An Analysis of Improving Bug Fixing in Software Development

Daniel Caliman[1], Valentina David[1] and Alexandra Băicoianu[2][a]

[1]*Siemens Industry Software, Braşov, Romania*
[2]*Faculty of Mathematics and Informatics, Transilvania University of Braşov 50, Iuliu Maniu, 500090 Braşov, Romania*

Keywords: Software Defects Context, Bug Report, Word Embedding, Text Similarity, Encoder, Summarization.

Abstract: When a defect arises during a software development process, architects and programmers spend significant time trying to identify whether any similar defects were identified during past assignments. To efficiently address a software issue, the developer must understand the context within which a software defect is reproducible and how it manifests itself. Another important aspect is how many other issues related to the same functionality were reported in the past and how they were solved. The current approach suggests using unsupervised machine learning models for natural language processing to identify past defects similar to the textual content of the newly reported defects. One of this study's main benefits is ensuring a valuable knowledge transfer process that reduces the average time spent on bug fixing and better task distribution across team members. The innovative aspect of this research is gaining an increased ability to automate specific steps required for solving software reports.

1 INTRODUCTION

Efficient bug fixing is a challenge, especially in the context of big projects. Identifying the appropriate resolver and identifying similar defects from the past can be quite difficult, while inefficient bug fixing leads to an increased time between the transition of a bug from one state to another (open, verified, closed, etc.), thereby reducing the overall bug fixing time.

The bug solving process could be streamlined if there was an application to check whether the problem faced is similar to one from the past and to investigate how it was resolved. By means of this solution, the developer receives different suggestions on what may be the proper solution for the matter at hand. The question of identifying the bug reports that refer to the same source problem is a demanding task in the software-engineering life cycle, and scientists have proposed specific methods on information-retrieval techniques (Yang et al., 2016), (Nguyen et al., 2012), (Saha et al., 2013). Because of the importance of this issue, multiple studies have been actively carried out to propose different solutions to this bug reporting software process (Jalbert and Weimer, 2008), (Ahmed et al., 2021), (Wang et al., 2022), (Nazar et al., 2016). In addition, significant work and an older view of the state of the art in search-based software engineering are described in the reference (Harman et al., 2012).

The experiments presented in this paper propose a new solution that would allow identifying other bug reports related to a given subsystem or functionality, which were also reported in the past, starting from a list of keywords or a bug identifier. Regarding the proposed solution from this research paper, it is handling the similarities between two or more bugs by using a natural language processing (NLP) approach that can predict the semantic relationship between two documents/files/problem reports (PRs)—deciding whether specific software problems are related or not can be achieved by using a text similarity algorithm.

The average developer spends approximately 30% of the working time debugging errors, costing the global software industry billions annually (Britton et al., 2013). Therefore, finding and fixing code problems/bugs faster, more predictable, constructive, and dynamic is crucial for developers and software development managers.

There are several principal objectives that this study seeks to attain, including reducing the time spent on solving bugs and increasing the time allocated for developing new features. In addition, one more important aspect is reducing the time spent on calls or pair programming with other developers and providing a summary of the solution areas for already solved problems. The presented solution offers the

[a] https://orcid.org/0000-0002-1264-3404

Caliman, D., David, V. and BĂČicoianu, A.
An Analysis of Improving Bug Fixing in Software Development.
DOI: 10.5220/0012119500003538
In *Proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023)*, pages 470-477
ISBN: 978-989-758-665-1; ISSN: 2184-2833
Copyright © 2023 by SCITEPRESS – Science and Technology Publications, Lda. Under CC license (CC BY-NC-ND 4.0)

possibility to display the modified files that have already been solved in the case of similar bugs.

2 SEMANTIC TEXTUAL SIMILARITY AND SUMMARIZATION

Information extraction is the process of recognizing key phrases and structured knowledge in the source text by seeking predefined sequences in the analyzed text with pattern matching and also NLP (Singh, 2018) approaches.

2.1 The Performance of the Text Similarity Algorithms

Similarity algorithms measure how similar two text descriptions are. To calculate text similarity, it is necessary to transform the text into a vector of content-specific features. A measure of similarity is a data mining process that accounts for the distance between the vectors of features. If the distance is small, the characteristics have high similarity. Similarity functions are used to measure the distance between two vectors.

