

AI-Based Duplicate Bug Detection and Intelligent Bug Relationship Analysis Using Natural Language Processing

Sheetal Rajbhar¹, Cynthia Sankhe²

¹*Sonpant Dhandekar Shikshan Mandali*

²*Professor, Sonpant Dhandekar Shikshan Mandali*

Abstract—Most bug tracking systems used in the software development lifecycle treat individual bug reports as isolated entries with no inherent means of determining if one report describes the same defect as another. Given two reports, one stating “the application crashes when I try to log in” and another stating “the application closes after submission of login credentials,” it is entirely reasonable to assume that they describe the same bug, but a keyword-based system would be incapable of recognizing this. This project implements a bug management system with an intelligent layer recognizing potential duplicates based on semantic similarities between reports using sentence embeddings and TF-IDF comparison, providing an alternative to the usual create-view-update-assign cycle in bug tracking software. For each new report, semantic similarity is calculated using two different approaches – cosine similarity of TF-IDF vectors and cosine similarity of pretrained Sentence-Transformer embeddings – and the report is classified as a potential duplicate, related to an existing report, or distinct based on the combined similarity score. Additionally, a rule-based system is used to assign a priority and severity classification to each report with natural language reasoning provided for each recommendation to allow for manual review. The system was implemented as a functioning web application using FastAPI, MySQL, and React instead of a standalone script, and benchmarked using a manually-created labeled dataset of eight reports with three pairs of duplicates.

Index Terms—Duplicate Bug Detection, Natural Language Processing, Semantic Similarity, Sentence Embeddings, Transformer Models, TF-IDF, Cosine Similarity, Bug Triage, Software Testing

I. INTRODUCTION

Any software project which is developed for more than a sufficiently short period of time accumulates a significant amount of bug reports. Some of these reports contain duplicate information — either because the reporter was not aware of an existing report with the same issue or the tracker was not able to find the required report.

The following example demonstrates a case which is fairly common in the wild. Given a tracker with the following three reports, it would be hard for a person to tell which of them are duplicates:

Bug #101: Login page crashes when I enter the password.

Bug #235: Application closes after submitting login credentials.

Bug #417: Login functionality is crashing during password submission.

A tracker which relies solely on keyword matching would not be able to detect duplicates in this case. It can only check if a report contains the same words as the previous ones, and in this example, there are none. The reason for this is that duplicate reports do not necessarily have any of the same keywords, and the only way to tell if two reports are similar is by analyzing their meaning.

The purpose of this project is to detect such duplicates. In practice, this means that instead of a human having to manually search for existing reports every time before submitting a new one (or a tracker not being able to find them due to lack of keywords), a new

report can be automatically checked against a database of existing reports using natural language processing. For each new report, the system finds existing similar reports and suggests classifications for them (duplicate, related, or new) along with the probability for each. It also classifies the report itself with a priority level and severity and explains the reasoning for its conclusions.

II. EXISTING SYSTEM

Conventional bug tracking systems — including widely used tools in the industry — generally provide the following standard capabilities: creating a bug report with fields such as title, description, steps to reproduce, severity, and priority; assigning the bug to a developer; updating its status as it moves through the workflow; and a text search feature for locating existing reports.

The search feature in most of these systems is built on basic text matching or, at best, a full-text search index. This is adequate for finding a bug when the searcher already knows roughly what words were used in the original report, but it does not help in the more common situation — a tester encountering a new-looking problem and having no way of knowing it was already reported under different wording.

Academic research on this exact issue goes back further than one might expect. Runeson, Alexandersson, and Nyholm published one of the earliest studies applying natural language processing to duplicate bug detection, working with bug reports from a commercial software company and showing that NLP-based comparison could flag a meaningful share of duplicates that manual triage missed [1]. Wang, Zhang, Xie, Anvik, and Sun extended this line of work by combining natural language information with execution trace information, since many bug trackers also store stack traces and crash logs that carry additional signal beyond the written description [2]. Sun, Lo, Khoo, and Jiang combined information retrieval techniques with topic modeling to improve retrieval accuracy on larger bug repositories [3]. Sureka and Jalote took a different angle, using character n-gram features rather than word-level features, arguing this makes the comparison more robust to spelling variation and domain-specific terminology [4].

What these studies have in common is that they treat duplicate detection as a retrieval problem — given a new bug report, rank the existing reports by how likely each one is to be the same issue — and most of the earlier work relies on relatively traditional information retrieval techniques such as term frequency weighting, topic models, or n-gram overlap. More recent work has started incorporating transformer-based language models, which can capture meaning beyond surface wording. Reimers and Gurevych's Sentence-BERT work, while not specific to bug reports, established that sentence-level embeddings from a fine-tuned transformer model can be compared directly using cosine similarity and produce results that correlate well with human judgments of sentence similarity [5]. This is the technique this project's semantic similarity component is built on, applied specifically to the domain of bug report text.

