Visually Explaining The Predictions Of Bug Localization

Parvez Mahbub
Department of Computer Science
Dalhousie University
Halifax, NS, Canada
parvezmrobin@dal.ca

***Abstract*—Fixing bugs is one of the crucial tasks of software development and maintenance. Reportedly, it spans 50% of the development time and consumes up to 80% of the budget. The prerequisite of fixing a bug is locating the bug, given a bug report. Therefore, bug localization has always been a major research interest in the software engineering domain. Over the last five decades, there have been numerous studies on localizing the bugs. However, the predictions of these bug localization techniques are still not practical, explainable, or actionable. As a result, these techniques are yet to make a noticeable impact in the software industry. Recent studies show that the lack of explainability is one of the core reasons behind the industry's lack of adoption of software engineering tools. In this paper, we propose a novel tool to visually explain the decision-making of a word embedding and information retrieval based *state-of-the-art* bug localization technique. We use Principal Component Analysis (PCA) and t-Distributed Stochastic Neighbor Embedding (t-SNE) to visualize the embedding space of bug reports and candidate bug locations. We also visualize the similarity calculation and decision-making regarding the bug localization. Our proposed techniques provide better understanding and valuable insight into the bug location to the developers, enabling them to understand the bug better and fix it faster.**

I. Introduction

Fixing bugs is one of the crucial tasks of software development and maintenance. According to several reports, it spans $\approx$50% of the development time [1, 2], consuming up to 80% of the total budget [3, 4], and costing billions of dollars each year [5, 6].

When an end-user reports a bug, a developer identifies the source code responsible for it, understands the problem, and fixes it. As a result, the whole bug fixing process depends on successful bug localization. Given the vast importance, over the last five decades, there have been numerous works in bug localization [7, 8, 9, 10]. However, often these techniques are implemented as a black box that the users (i.e. the software developers) do not understand. The lack of explainability results in a lack of trust, leading to industry-reluctance to adopt or deploy such tools [11].

In recent years, the explainability of artificial intelligence (AI) systems has become a serious concern. European Union's General Data Protection Regulation (GDPR) states that if a data-driven decision affects an individual or a group, then the decision must be explainable [12]. Since then, explainability has become a serious concern of the social science and artificial intelligence community [11, 13, 14, 15]. However, despite being a crucial domain of computer science research, software engineering is yet to onboard in the journey of explainability. Recently, Tantithamthavorn et al. [16] conducted a literature survey on defect prediction, a crucial research area of software engineering. They found that only 5% of the studies explain their prediction. According to Dam et al. [11] mere scoring metrics (e.g. accuracy, f1-score) are not sufficient to gain users' trust. Since the testing data are already known to the researchers, they can tune the model to work better on the testing data. Therefore, these scoring metrics cannot guarantee a system's success once deployed "in the wild". According to Ribeiro et al. [17] a user's trust is directly impacted by how much they can understand and predict the model's behaviour, as opposed to treating it as a black box.

Explanations of predictions by an AI system can be texts, audio, or visualization [17]. While the textual explanation is easier to implement using templated messages, visualizations can use human cognitive power and requires less effort. In this paper, we propose a novel tool to visualize the decision making of one of the *state-of-the-art* bug localization techniques – **LR+WE** proposed by Ye et al. [18]. This technique combines logistic regression and word embedding techniques to localize bugs. First, they compute the similarity between a bug report and candidate source codes using word embedding. Then, using these similarity scores, they produce a ranking of possible bug locations using a linear regression model. In this project, we will visually explain the decision-making of the word embedding component. To design and develop our visual explanation system, we will answer the following research questions.

- **RQ_1**: How can we visualize similarity calculations between bug report and candidate source codes?
- **RQ_2**: How can we visualize high dimensional word embedding in two-dimensional space that users can percept and understand?

The rest of this paper is organized as follows. Section II discusses a few essential concepts we will use in our work, including the bug localization technique of Ye et al. [18]. Section III describes our progress to produce a visual explanation for the bug localization. Section IV briefs about the existing works on bug localization and visualization of bug localization. Potential threats to the validity of our works are discussed in section V. Finally, section VI narrates the conclusion.

II. Background

A. Word Embedding

Word embedding is a distributed representation of words in a vector space model where semantically similar words appear close to each other [19]. An embedding function $E : \mathcal{X} \rightarrow \mathbb{R}^d$ takes an input X in the domain $\mathcal{X}$ and produces its vector representation in a d-dimensional vector space [20]. The vector is distributed in the sense that a single value in the vector does not convey any meaning; rather, the vector as a whole represents the semantics of the input word. Word embedding alleviates many limitations of other vector space models (VSM), such as sparse representation problem of one-hot encoding or vocabulary mismatch issue of TF-IDF.

Word embedding can be learned in an unsupervised fashion from a given text corpus. Word2vec technique proposed by Mikolov et al. [21] is one of the most widely used embedding generation techniques. Word2vec is implemented using a simple three-layer neural network. The input is a word or a set of words, and from there, the network learns to predict the surrounding words of that input. While doing so, the hidden layer learns an internal representation of the input words, which is essentially the representation of those words in a distributed vector space model (i.e. word embedding).

Word2vector first generates n-grams from the training corpus. In skip-gram, a word2vec model, the input is the middle word from the n-gram, and the model is trained to predict the rest of the words from the n-gram. In continuous bag of words (CBOW), another word2vec technique, the input and output are reversed; that is, the surrounding words are given as input, and the model learns to predict the middle word.

B. LR + WE

Ye et al. [18] proposed **LR+WE** bug localization technique that incorporates both a "learning to rank" and a "word embedding" approach. First, they created word embeddings from several Eclipse API references and guide repositories. Using 21,848 unique tokens from these repositories, they used the skip-gram model to generate the word embedding. Once they have the embedding representation for a bug report and candidate source codes, they use the following equations to compute the similarities for each pair.