Within the software solution, we have chosen to introduce several methods of similarity measurement considering the following aspects:

- By applying several methods to the same dataset, it can be established which algorithm is more efficient;
- In certain cases, the time difference between the algorithms can be quite significant, an aspect that can be taken into account when wanting to make a calculation;
- With a large data set, consulting the results by using another algorithm may result in a higher chance for the user to find the right solution.

For the similarity comparison between two descriptions, the text is introduced into the similarity algorithm, which provides the distance, and the greater the distance, the fewer the similarity features. For obtaining a useful metric that measures the similarity, not the dissimilarity, the distance is first normalized, following which the results will be between [0,1], where 0 means not similar, and 1 means identical descriptions. A wide variety of quality metrics can be used (Vidal, 2021), but in this test case, three types of similarity algorithms are involved: Cosine, Euclidean (Elmore and Richman,), and Jaccard (Chung et al., 2019).

2.2 Text Summarization Approaches

The similarity algorithms offer the most similar top results that exceed a certain minimum similarity threshold. If there are few results for the user, it is easy to check them thoroughly, but when the algorithm offers more than 10 results, the process becomes more time inefficient. That is why, within the application, we have also implemented a summarization feature that can extract only the important parts from the solutions offered.

Summarization is a technique by which a text is truncated by extracting only important points in the document. The extractive based summary suggests retrieving the dominant phrases and lines in the analyzed documents. Then all the decisive lines are combined to create the summary. In this case, every line and word in the summary is part of the document's original text being analyzed and summarised. We have chosen to use extractive summarization algorithms because the sentences the developers offer as the right solution can have a completely different meaning if the same words are not used.

We used the library *Sumy*, a Python library for extracting the summary from HTML pages or plain texts to implement the summarization algorithms. We chose four algorithms for text summarizing: *Luhn*, *LexicalRank (LexRank)*, *Latent Semantic Analysis (LSA)*, and *Text Rank*.

Luhn: Luhn's algorithm is based on TF-IDF (Term Frequency - Inverse Document Frequency) and it selects words of greater significance according to their frequency. Importance is attributed to the words present at the beginning of the text and the algorithm considers the most important words (Luhn, 1958). Notice that the experiments demonstrate that Luhn's algorithm provides satisfactory results when: words that do not appear frequently are not necessary, words that appear very often are not important (e.g. "is", "and") and the input is a technical document. All three approaches are applicable to this study.

LexRank: LexRank is an algorithm for summarization based on unsupervised graphs (Erkan and Radev, 2004). It collects only the words of greater significance depending on their frequency. LexRank is used to calculate the importance of sentences based on the concept of centrality of the eigenvector in a representation graphics of sentences where the sentences are placed at the tops of the graphs and importantly, propositions are calculated using cosine similarity procedure. Moreover, it is worth mentioning that the LexRank algorithm provides satisfactory results when the sentence considered important is similar to other sentences in the text.

LSA: LSA is a NLP technique to analyze the common points between an input and the terms contained in that input.

TextRank: TextRank is a text summarization technique that uses the PageRank algorithm (Bianchini et al., 2005). PageRank is mainly used for ranking web pages in online search results. It ranks web pages according to the number of references a web page has, web pages with more links to it are considered more important. TextRank is based on the same representation as PageRank, with the following remarks:

- Instead of web pages, we use sentences;
- The similarity between two sentences is used as equivalent to the probability of referring to the web page.

The TextRank algorithm provides satisfactory results when the sentence considered important is referenced as often as possible in the document.

3 PREREQUISITES

3.1 Baseline Systems

Our research is based on the output of more than 100 software developers within a software company that are using *TAC* and *Jira*.

TAC: *TAC* is an internal tool used for bug tracking and management. For this paperwork, we used a data set used within a software company. This data set contains the history of solved bugs within a development team. The data set is stored in a database, but for easier handling and processing it has been exported in a CSV file. *TAC* is an existing system being used in software development to report, track and resolve PRs, and Enhancement Requests (ER), arising from the software company products. When a software defect is found in a product by an external customer or by internal users of the product, a PR is logged in the *TAC* system. Internal users or developers at remote sites, without access to the company network, but with access to the Internet, can log PRs using the *TAC* system.

