-score: 0.547). Across all metrics, the model trained on our dataset performs considerably worse, thus highlighting the additional challenge of the nuanced *accepted unresolved* and *tolerated* warning labels.

Summary RQ4: While our developer-labelled dataset allows for a fine-granular reflection of developer-intention, this granularity and accompanying subtle difference between not yet vs. intentionally not fixed warnings poses additional challenges for all machine learning models evaluated in this paper. These challenges are not observable in heuristically determined labels.

4.5 RQ5: Misclassifications of Tolerated and False-positive Warnings

While the choice of recall as the main evaluation metric intends to minimise the number of *actionable* warnings being shown as *unactionable*, it should not result in the models frequently presenting *unactionable* warnings as *actionable* either. This would result in the model no longer achieving the main goal of reducing the number of irrelevant warnings a developer has to check.

As Fig. 4a shows, the feature-based model is the only one of the binary models that can predict both types of *unactionable* warnings with an error rate of less than 5 %. Both the *word2vec*- and Transformer-based model have an error-rate of 45 % for *tolerated* and 30 % for *false-positive* warnings. Also as the only model, the *Balanced Random Forest* (BRF) can transfer this classification ability to the other dataset. While this model can still identify non-actionable warnings nearly perfectly, the other two models have an error-rate of 75 % for *false-positives*. The near perfect ability to predict

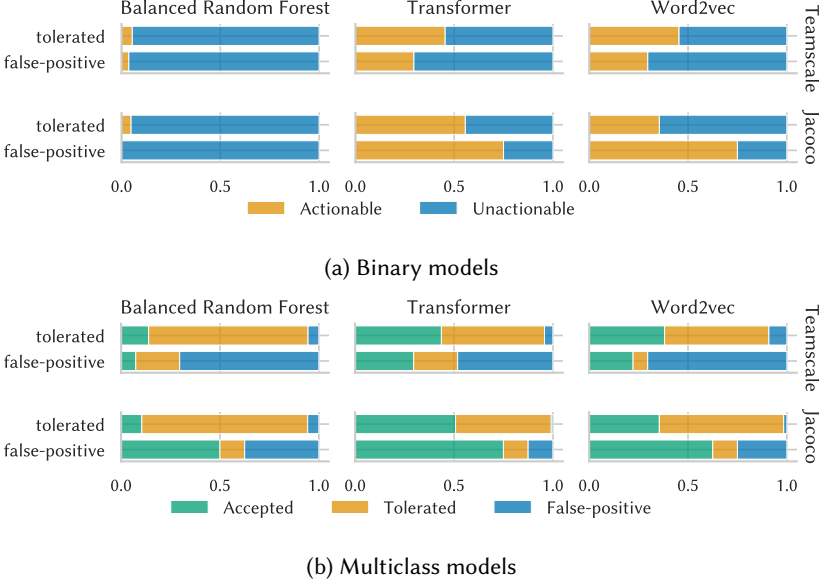


Fig. 4. Classification of *tolerated* and *false-positive* warnings on the two datasets. The y-axis marks the actual label; the x-axis represents the proportions of the predicted labels.

the minority class combined with the non-weighted recall metric results in the feature-based model achieving the best recall score on the *Jacoco* dataset (cf. Section 4.3 and Table 5).

As demonstrated in our preliminary experiment in Section 3.3, the binary Transformer-based model fails to generalise to a different dataset using a different static code analysis tool. The metrics outlined here also suggest that the code embedding vector alone is not enough to support such a generalisation. However, the performance of the BRF on the unseen *Jacoco* dataset suggests that these features could help to overcome this limitation by creating a model that combines the code embedding vector and hand-crafted features in its input.

The pattern observable in the binary classification task repeats in the multiclass case (Fig. 4b). The BRF again shows the best performance and also achieves the best overall recall score on the *Teamscale* dataset. In this instance it fails to apply the learned information to the *false-positive* warnings of the *Jacoco* dataset and misclassifies more than half of them. Both the *word2vec* and Transformer models can predict the *false-positives* even less frequently, but for both of them this is also the case for the binary variant.

The types of misclassifications of the *tolerated* warnings again highlight the challenge of the nuanced distinction between *accepted unresolved* and *tolerated* as discussed in Section 4.4. For all models, a *tolerated* warning is more likely to be misclassified as *actionable* than *false-positive*. The *word2vec*-based model falsely predicts 38 % of the *tolerated* warnings to be *actionable*, for the Transformer this ratio is even higher with 44 %. The models only rarely (for less than 9 % of the cases) wrongfully assign the *false-positive* label.