First, they compute the similarity between two words, w_0 and w_1, as

$$sim(w_0, w_1) = cosine(w_0, w_1) = \frac{w_0^T w_1}{||w_0||\,||w_1||} \tag{1}$$

Then, they compute the similarity between a word w and a document T as the maximum similarity between w and any word in T.

$$sim(w, T) = \max_{w' \epsilon T} sim(w, w') \tag{2}$$

After that, the asymmetric similarity from a document T to another document S is defined as

$$sim(T \rightarrow S) = \frac{\sum\limits_{w \epsilon P(T \rightarrow S)} sim(w, S)}{|P(T \rightarrow S)|} \tag{3}$$

where $P(T \rightarrow S)$ is defined as

$$P(T \rightarrow S) = \{w \,\epsilon\, T \mid sim(w, S) \neq 0\} \tag{4}$$

Finally, the symmetric similarity is the summation of two asymmetric similarities.

$$sim(T, S) = sim(T \rightarrow S) + sim(S \rightarrow T) \tag{5}$$

In the end, a linear regression based learning to rank model takes the similarity scores along with some other information retrieval parameters (e.g. author, component) and produces a final ranking.

III. Methodology

A replication of the work of Ye et al. [18] prerequisites our explanation system. We replicate and wrap the bug localization system in a web app to provide necessary data on demand. Based on the data, we generate explanations at four levels – word to word, word to document, document to document asymmetrically, and document to document symmetrically.

A. Dataset

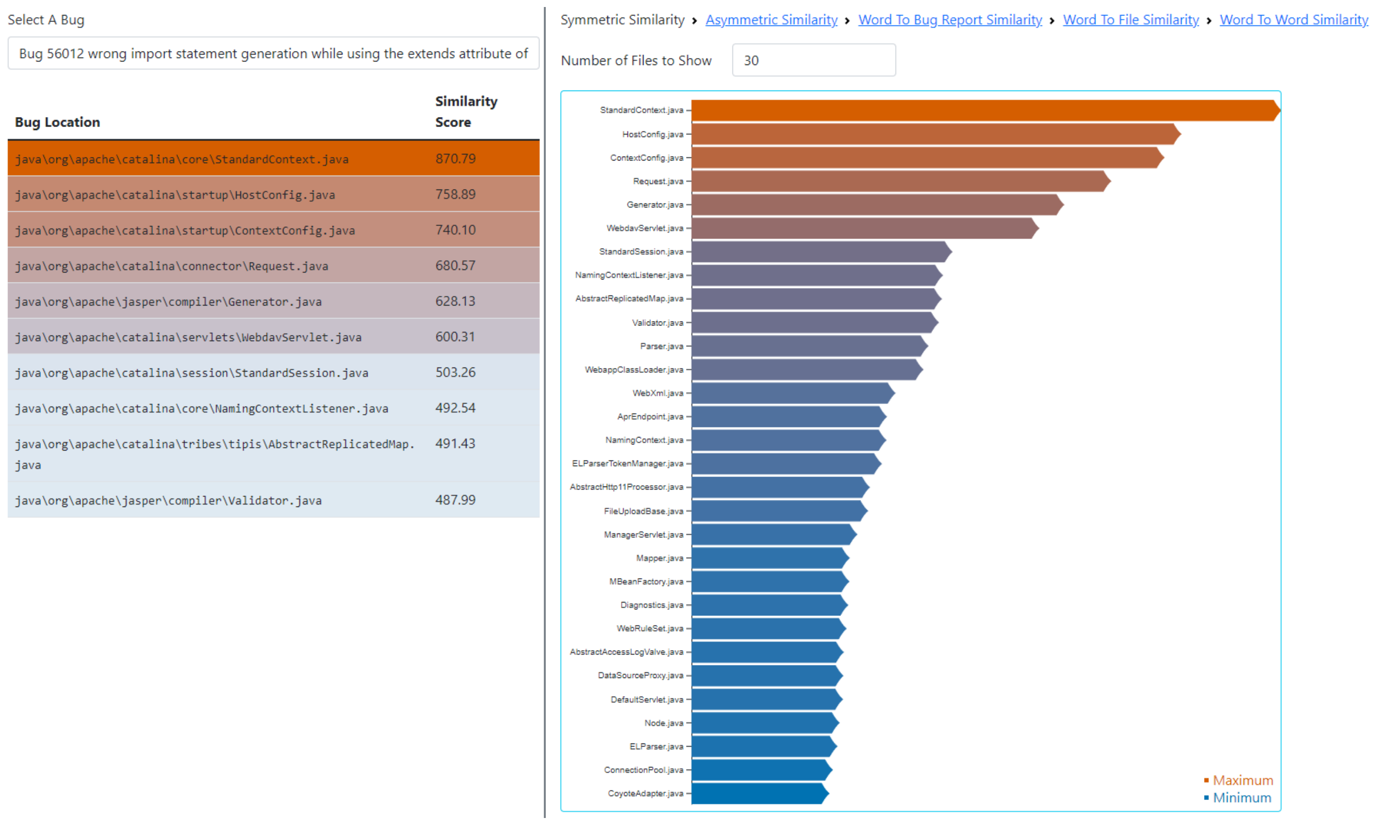
Ye et al. [22] created a benchmark dataset for the evaluation of bug localization tasks from six open source projects. From there, we only use the Apache Tomcat[1] project as our dataset. Apache Tomcat is an open-source Java HTTP web server developed by the Apache Foundation. It uses BugZilla[2] as its bug tracking system, and its bug reports are publicly accessible. Ye et al. [22] followed a heuristic proposed by Dallmeier and Zimmermann [23] to map the fixed bug reports to corresponding source files. The Tomcat segment of the dataset contains 1,056 bug reports and 1,668 source files, excluding the test scripts.