III. PROPOSED SYSTEM

The system described in this project is one for bug tracking with an additional intelligence layer that activates the moment a tester submits a report with a bug. Instead of simply storing the information and waiting for someone to check if this is not just a duplicate that the testers have already encountered, the system actively analyzes this new report in comparison to all of the bugs in the database and notifies the claimer if there are known issues that are similar or related to the one that the user is trying to report or if it is indeed an unprecedented problem.

There are two types of similarity checks performed, and both have their own unique approach. The first is the TF-IDF model that analyzes the weight of the terms that the bug report contains in comparison to the rest of the database. The second is the sentence-embeddings model that attempts to analyze the context of all of the information provided and try to assess the similarity between them. The similarity scores of both approaches are then combined, with the higher weight being assigned to the sentence-embedding model because it is supposed to be more accurate. The resulting value is then compared to pre-set thresholds that decide whether the new report is similar enough to an existing one to be a duplicate, similar enough to have some relation to the bugs reported before, or completely unique.

The similarity rankings are then combined with the relevance thresholds for each similarity score and presented to the tester along with a list of reports that the new one was found to be similar to, along with the percentages for each. It is important that the final decision whether a report is a duplicate or not is always up to the tester. The same goes for the priority suggestions; another pair of rule-based algorithms scans the report and tries to assign a set of features to it, such as a crash, data loss, privacy or security issue, and other possible problems. The weights for these features then determine what priority the report should have in the system, and, similarly to the similarity checks, the user can see which features have triggered which recommendations.

The system also has a search function that analyzes natural language queries and attempts to find existing bugs that are most similar to the ones described by the query. This is a great showcase for the performance of the semantic similarity algorithms. While the database contained a report for not being able to receive a password reset email, when a user searched for “not getting email after login”, the system still managed to prioritize this report among the search results.

However, the system described above is not meant to be an additional tool that only gets used by testers when they need to check if an issue that they discovered has been already reported. Instead, it is fully integrated into the bug tracking system as a set of features that can be used by the testers, the developers, and the administrators. Each of them has their own set of tools available only to them and appropriate for their tasks. A bug can be claimed by a tester, then assigned to a developer, commented upon by both sides, edited if necessary, and marked as resolved when the issue is fixed. The AI components are also implemented directly into the interface so that the users do not feel overwhelmed by the amount of information that they need to process. The idea behind this approach was to make the value of applying such techniques clear to the user of the system. A feature for checking reports on duplicates only makes sense when it is built into the tool that is used to report bugs, not an additional standalone utility.

IV. PROPOSED METHODOLOGY

The overall flow the system uses for each new bug report is:

Bug Submission → Text Preprocessing → Feature/Semantic Representation → Similarity Calculation → Classification → Priority/Severity Recommendation → Explanation → Result Shown to User

The title, description, steps to reproduce, expected result, actual result, and component fields of each new bug are concatenated into one block of text. This text is then cleansed (lowercased, URLs and other strange characters removed, tokenized, stopwords removed, and each token lemmatized) and is represented by two parallel encodings:

A TF-IDF vector, created using scikit-learn's TfidfVectorizer, which provides a ranking of how informative each word (and bigram!) is for this bug relative to the others in the set.

A semantic embedding, a 384-vector representation created using a pretrained Sentence-Transformer, which is supposed to encode the meaning of the text rather than the specific words used.

These two representations are compared to every other bug report in the database using cosine similarity, which is a measure of the angle between two vectors (with 1 being completely overlapping and 0 being not similar at all).

The two similarity scores are then combined using a weighted average (30 percent TF-IDF, 70 percent semantic) to get a final score. This was a design decision, rather than an optimally tuned one, as the semantic representation is likely to be more robust to the kinds of text variation described in the introduction, while the TF-IDF is a quick score that still captures some signals that are definitely useful in cases where two reports are using the same error codes, modules, etc.

The combined score is compared to two thresholds to classify the bug:

Score > 0.85 → Potential Duplicate

Score > 0.60 (but < 0.85) → Related

Score < 0.60 → New

Again, these were decided using the same intuition as the weighting, and are stored as configuration variables rather than being written directly into the algorithm. They also have not been tuned on a large labeled dataset - which is discussed more honestly in the Limitations and Future Scope sections.