Summary RQ5: Only the feature-based model can clearly detect *tolerated* warnings. Both embedding models frequently misclassify this warning type as *accepted*. All models can detect *false-positives* more reliably and are less likely to present them as *actionable*.

Table 6. Warning prioritisation proposed by the models compared to the intended priority measured as Kendall- τ for the best-performing binary (τ_b) and multiclass (τ_m) models. Differences between binary and multiclass rankings (τ_{bm}) and its significance level p_{bm} are also measured using Kendall- τ .

Model	Teamscale				Jacoco Agent			
	τ_b	τ_m	τ_{bm}	p_{bm}	τ_b	τ_m	τ_{bm}	p_{bm}
BRF	0.179	0.129	0.017	0.25	0.393	0.254	0.213	8×10^{-13}
<i>word2vec</i>	0.158	0.168	0.058	9×10^{-5}	0.380	0.383	0.215	6×10^{-13}
Transformer	0.187	0.196	0.058	0.000 1	0.204	0.338	0.060	0.04

4.6 RQ6: Alignment of Warning Prioritisation with Developer Intention

While classification models give rough guidance on which warnings should be prioritised, they do not clarify which warnings within the categories are more important than others. We therefore evaluate whether our models can also be directly used as ranking models.

With the feature-based *Balanced Random Forest* (BRF) performing slightly better when trained for multiclass classification according to the recall metric, the ranking produced by this model variant could be expected to be better than the binary one. However, as Table 6 shows, the ranking of the binary model is slightly better ($\tau_b = 0.179$, $p = 3 \times 10^{-33}$) than the one of the multiclass model ($\tau_m = 0.129$, $p = 5 \times 10^{-18}$). The statistically insignificant difference ($p_{bm} = 0.25$) between the two rankings reflects the closeness of the recall scores between the two model variants. For both the *word2vec*- and Transformer-based models, the pattern is reversed. The binary model variants have higher recall scores, but the ranking of the multiclass models is better. In both cases the difference is small with $\tau_{bm} = 0.058$ but is statistically significant in both cases ($p = 9 \times 10^{-5}$ for *word2vec*, $p = 1 \times 10^{-4}$ for the Transformer).

Interestingly, even though the feature-based models were the only ones to which the warning time to live (TTL) sorting criterion was in principle available as a feature during training, the *Balanced Random Forest* does not perform best overall. Instead, the multiclass Transformer-based model achieves the best τ coefficient of $\tau_m = 0.196$. Since more important features were retained during the feature elimination step (see Section 3.2.1), explicitly retaining the TTL feature would therefore worsen the classification performance, but could on the other hand improve the model performance when ranking the warnings afterwards. However, this feature might have automatically been excluded due to it reflecting noisy data. Some warnings may be fixed within a day, while others remain unresolved for multiple years. This suggests that the TTL could still be integrated into a feature-based model by discretising the time range (e.g. the warning remained unresolved for one day/week/month/year). Additionally, the TTL may not directly relate to the warning’s importance. Some important warnings may be missed and remain unresolved while others may be resolved accidentally by moving code to another file. Training a regression model directly on the TTL to rank the warnings is unlikely to be successful due to the high amount of noise.

For the rarely occurring classes of *tolerated* and *false-positive* warnings, only the multiclassification training of the model allows it to discern between these two cases. In cases where the multiclass model misclassifies a *false-positive* label into the *tolerated* group, it is still more likely to end up closer to the intended ranking since the primary ranking decision is made based on the labels while the confidence value only decides on the ranking within the relatively small group of *tolerated* warnings. The imbalance of only few *tolerated* and *false-positive* warnings explains why this effect is not more pronounced and thus cannot improve the overall ranking much over the one generated by the binary model.

The τ rank correlation coefficients below 0.2 for all variants on the *Teamscale* dataset suggest that overall the rankings proposed by the models are not close to the ground-truth. To estimate whether this is only due to the number of datapoints that need to be ranked, or if the model indeed cannot successfully rank the warnings, we apply it to the smaller Jacoco dataset containing only 504 items. The higher ranking coefficients ranging between 0.338 and 0.393 show that the ranking metric considerably improves for the smaller dataset, even though the models were not trained to rank its warnings. This indicates the models have learned to predict a useful ranking and can generalise the trained information to another project.