B. Web Server

We replicate the word embedding component of **LR+WE** bug localization system as described in the paper [18]. The authors demonstrate that the corpus used to train the word embedding does not significantly affect the bug localization. Therefore, instead of training a word embedding ground-up, we use a pre-trained word embedding named GloVe [24]. We use the 300-dimensional version of their embedding, which is trained on Wikipedia [25] and Gigaword [26] datasets. After the replication, we wrap it in a minimalistic web app using Flask Python framework [27]. It is a lightweight web framework and often provides research work as a service or replication package. The web app provides several application package interfaces (API) to provide word embeddings (at 2-dimension) and similarity information at different levels.

[1]http://tomcat.apache.org

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



Select A Bug

Bug 56012 wrong import statement generation while using the extends attribute of

Bug Location	Similarity Score
java\org\apache\catalina\core\StandardContext.java	870.79
java\org\apache\catalina\startup\HostConfig.java	758.89
java\org\apache\catalina\startup\ContextConfig.java	740.10
java\org\apache\catalina\connector\Request.java	680.57
java\org\apache\jasper\compiler\Generator.java	628.13
java\org\apache\catalina\servlets\WebdavServlet.java	600.31
java\org\apache\catalina\session\StandardSession.java	503.26
java\org\apache\catalina\core\NamingContextListener.java	492.54
java\org\apache\catalina\tribes\tipis\AbstractReplicatedMap.java	491.43
java\org\apache\jasper\compiler\Validator.java	487.99

Number of Files to Show 30

Fig. 1: The home screen[3]

C. *Client Server*

To visually explain the bug localization, we build a client server using TypeScript [28], Vue JS [29], and D3 [30]. Figure 1 is a screenshot of the home page of the client app. Selecting a bug report from the dropdown in the top-left corner fetches the necessary information from the webserver. Immediately below the dropdown, ten most likely bug locations are listed. These locations are colour-coded as per their similarity to the bug report. The user can investigate why a file is identified as a bug location at three levels – document to document similarity (high level), word to document similarity (medium level), and word to word similarity (low level).

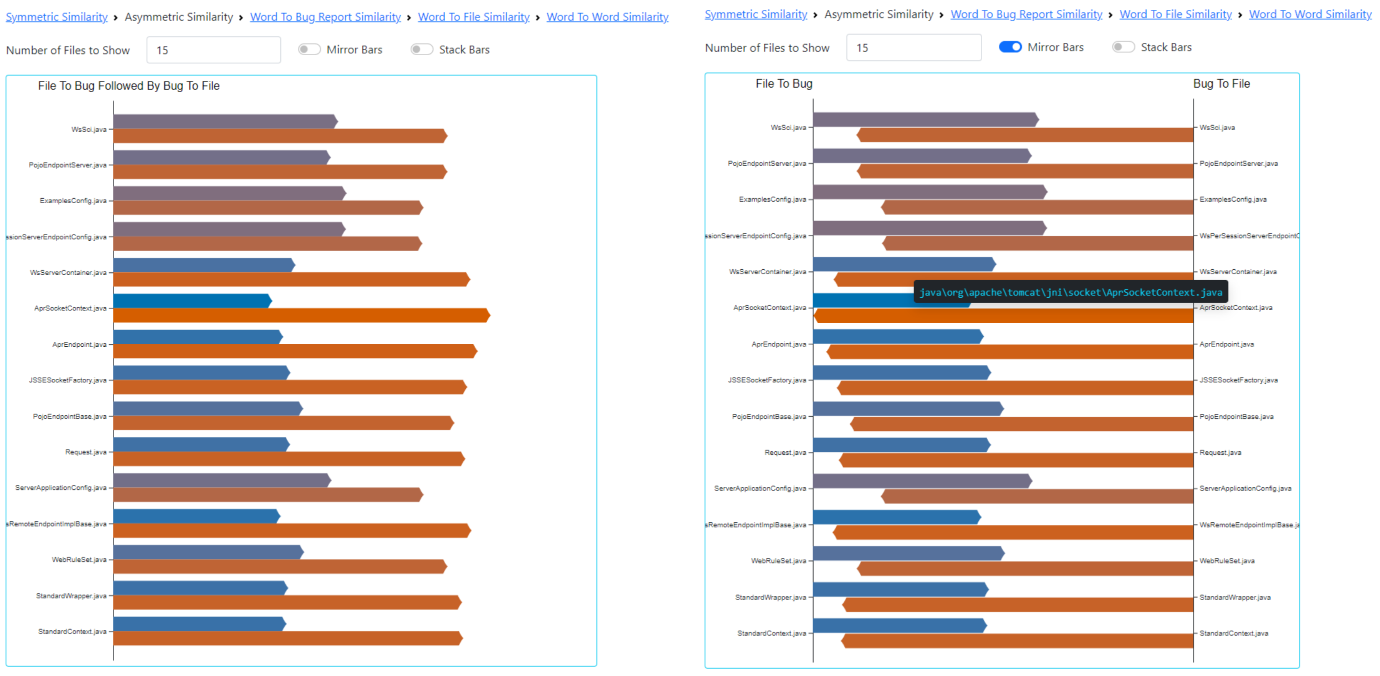
The app is available for public access[4]. Details of different visualizations are discussed in the following.

1) Visualizing Document To Document Similarity: The similarity between two documents can be computed symmetrically (equation 5) and asymmetrically (equation 3). Figure 1 shows the symmetric similarity between the selected bug report and top-30 candidate bug locations. The candidate bug locations are sorted by their similarity score in descending order. The bars associated with the bug locations are colour-coded, where orange indicates the highest probability and blue indicates the lowest.