Jira: In large companies, the bug fixing effort must be tracked to establish maintenance costs and provide better estimations. For our study, the dataset extracted from *Jira* brings additional details related to the time spent on most PRs. Jira dataset is stored in a MongoDB database due to the fact that the information describing a bug can be stored as a single entity, and MongoDB provides simple document storage facilities. In our case, this is especially useful for string manipulation steps, the pre-processing operations applied on the entire input.

There is a general lack of efficiency considering the data processing of both mentioned systems, *TAC* and *Jira*. Our study finds the following root causes:

- Duplicates of newly created PRs.
- Incorrect assignments of PRs to appropriate persons to fix them in 20% of the cases;
- Manual process to assign PRs instead of (semi) automatic one;
- The absence of any relevant content processing of existing PRs in order to optimize the creation of new PRs;
- Lack of enforcement for a uniform style for the PRs creation.

The *TAC* and *Jira* databases constitute the main data sources for this research proposal. Even the fact that two tools overlap in the bug fixing matter affects the process efficiency. Therefore, another outcome of this research is to offer guidelines related to the two databases' structural and logical integration on just one unique PRs management tool. The content of the data sources is written in English to have a common understanding among all developers working for the same company from different countries. Also, the information present in both data sets refers to the same project. *TAC* dataset is bigger than the one from *Jira* because *Jira* was recently adopted as the main time tracking tool.

3.2 Datasets

Two different datasets were used for this research study, one generated from *TAC* and another one from *Jira*. While different in appearance, each dataset has the same summary features and characteristics. Both can help understand how a PR should be handled. *Jira* dataset completes the one from *TAC* with information about the time spent on a certain task.

Dataset1 - *TAC* - According to the *TAC* tool, a fixed bug is structured as follows:

- *PR Number*: represents the number of the PR by means of which we can find the bug in the database;
- *Short Description*: represents a succinct description of the problem that the user encounters;
- *PR Text*: represents a more detailed description of the existing problem. Here are specified details about the mode of reproduction of the bug, the environment used, and general characteristics;
- *Final Response*: represents the steps that the software developer followed to solve this problem;

- *Assigned Employee*: represents the software developer who was assigned to solve this problem.

The dataset has 9918 entries/rows (1.06 MB of data). From a total number of 14 columns, only 5 columns were used due to the fact that they contained the most relevant information for our study.

Dataset2 - *Jira* - A dataset containing 8068 *Jira* records (352KB) was used for this paperwork. In general, one can retrieve information from different types of *Jira* issues, like epics, stories or tasks. Still, for this paper, we only used the task issue since they give the most accurate details in terms of time spent on a certain problem and the exact name(s) of the staff members involved. Furthermore, even though in *Jira* the information is stored and organized in more than 17 columns, we use only 4 of them, namely those containing PR workloads.

The following attributes describe each record:

- *Title*: The task title from *Jira* needs to be at least the same as the “Short description” field from *TAC*, but in most cases, it is a combination of the “PR number” and the “Short description” fields.
- *Description*: This field usually contains the same information as the “PR Text”.
- *Assignee*: The assignee of a task is the person that fixed the bug that the task was created for.
- *Time Spent*: the amount of time needed to complete the task.

4 RESULTS

4.1 Pre-Processing the Input Data

The data pre-processing phase aims to turn the raw data into a more understandable, useful, and efficient format for machine learning models. Hence, before applying any information retrieval algorithm, the dataset must go through a pre-processing step, which implies that the information is cleaned and ready to be used. For any NLP problem, one would choose to perform at least one of the following operations to ensure the quality of the dataset:

1. Convert all the words to lowercase - this guarantees that a word written in lowercase, uppercase, or a combination of both is treated as a single term;
2. Remove punctuation - punctuation marks are treated as noise in this case since only the words are relevant in the computation phase;
3. Remove stop words - stop words can also be considered noise. Usually, they represent a collection of prepositions or verbs that do not really add value to a phrase or document.
4. Spell checking - this phase ensures that the words are correctly written in order to maintain their true meaning;
5. Lemmatization - a procedure used for removing suffixes or prefixes from a word and also for grouping different inflected forms of a term in order to be treated as the same item.

All the steps mentioned above have been applied to both datasets used for the experiments in this paper such that we obtain more valuable input for the similarity and summarization algorithms. For our case, the pre-processing operations have been applied in the same order as detailed in this section. However, depending on the problem one is trying to solve, one or more steps can be skipped or applied in a different order.