The same pattern as for the *Teamscale* dataset emerges for the Jacoco dataset: For the feature-based model, the binary variant performs better, while for the other two models the multiclass variants yield the best results. In this case the differences between all pairs of binary and multiclass rankings are statistically significant. Nevertheless, since the τ correlation coefficients below 0.4 for this dataset are still far away from a perfect ranking, this highlights that future improvements to optimise the ranking should be made.

Summary RQ6: The rankings produced by the multiclass model variants are closer to the intended ranking compared to the ones produced by the binary classification models in most cases. Nevertheless, further improvements are necessary to ensure the proposed ranking matches the developer intention more closely.

5 Related Work

To the best of our knowledge, this is the first study that does not model the problem of machine-learning-based static code analysis warning classification as a binary task. We use a more fine granular multiclass classification into *accepted*, *tolerated*, and *false-positive* warnings. Combined with the developer-labelled dataset, this highlights additional challenges for the classification (Section 4.4) which could be avoided previously by using clearer automatically mined labels (cf. Section 2.3). Additionally, this is the first study applying this classification to warnings generated by the *Teamscale* [18] analysis tool.

Following previous research [38, 20, 32], we evaluate in RQ1 whether feature-based models usually applied to the binary classification problem can also be used for the multiclass one. Our experiment highlights that such models can successfully be adapted to new challenges posed by the dataset and also different types of warnings as generated by a different static code analysis tool.

Continuing to follow prior work, we extend the *word2vec*-based approach by Hegedus and Ferenc [16] to the multiclass classification as part of RQ2. Similarly, Lee et al. [25] also used *word2vec* to generate the token-embeddings, but used a convolutional neural network in the classification step. As a simpler alternative to *word2vec* as the language model, also long short-term memory (LSTM)-based approaches have been proposed [29]. To adapt the *word2vec*-based model [16] to recent developments in language modelling, we additionally show that a more powerful Transformer-based language model could potentially be a more accurate classifier, but only when given a sufficiently large training dataset (RQ2). To further shift the focus towards language-modelling, Kharkar et al. [23] fine-tuned Transformer-based language models to tackle this problem. Since the dataset size already posed potentially problematic for the smaller model in our experiment (RQ2), we did not evaluate a similar approach but instead consider it to be a promising future combination with our work if a sufficiently large developer-labelled dataset is available. However, their experiments focussed on classifying a single warning type. This matches other experiments also either using a small set of warning types [25], or using similar models to classify potential security vulnerabilities instead [29]. We train our model to classify all 173 warning types appearing in our dataset.

6 Conclusions

Developers can benefit from tool support in the form of static analysis warnings. These tools tend to create many false-positive warnings hampering the resolution of actually important ones. While prior results are already promising, they tend to apply only a rough classification of warnings into *actionable* and *unactionable* ones. Since these labels are heuristically determined from unlabelled open-source data, an open question remains whether more fine-granular developer-labelled data can improve the classification. To answer this question, we gathered a multiclass dataset based on the meticulous developer labelling applied at *CQSE*, and adapted previously proposed models to this new domain.

While our experiments show no conclusive improvement for the multiclass warning classification task over the binary case, they reveal several limitations adherent to the machine-learning based warning classification. It is not enough to evaluate the models only to standardised metrics, but a deeper inspection of the results is necessary to understand which scenarios are challenging for the models. Our use of developer-labelled data shows that the assumption of clearly separated labels is not true in practice, but that there exists a nuanced difference between relevant warnings that should be fixed later and irrelevant ones that remain unresolved. This highlights the importance of gathering large manually labelled datasets and of future research on the development of labelling heuristics that can reflect the developer intention to then align the prediction models to real-world practices.

A further challenge in practice is the large number of valid warnings generated by the static analysis tools. Therefore, they not only need to be classified, but also prioritised within the *actionable* category. Our evaluations on this task highlight a difference in model performance when considering the basic classification metrics versus their performance in the real-world inspired application. Even though multiclass models did not conclusively outperform binary ones in the previous experiments, when applying our models to the prioritisation task we observed contrasting results highlighting the requirement of performing a more in-depth analysis of the model performance. We observed a statistically significant improvement of the warning ranking as generated by the multiclass models compared to the rankings generated by the binary model variants.