[3]Better quality: https://parvezmrobin/ss/home.png
[4]http://bugloc.parvezmrobin.com

Figure 2 shows the visualization of asymmetric similarity between the bug report and source files. Similar to the former visualization, these bars are also colour-coded. Figure 2a illustrates two similarity scores for each file side by side, where the upper one is "from source file to bug report" similarity, and the latter one is "from bug report to source file" similarity. However, this visualization is quite a little cluttered, and there is often unused space on the right. To reduce the clutter and improve per-pixel information density, we introduce a mirror version of this visualization (figure 2b). The user can switch between them by clicking the 'Mirror Bars' switch. This visualization contains two different bar charts. The left chart has the similarity "from source files to bug report", and the right chart contains the similarity "from the bug report to source files". Figure 2a is better for comparing similarity scores for a single file (e.g. similarity scores for `WsSci.java` in both direction). On the contrary, figure 2b is better for comparing similarity scores in one direction (e.g. "from source file to bug report" similarities for all files). The interpretation of asymmetric similarity "from source file to bug report" is the average similarity between each word in the source file with the whole bug report. Conversely, the interpretation of asymmetric similarity "from bug report to source file" is the average similarity between each word in the bug report with the whole source file.

While the visualizations in figure 2 are well-equipped for comparing similarity scores, they cannot show how the


Symmetric Similarity › Asymmetric Similarity › Word To Bug Report Similarity › Word To File Similarity › Word To Word Similarity

Number of Files to Show 15 Mirror Bars Stack Bars

(a) Paired visualization

(b) Mirrored visualization

Fig. 2: Asymmetric similarity between bug report and source files[5]

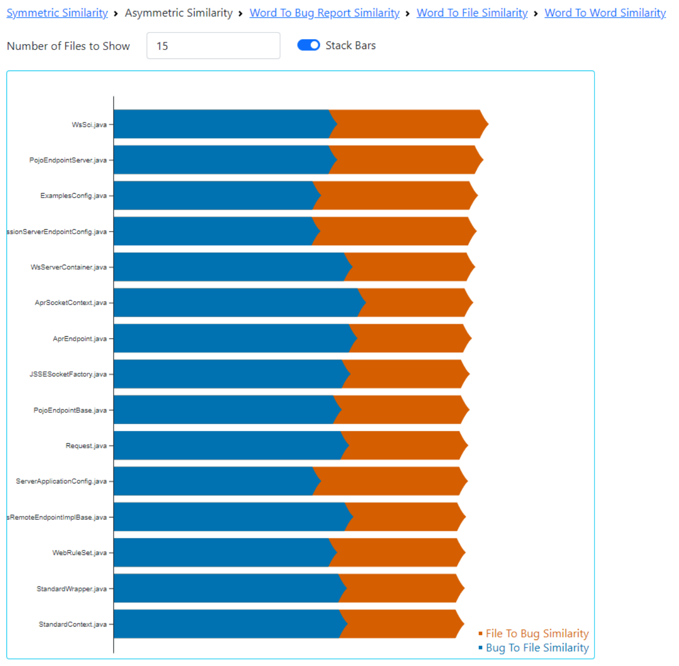
similarity in both directions contributes to forming the symmetric similarity. Therefore, introduce a third visualization for asymmetric similarity, namely – stacked asymmetric similarity (figure 3). This visualization shows the contribution of each asymmetric similarity to form the symmetric similarity. The user can click the 'Stack Bars' switch to turn stack visualization on or off.

By default, both symmetric and asymmetric similarity visualizations show only the top 30 similar bug locations. However, the user can increase or decrease the number of source files shown from the number input field on the top.

Neither the symmetric nor the asymmetric visualization contain the x-axis or any other way to see the similarity scores. We intentionally hide this detail from the user because the scores are just a means of comparing the relationship between different pairs of the bug report and source files. Other than that actual values of the scores do not convey any meaning. Therefore, we decide to hide the redundant information so that the user does not get overwhelmed.

We further hide the fully qualified names (FQN) of the Java classes (i.e. source files) from the visualization as they are often quite long. Instead, we only show the filenames by default. To see the fully qualified name for a particular file, the user can hover over a file name in both symmetric and asymmetric similarity visualizations (see figure 2b).

2) Visualizing Word To Document Similarity: While the former section illustrates visualizations with documents on each side, in this section, we break documents from one side into words. In other words, the visualizations in this section explain how different words from a document connect to another document and constitute the asymmetric similarity. The word-to-document similarity is computed twice – first, the similarity of words from source files with the bug report

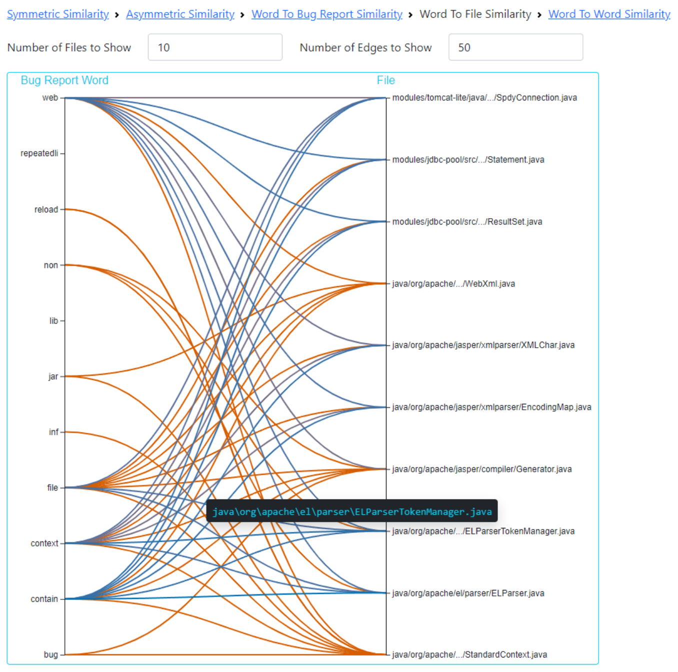

Symmetric Similarity › Asymmetric Similarity › Word To Bug Report Similarity › Word To File Similarity › Word To Word Similarity