A real PR text, labeled $T1$, is converted into $T5$, after performing all the operations mentioned above, as follows:

$T1$: “*The request to free a license (curfew) does not work when using a license server Run a local license server. Use from out desktop this local license server. Define a curfew and restart desktop. Wait until the curfew passes... nothing happens. However, point to a license file and do the same. Here the curfew message is shown while it should not!*”

$T5$: “*request free license curfew work using license server run local license server use desktop local license server define curfew restart desktop wait curfew passes nothing happens however point license file curfew message shown*”

The output of $T4$ and $T5$ is the same because those steps have no effect on the selected example. We have collected some harmless examples without using sensitive business information from our client, but each of these steps is an essential step in pre-processing phase.

4.2 The Input Data Processing Cycle

In this section, we present our approach for the launched problem, the architecture, and the main features built to facilitate the analysis and resolution of bugs by providing similar bugs, which could lead to solving the problem developers are facing. The purpose of the solution is to find and analyze similar bugs that have already been solved in the past, and the solutions they propose may also be useful in other cases.

In order to solve these features, but also to easily and quickly test the results of this proposal, a *Bug*

Fix Suggester (BFS) web framework application was developed. Granting a bug similar to the one that we're encountering is one of the most significant features that *BFS* provides for users. It is through these searches that the submitters find similar bugs in order to get a useful starting point to solve their bug. The architecture of the implemented framework consists of a database component, a core module, and a Web interface unit. Figure 1 displays a simplified structure of the application.

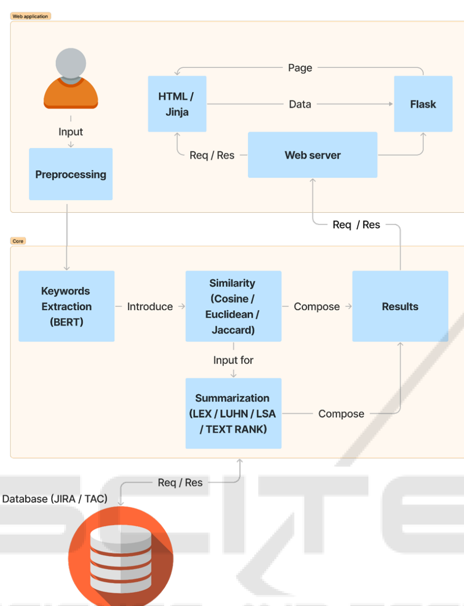

Figure 1: *BFS* Architecture.

The database stores in an organized way - see the pre-processing step - the bug reports from the bug repositories to facilitate further searches and perspectives. Correct, consistent, and usable input datasets are mandatory to facilitate more accurate further processing and analysis.

The Web module implements the user interaction features through a Web Browser. The easiest way to provide fast and easy-to-interpret results is by implementing a web application where the user can enter the input and see the results. To perform such a purpose, we chose the following instruments and technologies: HTML, Jinja, JavaScript, CSS for the interface, and Flask for the Web application.

The main unit (core) has sub-units for text pre-processing and keyword extraction, implementation of similarity/summarization algorithms, and delivering results. *BFS* Core module contains the main components of the tool, which are responsible for running the pre-processing, keyword extraction, and similarity/summarization algorithms.

Keyword extraction is the automated process of extracting the words and phrases that are most relevant to an input text. For our study, we have built a basic keyword extraction pipeline that can identify and return noticeable keywords from the original text. For our research, the semantics of the words and the words themselves are imperative due to our engineering objective.

The embedding process uses an unsupervised sentence transformer autoencoder, such as Bidirectional Encoder Representations from Transformers (BERT) (Devlin and Chang, 2018). The training uses a new approach called Masked Language Modeling (MLM), which allows bidirectional training of the transformer in models. Usually, a transformer is meant to learn contextual relationships between words in a document, following two techniques. The first is represented by an encoder that reads the input, and the second is identified as a decoder that outputs predictions for a given task. However, BERT's primary goal is to produce a language model, so only the encoder part of the transformer is used by this algorithm. The input for BERT is represented using three different embeddings: token, segment, and position, while the pre-training step is made of two components, the MLM and the Next Sentence Prediction (NSP) model, as follows:

1. *MLM*: Before feeding the model, a small amount of the words from the input dataset are replaced with a mask token. Then, the model tries to predict the original value of the masked terms using the context given by the non-masked version of the words. The prediction is made in three steps:
   (a) A classification layer is needed on top of the encoder outcome;
   (b) The output vectors are multiplied by the embedding matrix and then transformed into a vocabulary dimension;
   (c) The probability of each term in the vocabulary is computed using SoftMax.
2. *NSP*: The input for NSP is structured in pairs of sentences because the model tries to predict if the second sentence in the pair is the sentence that follows after the first one in the original phrase. Half of the dataset is built up using this rule, while the other half comprises pairs in which the second sentence is picked randomly from the given corpus. In the latter case, it is assumed that the two sentences will be disconnected. Before entering the model, the input goes to a pre-processing phase, built up from three steps, as follows:
   (a) A "CLS" token is inserted in the first position from the first sentence, and a "SEP" token is

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

inserted in the last position of the second sentence;

(b) Each token receives a sentence embedding - used for labeling one of the two sentences in the pair;

(c) In a sentence, the token position is identified using a positional embedding that is added to that token.

The results obtained through the similarity algorithm are bugs from history that a software developer resolved. The multitude of solutions provided by the similarity algorithm also helps when after consulting the results, the user still cannot find a solution for resolving the bug. When the software developer needs to find out who is most advised to help with the problem at hand, a list with the number of occurrences in the final results of the algorithms is drawn up, and these results are displayed to the user. This list of software developers is formulated using the results provided by the similarity algorithm. After the list of bugs similar to the description given as input is obtained, the software developer further examines who solved these specific bugs and creates a new list of software developers who hold the necessary knowledge to help solve the problem encountered. A software developer who has solved several bugs will be displayed higher on the list.

5 DISCUSSIONS AND RESULTS

In order to visually analyze the numerical results provided by the three similarity algorithms, the similarity scores offered by each of them were plotted. The Cosine similarity algorithm provides the most evenly distributed results in the interval [0, 1], whereas the Euclidean similarity algorithm is focused on point 0.4 but provides uniformly distributed results in the interval [0.5, 1]. Furthermore, the Jaccard similarity algorithm provides uniformly distributed results in the interval [0, 0.5]. While all the algorithms yielded satisfactory results, it is evident that Cosine provided all outcomes equally likely, see Figure 2. Also, in the last one, it is to notice the distribution of all the points for each of the chosen algorithms.

For each similarity algorithm, a set of different tests was done so that an optional threshold could be selected. The selected optimal threshold varies under different test criteria, so the optimal empirical threshold used had a value of 0.2.

To conclude the effect of these algorithms on the data sets considered, an investigation was carried out to enable us to draw conclusions about the behavior

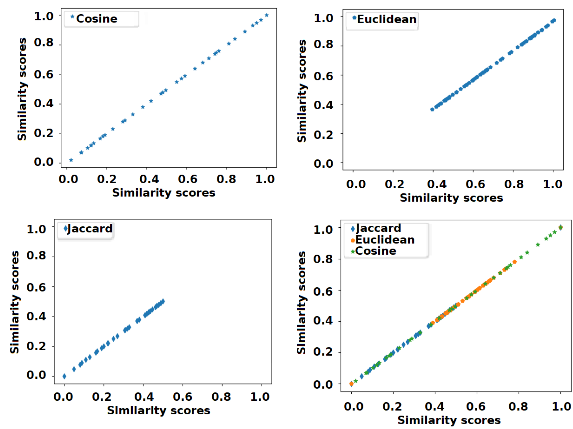

Figure 2: Performance for the similarity algorithms.

of these algorithms. The approach was implemented by following two steps:

1. *First Step*: Accessing the PRs that are marked as being duplicated from the database and analyzing the result of the similarity algorithm, namely the Cosine similarity algorithm;
2. *Second Step*: For each bug present in the database 10 paraphrased descriptions were analyzed where the most similar result is the original bug. The Pegasus paraphrase model was adopted.

First Step: For the duplicate analysis, those PRs marked as duplicated were extracted from the database and observed by considering the field (Short description) containing the duplicate PR. Overall, 14 bugs have been cataloged as duplicate PRs. In order to demonstrate the effectiveness of the Cosine similarity algorithm, the PRs duplicates from the database were analyzed and then entered into the *BFS* software solution. In Figure 3 it can be noticed that in 78% of cases, the algorithm displayed the duplicate PR among the results. In Figure 3 it can be noticed that when we introduced the data, the algorithm resulted that in 78% of the cases, the text given was from a duplicated PR.