In this paper, we only consider the warnings generated for the default branch of the projects. However, in practice, many warnings are already fixed by developers within the feature branches or merge requests before the changes are merged into the default branch. Our experiments therefore aim to predict which long-standing warnings of the project should be prioritised to increase long-term code quality, whereas future research could focus on the fast feedback cycle before the merge. Due to the different requirements, we expect that especially for the feature-based models a different set of relevant features will emerge. Since there are also many more commits and therefore warnings within feature branches compared to the default one, our newly proposed Transformer-based model that was hampered by the lack of enough training data could potentially work to its full potential in this scenario. This distinction between short-lived and long-term warnings is not only important when using developer-labelled data, but also should be taken into account when mining open-source repositories to ensure that models are trained on homogeneous data reflecting the intended use-case.

Data Availability

We provide the implementations of the models, their training and evaluation scripts, the raw evaluation results, and the scripts used to generate the data and plots shown in this paper at <https://figshare.com/s/2d35074bc8e49fcfab3b>.

References

- [1] Vipin Balachandran. 2013. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *International Conference on Software Engineering (ICSE)*. IEEE, (May 2013). doi:[10.1109/icse.2013.6606642](https://doi.org/10.1109/icse.2013.6606642).
- [2] Gustavo E. A. P. A. Batista, Ana L. C. Bazzan and Maria Carolina Monard. 2003. Balancing training data for automated annotation of keywords: a case study. In *Brazilian Workshop on Bioinformatics*, 10–18.
- [3] Moritz Beller, Radjino Bholanath, Shane McIntosh and Andy Zaidman. 2016. Analyzing the state of static analysis: a large-scale evaluation in open source software. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, (Mar. 2016). doi:[10.1109/saner.2016.105](https://doi.org/10.1109/saner.2016.105).
- [4] N. V. Chawla, K. W. Bowyer, L. O. Hall and W. P. Kegelmeyer. 2002. Smote: synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16, (June 2002), 321–357. doi:[10.1613/jair.953](https://doi.org/10.1613/jair.953).
- [5] Chao Chen, Andy Liaw and Leo Breiman. 2004. Using random forest to learn imbalanced data. Tech. rep. University of California, Berkeley.
- [6] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: a scalable tree boosting system. In *International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, (Aug. 2016), 785–794. doi:[10.1145/2939672.2939785](https://doi.org/10.1145/2939672.2939785).
- [7] Dino Distefano, Manuel Fähndrich, Francesco Logozzo and Peter W. O’Hearn. 2019. Scaling static analyses at facebook. *Communications of the ACM*, 62, 8, (July 2019), 62–70. doi:[10.1145/3338112](https://doi.org/10.1145/3338112).
- [8] Zhangyin Feng et al. Codebert: a pre-trained model for programming and natural languages. (Feb. 2020). doi:[10.48550/arxiv.2002.08155](https://doi.org/10.48550/arxiv.2002.08155).
- [9] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou and Jian Yin. Unixcoder: unified cross-modal pre-training for code representation. (Mar. 2022). doi:[10.48550/arxiv.2203.03850](https://doi.org/10.48550/arxiv.2203.03850).
- [10] Daya Guo et al. Graphcodebert: pre-training code representations with data flow. (Sept. 2020). doi:[10.48550/arxiv.2009.08366](https://doi.org/10.48550/arxiv.2009.08366).