Number of Files to Show 15 Stack Bars

Fig. 3: Stacked visualization of asymmetric similarity[6]

[5]Better quality: https://parvezmrobin.com/ss/asym.png
https://parvezmrobin.com/ss/asym-mirror.png

[6]Better quality: https://parvezmrobin.com/ss/asym-stacked.png



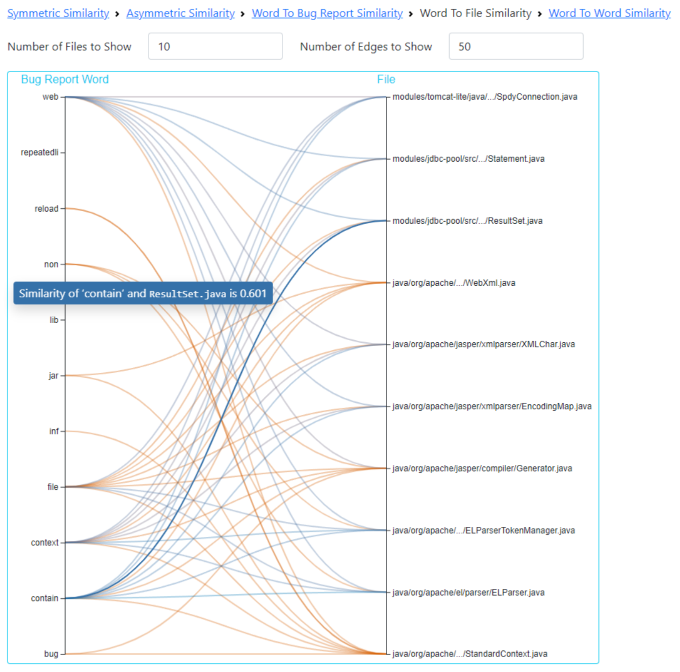
Symmetric Similarity › Asymmetric Similarity › Word To Bug Report Similarity › Word To File Similarity › Word To Word Similarity

Number of Files to Show: 10 Number of Edges to Show: 50

java\org\apache\el\parser\ELParserTokenManager.java

Bug Report Word
File
web
repeatedli
reload
non
lib
jar
inf
file
context
contain
bug
modules/tomcat-lite/java/.../SpdyConnection.java
modules/jdbc-pool/src/.../Statement.java
modules/jdbc-pool/src/.../ResultSet.java
java/org/apache/.../WebXml.java
java/org/apache/jasper/xmlparser/XMLChar.java
java/org/apache/jasper/xmlparser/EncodingMap.java
java/org/apache/jasper/compiler/Generator.java
java/org/apache/.../ELParserTokenManager.java
java/org/apache/el/parser/ELParser.java
java/org/apache/.../StandardContext.java

(a) General visualization



Symmetric Similarity › Asymmetric Similarity › Word To Bug Report Similarity › Word To File Similarity › Word To Word Similarity

Number of Files to Show: 10 Number of Edges to Show: 50

Similarity of 'contain' and ResultSet.java is 0.601

Bug Report Word
File
web
repeatedli
reload
non
lib
jar
inf
file
context
contain
bug
modules/tomcat-lite/java/.../SpdyConnection.java
modules/jdbc-pool/src/.../Statement.java
modules/jdbc-pool/src/.../ResultSet.java
java/org/apache/.../WebXml.java
java/org/apache/jasper/xmlparser/XMLChar.java
java/org/apache/jasper/xmlparser/EncodingMap.java
java/org/apache/jasper/compiler/Generator.java
java/org/apache/.../ELParserTokenManager.java
java/org/apache/el/parser/ELParser.java
java/org/apache/.../StandardContext.java

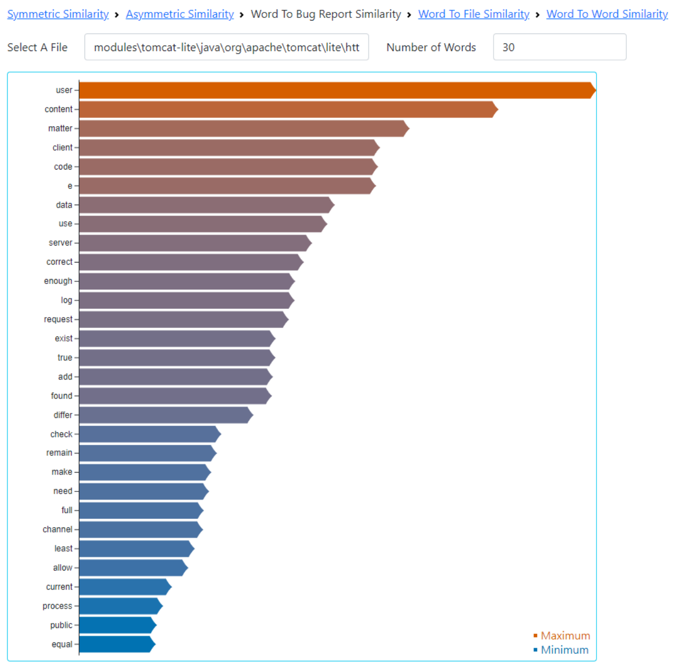
(b) Highlight hovered similarity

Fig. 4: Visualizing bug report words to file similarities[7]

and the similarity of words from bug report to source files.