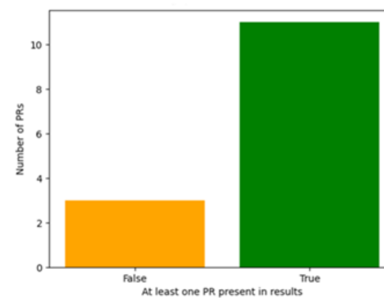

Figure 3: Duplicate PRs results.

Second Step: In order to test the Cosine similarity algorithm on several examples, 1200 bugs were con-

sidered, and for each bug, a paraphrasing approach was applied, which would generate 10 slightly different descriptions of the bug to see if the similarity algorithm would give the same bug as the most similar.

In general, transformers are semi-supervised machine learning models mainly used with text data and have replaced recurrent neural networks in NLP tasks (Vaswani et al., 2017). Transformers are designed to work with sequence data by using an input sequence to generate an output sequence (Chirkova and Troshin, 2021). One transformer has two main components: an encoder that primarily operates on the input sequence and a decoder that operates on the target output sequence during training and predicts the next element. The Pegasus model is pre-trained similarly to a summarization algorithm, where important sentences or words are extracted from an input document and merged to provide the output. The way the

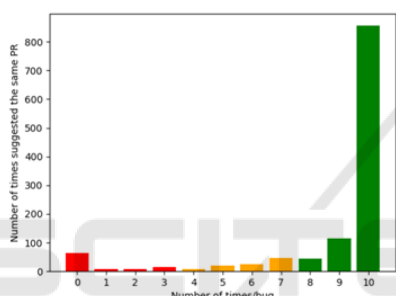

Figure 4: Results for PRs - Pegasus transformer.

BFS application works is the same, with the distinction that the input is not the one entered by the user but the 10 paraphrased sentences. Notice in Figure 4 how out of 1200 bugs, in 71% of the cases, the algorithm resulted in the bug that we paraphrased in all 10 iterations. As a result, 60 of the 1200 bugs have never managed to provide the paraphrased bug. In a more detailed examination, we observed that this happens because the paraphrasing algorithm excludes the differentiating word from the sentence, changing the meaning of the sentence completely.

We considered a study test example, and the correctness of the result, its relevance, and the execution time of the algorithm were observed. The interval [0,1] includes score values of similarity, where 0 represents a description, not at all similar, and 1 illustrates an identical description.

All the records in *TAC* and *Jira* are used in the same project, which implies that the datasets are relevant for the analysis. However, it is noteworthy that *TAC* has more records than *Jira*, which is attributed to the fact that *TAC* has been in use for a longer period of time. It is important to mention that the proprietary nature of the datasets prohibits the inclusion of a large number of records that do not include sensitive data, thereby limiting the scope of the analysis. A general bug description was considered input because the real used database is the client's property.

The input text is: "*The request to free a license (curfew) does not work when using a license server Run a local license server. Use from our desktop this local license server. Define a curfew and restart desktop. Wait until the curfew passes... nothing happens. However, point to a license file and do the same. Here the curfew message is shown while it should not!*" and we will introduce it into our *BFS* software solution to see if the results given would help us resolve the issue.

The best four results are:

1. *License curfew does not save the project when the project was not saved before* With the solution: *Synchronous saving forced in curfew dialog before sending to the new version of the application the message to kill the process. Tested on a previous release version and curfew does not work. Application does nothing. Fixed* - Cosine similarity is 0.4921.

2. *Reconnect to the license server does not work after network loss.* With the solution: *An extension to heartbeat with reconnect option has been introduced.* - Cosine similarity is 0.3746.

3. *Application crashes when connection to license server is lost* With the solution: *I tested on version 230317and I see in case of license lost that the dialog appears in the home page of our frontend application. So it seems to work fine. I could not test the scenario where I have a hardware connection active. I tried with the application connected to the hardware and the client open in the same time ... I cannot reproduce. What frontend did you used?* - Cosine similarity is 0.2991.

4. *Reconnect to license server does not work* With the solution: *Implement multiple tryouts to reconnect to server when user presses Reconnect. It gives more chance to establish the connection again. (it is not guaranteed to work from the first attempt).* - Cosine similarity is 0.19.