- [11] Isabelle Guyon, Jason Weston, Stephen Barnhill and Vladimir Vapnik. 2002. Gene selection for cancer classification using support vector machines. *Machine Learning*, 46, 1/3, 389–422. doi:[10.1023/a:1012487302797](https://doi.org/10.1023/a:1012487302797).
- [12] Andrew Habib and Michael Pradel. 2018. How many of all bugs do we find? a study of static bug detectors. In *International Conference on Automated Software Engineering (ASE)*. ACM, (Sept. 2018). doi:[10.1145/3238147.3238213](https://doi.org/10.1145/3238147.3238213).
- [13] Quinn Hanam, Lin Tan, Reid Holmes and Patrick Lam. 2014. Finding patterns in static analysis alerts: improving actionable alert ranking. In *Conference on Mining Software Repositories (MSR)*. ACM, (May 2014), 152–161. doi:[10.1145/2597073.2597100](https://doi.org/10.1145/2597073.2597100).
- [14] Sarah Heckman and Laurie Williams. 2009. A model building process for identifying actionable static analysis alerts. In *International Conference on Software Testing Verification and Validation (ICST)*. IEEE, (Apr. 2009), 161–170. doi:[10.1109/icst.2009.45](https://doi.org/10.1109/icst.2009.45).
- [15] Sarah Heckman and Laurie Williams. 2008. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *international symposium on Empirical software engineering and measurement (ESEM)*. ACM, (Oct. 2008), 41–50. doi:[10.1145/1414004.1414013](https://doi.org/10.1145/1414004.1414013).
- [16] Peter Hegedus and Rudolf Ferenc. 2022. Static code analysis alarms filtering reloaded: a new real-world dataset and its ml-based utilization. *IEEE Access*, 10, 55090–55101. doi:[10.1109/access.2022.3176865](https://doi.org/10.1109/access.2022.3176865).
- [17] Peter Hegedus and Rudolf Ferenc. 2022. Static code analysis alarms filtering reloaded: an ml approach and its empirical evaluation on a new real-world dataset. (2022). doi:[10.5281/zenodo.5885654](https://doi.org/10.5281/zenodo.5885654).
- [18] Lars Heinemann, Benjamin Hummel and Daniela Steidl. 2014. Teamscale: software quality control in real-time. In *International Conference on Software Engineering (ICSE)*. ACM, (May 2014), 592–595. doi:[10.1145/2591062.2591068](https://doi.org/10.1145/2591062.2591068).
- [19] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill and Robert Bowdidge. 2013. Why don’t software developers use static analysis tools to find bugs? In *International Conference on Software Engineering (ICSE)*. IEEE, (May 2013). doi:[10.1109/icse.2013.6606613](https://doi.org/10.1109/icse.2013.6606613).
- [20] Hong Jin Kang, Khai Loong Aw and David Lo. 2022. Detecting false alarms from automatic static analysis tools: how far are we? In *International Conference on Software Engineering (ICSE)*. ACM, (May 2022), 698–709. doi:[10.1145/351003.3510214](https://doi.org/10.1145/351003.3510214).
- [21] [SW] Hong Jin Kang, Khai Loong Aw and David Lo, Retrospective of Golden Features for detecting actionable warnings 2022. URL: https://github.com/soarsmu/SA_retrospective, swhid: {swh:1:rev:75a3284c22687d491fbcc2b00f5a044253c00611;origin=https://github.com/soarsmu/SA_retrospective;visit=swh:1:snp:a06f3b159947493c6657cf7d2f3ab371941654dc}.
- [22] Maurice G. Kendall. 1938. A new measure of rank correlation. *Biometrika*, 30, 1–2, (June 1938), 81–93. doi:[10.1093/biomet/30.1-2.81](https://doi.org/10.1093/biomet/30.1-2.81).
- [23] Anant Kharkar, Roshanak Zilouchian Moghaddam, Matthew Jin, Xiaoyu Liu, Xin Shi, Colin Clement and Neel Sundaresan. 2022. Learning to reduce false positives in analytic bug detectors. In *International Conference on Software Engineering (ICSE)*. ACM, (May 2022). doi:[10.1145/3510003.3510153](https://doi.org/10.1145/3510003.3510153).