V. RESULT

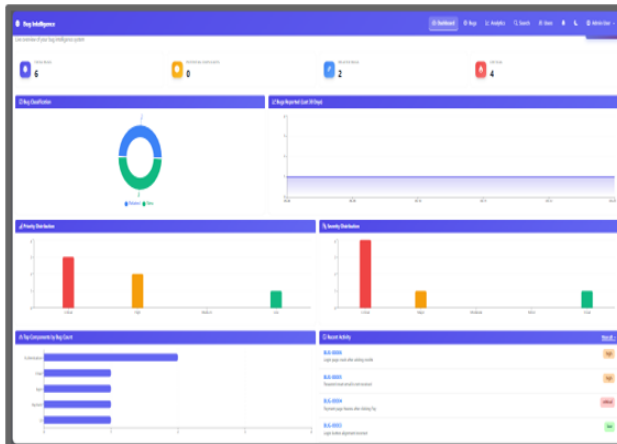


Fig.1. The Dashboard

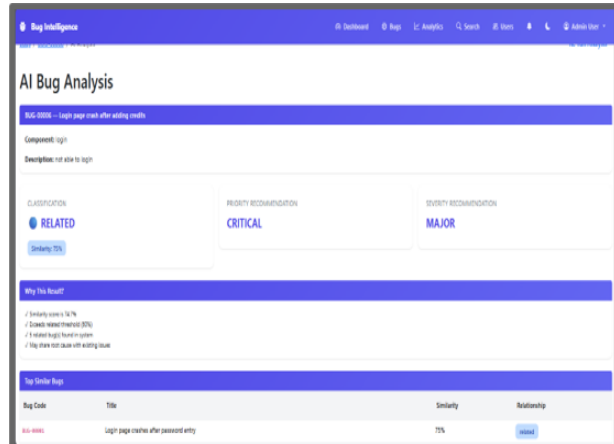


Fig.2. The AI Analysis result page

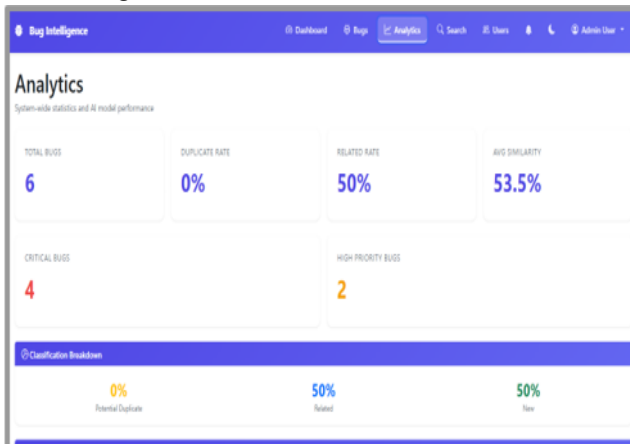


Fig.3. The Analytics page's Model Comparison table

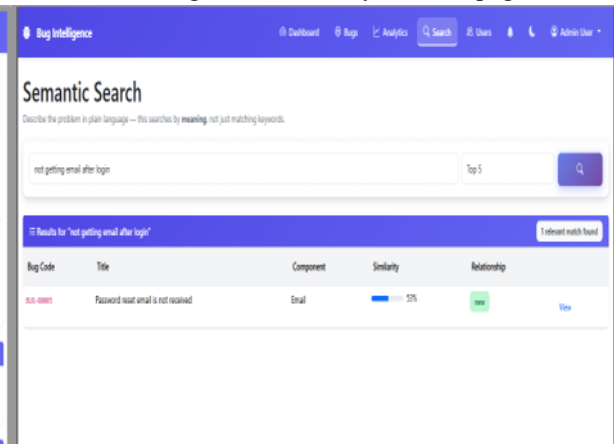


Fig.4. The Semantic Search result

VI. LIMITATIONS

- The app sends notifications such as “bug assigned to you” or “critical bug detected” by periodic polling (every 30 seconds while the app is opened) rather than real-time push — this means that a fresh notification may take up to half a minute to appear in the interface.
- The app does not currently support attaching any files (such as screenshots or log files) to bug reports, despite there being a database table for this.

VII. FUTURE SCOPE

- Implement CSV-based bulk import of bug datasets, which is part of the original project scope

but deferred to allow the current version's scope to remain focused on the essentials.

- Add file attachment support for screenshots and log files.
- Move from polling-based notifications to something like WebSockets for real-time delivery.

VIII. CONCLUSION

This project set out to address a problem that is easy to describe but genuinely inconvenient to live with the same bug getting reported more than once because nothing in a conventional tracker helps a tester realize it was already filed under different words. The system built here does not solve this perfectly, and this paper has tried to be honest about where its results are strong, where they are limited by the small scale of testing that

was realistically achievable, and where features originally planned were deliberately deferred rather than rushed. The direct semantic search example discussed there, is that combining a classical technique like TF-IDF with a modern sentence-embedding approach genuinely helps recognize bugs that describe the same underlying problem in different words — which is the specific gap that motivated this project in the first place. The system was built not as an isolated model but as a working application with real authentication, a real database, and a real interface, because the intention from the start was to produce something that reflects how this kind of tool would actually need to function for a tester or developer to use it day to day.

REFERENCES

- [1]. P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of Duplicate Defect Reports Using Natural Language Processing," in Proceedings of the 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 2007.
- [2]. X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An Approach to Detecting Duplicate Bug Reports Using Natural Language and Execution Information," in Proceedings of the 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany, 2008, pp. 461–470.
- [3]. C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Duplicate Bug Report Detection with a Combination of Information Retrieval and Topic Modeling," in Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012), Essen, Germany, 2012.
- [4]. A. Sureka and P. Jalote, "Detecting Duplicate Bug Report Using Character N-Gram-Based Features," in Proceedings of the 2010 Asia Pacific Software Engineering Conference (APSEC), 2010.
- [5]. N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," in Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), Hong Kong, China, 2019, pp. 3982–3992.
- [6]. F. Pedregosa et al., "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
- [7]. A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the Severity of a Reported Bug," in Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), Cape Town, South Africa, 2010.