After analyzing the results mentioned above and their Cosine score, the first two suggestions are more likely to be used to solve the initial bug since they have the highest similarity scores. Also, the third result, which has a lower score, does not offer a solution since it is mentioned that the issue could not be reproduced, while the latest suggestion does not provide any helpful information - also, the similarity score for this one suggests that it is the worst-case scenario that one should consider since it is fuzzy and time consuming. The proposed solution should be straightforward, not based on tryouts or guessing.

6 CONCLUSIONS

Software defects must be tackled quickly and proactively because the quality and efficiency of any software product are essential features for it to survive on the market.

One of the main advantages of using the proposed solution is that it can work with any dataset from any industry as long as the bug related data is organized in a particular format - similar to the one mentioned in our datasets. In addition, this approach gathers details associated with a specific application component in a single place. It was developed to find similarities between two or more software bugs that were reported on a particular system so that the data that is being displayed refers to the same feature or category of features. Also, most of the time in the development process is spent on discovering why a bug is present in the software, how the code base needs to be changed to fix it, and what other similar issues have been reported in the past. All these answers can be easily formulated by analyzing the data shown in the web application module of the presented solution.

The solution presented can be improved and further enriched with multiple features so users can benefit even more when using it in their day-to-day activities. For example, in this research, we used a pre-trained model for the BERT algorithm. However, using an industry-specific vocabulary could benefit even more from the accuracy of BERT because the more he learns what different terms mean, the better the prediction will be. Furthermore, the product requirements represent another source of relevant information on a specific topic. For the case analyzed in this paper, the requirements reside inside epic contracts and epic analysis documents. Integrating this information into the dataset used for prediction will enlarge the possibility of finding the root cause of a bug.

REFERENCES

Ahmed, H. A., Bawany, N. Z., and Shamsi, J. A. (2021). Capbug-a framework for automatic bug categorization and prioritization using nlp and machine learning algorithms. *IEEE Access*.

Bianchini, M., Gori, M., and Scarselli, F. (2005). Inside pagerank. *ACM Trans. Internet Technol.*

Britton, T., Jeng, L., Carver, G., Cheak, P., and Katzenellenbogen, T. (2013). Reversible Debugging Software: Quantify the time and cost saved using reversible debuggers.

Chirkova, N. and Troshin, S. (2021). Empirical study of transformers for source code. New York, NY, USA. Association for Computing Machinery.

Chung, N. C., Miasojedow, B., Startek, M. P., and Gambin, A. (2019). Jaccard/tanimoto similarity test and estimation methods for biological presence-absence data. *BMC Bioinformatics*.

Devlin, J. and Chang, M.-W. (2018). Open Sourcing BERT: State-of-the-Art Pre-training for Natural Language Processing.

Elmore, K. and Richman, M. Euclidean distance as a similarity metric for principal component analysis. *Monthly Weather Review - MON WEATHER REV*.

Erkan, G. and Radev, D. R. (2004). Lexrank: Graph-based lexical centrality as salience in text summarization. *J. Artif. Int. Res.*

Harman, M., Mansouri, S. A., and Zhang, Y. (2012). Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*.

Jalbert, N. and Weimer, W. (2008). Automated duplicate detection for bug tracking systems. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*.

Luhn, H. P. (1958). The automatic creation of literature abstracts. *IBM Journal of Research and Development*.

Nazar, N., Hu, Y., and He, J. (2016). Summarizing software artifacts: A literature review. *Journal of Computer Science and Technology*.

Nguyen, A. T., Nguyen, T. T., Nguyen, T. N., Lo, D., and Sun, C. (2012). Duplicate bug report detection with a combination of information retrieval and topic modeling.

Saha, R. K., Lease, M., Khurshid, S., and Perry, D. E. (2013). Improving bug localization using structured information retrieval.

Singh, S. (2018). Natural language processing for information extraction.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*. Curran Associates, Inc.

Vidal, F. (2021). Similarity Distances for Natural Language Processing.

Wang, Z., Tong, W., Li, P., Ye, G., Chen, H., Gong, X., and Tang, Z. (2022). Bugpre: an intelligent software version-to-version bug prediction system using graph convolutional neural networks. *Complex & Intelligent Systems*.

Yang, X., Lo, D., Xia, X., Bao, L., and Sun, J. (2016). Combining word embedding with information retrieval to recommend similar bug reports. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*.