- [24] Sunghun Kim and Michael D. Ernst. 2007. Which warnings should i fix first? In *Joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering (ESEC/FSE)*. ACM, (Sept. 2007), 45–54. doi:[10.1145/1287624.1287633](https://doi.org/10.1145/1287624.1287633).
- [25] Seongmin Lee, Shin Hong, Jungbae Yi, Taeksu Kim, Chul-Joo Kim and Shin Yoo. 2019. Classifying false positive static checker alarms in continuous integration using convolutional neural networks. In *International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, (Apr. 2019), 391–401. doi:[10.1109/icst.2019.00048](https://doi.org/10.1109/icst.2019.00048).
- [26] Guangtai Liang, Ling Wu, Qian Wu, Qianxiang Wang, Tao Xie and Hong Mei. 2010. Automatic construction of an effective training set for prioritizing static analysis warnings. In *International Conference on Automated Software Engineering (ASE)*. ACM, (Sept. 2010), 93–102. doi:[10.1145/1858996.1859013](https://doi.org/10.1145/1858996.1859013).
- [27] Anton Lozhkov et al. Starcoder 2 and the stack v2: the next generation. (Feb. 2024). doi:[10.48550/arxiv.2402.19173](https://doi.org/10.48550/arxiv.2402.19173).
- [28] Tomas Mikolov, Kai Chen, Greg Corrado and Jeffrey Dean. Efficient estimation of word representations in vector space. (Jan. 2013). doi:[10.48550/arxiv.1301.3781](https://doi.org/10.48550/arxiv.1301.3781).
- [29] Kien-Tuan Ngo, Dinh-Truong Do, Thu-Trang Nguyen and Hieu Dinh Vo. 2021. Ranking warnings of static analysis tools using representation learning. In *Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, (Dec. 2021), 327–337. doi:[10.1109/apsec53868.2021.00040](https://doi.org/10.1109/apsec53868.2021.00040).
- [30] Sebastiano Panichella, Venera Arnaudova, Massimiliano Di Penta and Giuliano Antoniol. 2015. Would static analysis tools help developers with code reviews? In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, (Mar. 2015), 161–170. doi:[10.1109/saner.2015.7081826](https://doi.org/10.1109/saner.2015.7081826).
- [31] Nadeem Qazi and Kamran Raza. 2012. Effect of feature selection, smote and under sampling on class imbalance classification. In *International Conference on Computer Modelling and Simulation (UKSim)*. IEEE, (Mar. 2012). doi:[10.1109/uksim.2012.116](https://doi.org/10.1109/uksim.2012.116).
- [32] Mingshuang Qing, Xiang Feng, Jun Luo, Wanmin Huang, Jingui Zhang, Ping Wang, Yong Fan, Xiuting Ge and Ya Pan. 2021. A machine learning-based static analysis warning prioritization. In *International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, (Dec. 2021), 685–690. doi:[10.1109/qrs-c55045.2021.00103](https://doi.org/10.1109/qrs-c55045.2021.00103).
- [33] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon and Ciera Jaspan. 2018. Lessons from building static analysis tools at google. *Communications of the ACM*, 61, 4, (Mar. 2018), 58–66. doi:[10.1145/3188720](https://doi.org/10.1145/3188720).
- [34] Ferdian Thung, Lucia, David Lo, Lingxiao Jiang, Foyzur Rahman and Premkumar T. Devanbu. 2014. To what extent could we detect field defects? an extended empirical study of false negatives in static bug-finding tools. *Automated Software Engineering*, 22, 4, (Sept. 2014), 561–602. doi:[10.1007/s10515-014-0169-8](https://doi.org/10.1007/s10515-014-0169-8).
- [35] Kristin Fjola Tomasdottir, Mauricio Aniche and Arie van Deursen. 2017. Why and how javascript developers use linters. In *International Conference on Automated Software Engineering (ASE)*. IEEE, (Oct. 2017). doi:[10.1109/ase.2017.8115668](https://doi.org/10.1109/ase.2017.8115668).
- [36] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall and Andy Zaidman. 2019. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 25, 2, (Nov. 2019), 1419–1457. doi:[10.1007/s10664-019-09750-5](https://doi.org/10.1007/s10664-019-09750-5).
- [37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Pape_r.pdf.
- [38] Junjie Wang, Song Wang and Qing Wang. 2018. Is there a “golden” feature set for static warning identification?: an experimental evaluation. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, (Oct. 2018). doi:[10.1145/3239235.3239523](https://doi.org/10.1145/3239235.3239523).
- [39] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li and Steven Hoi. 2023. Codet5+: open code large language models for code understanding and generation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, (Dec. 2023). doi:[10.18653/v1/2023.emnlp-main.68](https://doi.org/10.18653/v1/2023.emnlp-main.68).
- [40] Yue Wang, Weishi Wang, Shafiq Joty and Steven C. H. Hoi. 2021. Codet5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, (Nov. 2021). doi:[10.18653/v1/2021.emnlp-main.685](https://doi.org/10.18653/v1/2021.emnlp-main.685).
- [41] Xueqi Yang, Jianfeng Chen, Rahul Yedida, Zhe Yu and Tim Menzies. 2021. Learning to recognize actionable static code warnings (is intrinsically easy). *Empirical Software Engineering*, 26, 3, (Apr. 2021). doi:[10.1007/s10664-021-09948-6](https://doi.org/10.1007/s10664-021-09948-6).
- [42] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora and Massimiliano Di Penta. 2017. How open source projects use static code analysis tools in continuous integration pipelines. In *International Conference on Mining Software Repositories (MSR)*. IEEE, (May 2017). doi:[10.1109/msr.2017.2](https://doi.org/10.1109/msr.2017.2).
- [43] Min Zeng, Beiji Zou, Faran Wei, Xiyao Liu and Lei Wang. 2016. Effective prediction of three common diseases by combining SMOTE with Tomek links technique for imbalanced medical data. In *International Conference of Online Analysis and Computing Science (ICOACS)*. IEEE, (May 2016). doi:[10.1109/icoacs.2016.7563084](https://doi.org/10.1109/icoacs.2016.7563084).