Figure 4 shows a visualization of similarities between bug report words and different source code files (equation 2). We draw the similarities in the form of a bipartite graph where the bug report words are in one part and the file names are in the other. The user can limit the maximum number of edges (i.e. similarities) or file names to show. We show only the edges and source files with the maximum similarities. In the figure, some nodes on the left do not have any edges due to the limit on the number of edges. The more restrictive one will have an effect between the two limits (file and edge). For example, if the user limits to 10 files and 100 edges, and if 10 files can have at most 80 similarities, then only 80 edges will be shown. On the contrary, if the top 100 edges contain only 8 files, then only those files will appear. The edges are colour-coded as per their strength of similarity, where orange is the most similar and blue is the least similar.

With the growth of the number of edges, this visualization gets cluttered, and it becomes hard to understand the paths of different edges. To reduce the clutter, we bump the edges in the horizontal direction. Consequently, it becomes easier to understand the origin and destination of edges. However, the edges are still cluttered in the middle, and it is easy for the user to be lost. Furthermore, a human cannot perceive the continuous change of colour as good as size (e.g., bar length) [31]. Thenceforth, to get over the clutter and have an accurate perception, when a user hovers on an edge, we fade all other edges and show relevant information about the hovered edge. This information includes the origin and destination of the edge and the corresponding similarity score.



Symmetric Similarity › Asymmetric Similarity › Word To Bug Report Similarity › Word To File Similarity › Word To Word Similarity

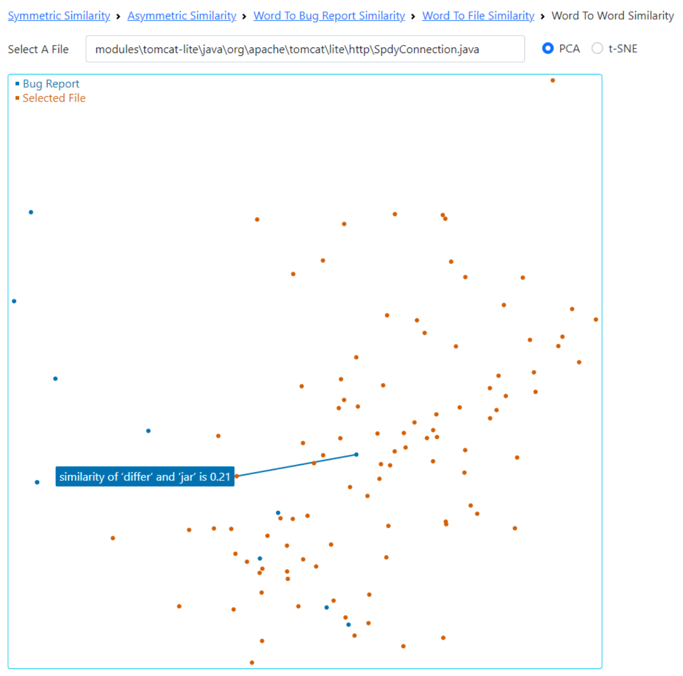
Select A File: modules\tomcat-lite\java\org\apache\tomcat\lite\htt Number of Words: 30

Fig. 5: Visualizing similarities of words of source files with bug report[8]

Figure 5 shows a visualization of similarities between the

[7]Better quality: https://parvezmrobin.com/ss/word2file.png
https://parvezmrobin.com/ss/word2file-hover.png

[8]Better quality: https://parvezmrobin.com/ss/word2bug.png

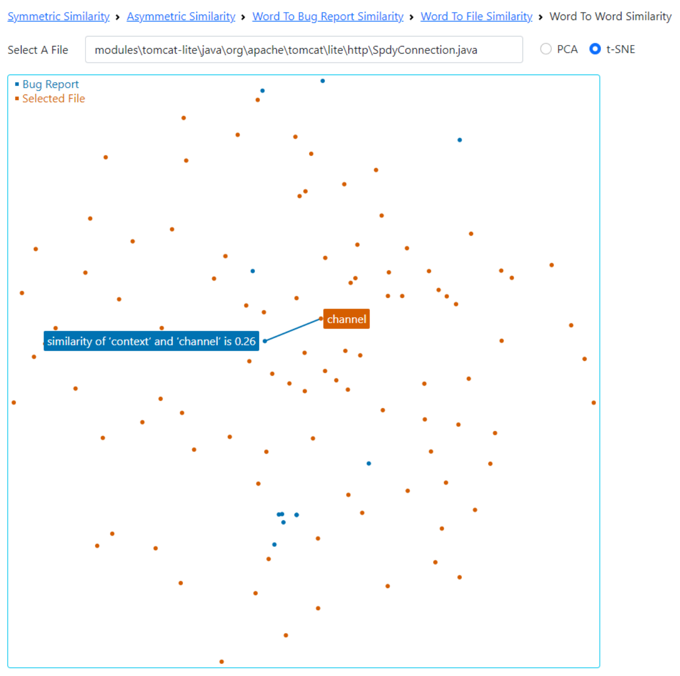

Symmetric Similarity › Asymmetric Similarity › Word To Bug Report Similarity › Word To File Similarity › Word To Word Similarity
▪ Bug Report
▪ Selected File

Select A File: modules\tomcat-lite\java\org\apache\tomcat\lite\http\SpdyConnection.java

[✓ PCA] t-SNE

similarity of 'differ' and 'jar' is 0.21

(a) A sample PCA visualization


Symmetric Similarity › Asymmetric Similarity › Word To Bug Report Similarity › Word To File Similarity › Word To Word Similarity
▪ Bug Report
▪ Selected File
channel

Select A File: modules\tomcat-lite\java\org\apache\tomcat\lite\http\SpdyConnection.java

PCA [✓ t-SNE]

similarity of 'context' and 'channel' is 0.26

(b) A sample t-SNE visualization

Fig. 6: Visualizing word to word similarities[9]

words of a source file and the whole bug report. The user can select a source file from the dropdown, and it will show the similarities of each word in the file with the bug report. Since, in this case, there is only one document (the bug report), instead of a bipartite graph, we implement this visualization as a bar-plot. The user can select a source file from the dropdown at the top, and similarities for each word in the source file will be shown. The user can increase or decrease the number of words to show. The bars are colour-coded as per the similarity strength. Similar to former bar charts, we do not show the similarity scores in this visualization.

3) Visualizing Word Embedding: Since word embedding is a high dimensional data (e.g. 300 in GloVe [24], 768 in BERT [32]), a human cannot visualize it directly. The use of dimensionality reduction techniques like principal component analysis (PCA) [33] and t-Distributed Stochastic Neighbor Embedding (t-SNE) [34] is common to visualize such data. To answer our **RQ$_2$**, we develop a visualization system that can render the word embeddings (reduced to 2-dimension) of both bug report and candidate bug locations in the same canvas.

While the visualizations in section III-C2 have words on one end and one or more documents on the other end, this visualization shows word-level similarities at both ends. Therefore when the user needs finer-grained information than a word to document similarity, the user can use this visualization to understand why a particular word has a particular similarity with a document. These visualizations further show the distributions of words from different documents over a shared canvas. In figure 6a, we see the user can select a particular file to see the word embeddings. In the canvas (marked by the lite blue square), blue dots are the words from the bug report, and orange dots are words from the selected file.

The user can hover on a dot to see the corresponding word. To see the similarity between two words, the user can click on one word and move the cursor to the other. Presumably, these two words should be from two different documents. Figure 6a illustrates an example case where the visualization shows the similarity between the words ‘statement’ and ‘case’.

Upon selecting the `t-SNE` radio button, the same words will appear using t-SNE (figure 6b). This visualization also provides functionality similar to the PCA visualization.

While PCA is purely a mathematical model, it is not always easy to perceive it (e.g. occlusion). On the contrary, t-SNE provides better visuals with less accurate information (e.g. two neighbouring nodes in t-SNE might not be neighbours in a higher dimension). Comparing figure 6a and 6b we can see it in effect. The PCA visualization leaves a lot of empty space over the canvas, but t-SNE makes full use of it. The user can selectively see information based on their preference.

IV. Related Works

In the following sections, we discuss bug localization techniques and visualization in bug localization.

A. Bug Localization

Over the last five decades, there have been numerous works in bug localization [7, 8, 9, 10]. These approaches use different techniques like information retrieval [35, 36, 37, 38, 22] and machine learning [39, 18, 40].

[9]Better quality: https://parvezmrobin.com/ss/pca.png
https://parvezmrobin.com/ss/tsne.png

Historically, information retrieval (IR) based approaches dominate in bug localization. Wang and Lo, in their *AmaLgam* method, utilized version history, similar reports, and the structure of the bug report to localize the bug [37]. *BugLocator*, proposed by Zhou et al., considers information about similar bugs that have been fixed before [38]. Even though both works achieved impressive improvement over the baseline, their evaluation was based on only four projects. Such evaluation imposes a threat to the generalizability of their outcome.

Moreno et al. proposed *Lobster* that incorporates the use of stack-trace to improve text retrieval based bug localization [35]. They computed the textual similarity between the bug report and code element and the structural similarity between the stack trace and code elements. Although they introduced a novel direction in IR-based bug localization and evaluated it with 14 different projects, the number of bug reports was only 155, questioning their reported result.

Saha et al. introduced BLUiR, which took the use of structured information inside the bug report one step further [36]. BLUiR can extract and utilize class and method names to provide more accurate bug location. Despite achieving a promising result, their evaluation is limited to four projects only. They also utilized an open-source information retrieval tool named Indri [41] that requires the search space (i.e. the source code repository) to be indexed beforehand. However, due to the versioned nature of modern software development, bugs are reported for different versions of the same software. It is technically infeasible to keep all active versions indexed and update the indices as the development progresses.

Ye et al. [22] used learning to rank to produce a ranked list of candidate bug locations. They extracted several relevant features (e.g., class name similarity, bug fixing recency, and frequency) and trained a support vector machine (SVM) to produce the ranked list. Later they enhanced this work [18] by incorporating word embedding.

Haiduc et al. [39] employ three frequency-based term weighting methods – Rocchio [42], RSV [43] and Dice [44], and deliver the best performing query keywords from the source code for concept location using machine learning.

Lam et al. [40] extracted several features (e.g., TF-IDF) from both bug reports and source files and projected them in an embedding space to produce a deep learning based relevancy estimation. After that, they concatenate this relevancy estimation with other metadata (e.g., severity, importance) to produce the final ranked list.

While these works march towards better metric values, they become more complex and lose understandability. As a result, they warrant an embedded explanation system that they severely lack.

B. Visualization in Bug Localization

Visualization of bug localization tools is still an emerging field. Therefore, the amount of work in this domain is not substantial.

Ruthruff et al. [45] explored the paradigm of end-user programming (e.g., spreadsheets) and found that these programs are often faulty. They proposed several visual techniques for the fault localization of these programs. They found that no single visualization is unarguably the best and provides the cost-effectiveness of each technique.

Risi and Scanniello [46] proposed MetricAttitude – a visualization tool based on static analysis that provides a metal picture by viewing an object-oriented software system employing polymetric views. It allows the user to formulate a textual query and to show a visual representation of the subject software with elements that are more similar to the query. This is primarily important when the user needs to identify or localize a feature implemented in the source code. Later, they enhanced this work in MetricAttitude++ [47] by incorporating an information retrieval engine.

Lahijany et al. [48] proposed IdentiBug – a model-driven visualization for bug localization. It takes advantage of deep learning techniques to train a bug localization model to predict the connection between bug reports and the system's class diagram. Having the ranked list of bug locations (i.e., connections), it visualizes the classes connected to that.

Fiechter et al. [49] introduced a notion of *issue tale* – a visual narrative of the events and actors revolving around any GitHub issue. They presented an approach, implemented as an interactive visual analytic tool, to depict and analyze the relevant information pertaining to issue tales.

Tantithamthavorn et al. [16] developed a just-in-time defect prediction task and used LIME [17] to explain why a certain file is identified as defective. They further contributed a tool named LoRMikA to generate what to do to decrease the risk of being defective. While it is one of the earliest works in explainable AI for software engineering, it lacks significant technical contribution. Later, they complimented this work in PyExplainer [50] where they make a "dummy" model that behaves similar to the original model at a given data point. This dummy model can be a decision tree or linear regression, which are explainable by nature. Having the explanation, they visualized which features effects the defect prediction. They further added interactive features to adjust the features to provide intuition about how the users can turn a 'true' prediction into a 'false' prediction.

V. Threats To Validity

A. Internal Validity

Threats to internal validity denote the possibility of error while experimenting. In our case, it relates to potential bugs in the implementation of the explanation system. While bugs in software systems are inevitable, we double-checked all of our implementations to minimize the possibility of bug occurrence. The authors of **LR+WE** provide fine-grained details about their approach, which makes the implementation intuitive and obvious. We also keep our code-base modular and decoupled to mitigate potential sources of bugs. Furthermore, the first author of this report worked as a software engineer for more than two and a half years and is currently researching software engineering and deep learning. Given the experience, the possibility of defect turns to minimal.

B. External Validity

Threats to external validity denote the lack of generalizability of the work. In our case, it relates to the usability of our system to the general users. While performing a user study is the best way to mitigate such threats, we are restricted by the project timeline and approval of the research ethics board (REB). Therefore, we informally collect feedback from colleagues and peer researchers regarding the system's usability. We further open this project for public access so that anyone can use it and provide feedback. Moreover, this project is continuously evaluated by the course instructor, who has a vast experience in visual explanation, which further mitigates the risk of lack of usability.

Apart from the feedback, to ensure generalizability and usability, we use blue and vermilion (referenced as orange in earlier sections) as the colour code, which is visible to colour blind persons [51]. We also use CIELAB colour space [52] defined by International Commission on Illumination for the interpolation between colours. CIELAB colour space ensures the distance in "perceived" colour accurately reflects the distance in the underlying similarity score. Following McKinley's accuracy ranking of quantitative perceptual tasks [31], we try to use size and positions as much as possible to encode the numeric values. In cases where it is not possible, we show the similarity scores on focus (e.g. click or hover) to aid the colour encoding.

VI. Conclusion

In recent years, explainability has become a severe concern in AI-based applications. However, bug localization, a major domain of software engineering research, still severely lacks explainability support. In this paper, we propose a novel tool to explain one of the *state-of-the-art* bug localization techniques. We believe our work will work as a milestone towards explainable bug localization.

References

[1] Fiorenza Brandy. Cambridge University Study States Software Bugs Cost Economy $312 Billion Per Year, 2013. URL https://goo.gl/okoj21.

[2] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible Debugging Software: Quantify the time and cost saved using reversible debuggers, 2013.

[3] Liliana Favre. Modernizing software amp; system engineering processes. In *2008 19th International Conference on Systems Engineering*, pages 442–447, 2008. doi: 10.1109/ICSEng.2008.18.

[4] R.L. Glass. Frequently forgotten fundamental facts about software engineering. *IEEE Software*, 18(3):112–111, 2001. doi: 10.1109/MS.2001.922739.

[5] Scott Matteson. Report: Software failure caused $1.7 trillion in financial losses in 2017, 2018. URL https://tek.io/2FBNl2i.

[6] Weiqin Zou, David Lo, Zhenyu Chen, Xin Xia, Yang Feng, and Baowen Xu. How practitioners perceive automated bug report management techniques. *IEEE Transactions on Software Engineering*, 46(8):836–862, 2020. doi: 10.1109/TSE.2018.2870414.

[7] Mohammad Masudur Rahman, Foutse Khomh, Shamima Yeasmin, and Chanchal K. Roy. The forgotten role of search queries in ir-based bug localization: an empirical study. *Empirical Software Engineering*, 26 (6):116, Aug 2021. ISSN 1573-7616. doi: 10.1007/s10664-021-10022-4. URL https://doi.org/10.1007/s10664-021-10022-4.

[8] Mohammad Masudur Rahman and Chanchal K. Roy. A systematic literature review of automated query reformulations in source code search, 2021.

[9] Weiqin Zou, David Lo, Zhenyu Chen, Xin Xia, Yang Feng, and Baowen Xu. How practitioners perceive automated bug report management techniques. *IEEE Transactions on Software Engineering*, 46(8):836–862, 2018.

[10] Mohammad Masudur Rahman and Chanchal K Roy. Improving ir-based bug localization with context-aware query reformulation. In *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 621–632, 2018.

[11] Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. Explainable software analytics. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 53–56, 2018.

[12] European Union. Automated individual decision-making, including profiling, 2018. URL https://gdpr-info.eu/art-22-gdpr/.

[13] Lilian Edwards and Michael Veale. Slave to the algorithm: Why a right to an explanation is probably not the remedy you are looking for. *Duke L. & Tech. Rev.*, 16: 18, 2017.

[14] Sandra Wachter, Brent Mittelstadt, and Luciano Floridi. Why a right to explanation of automated decision-making does not exist in the general data protection regulation. *International Data Privacy Law*, 7(2):76–99, 2017.

[15] Bryce Goodman and Seth Flaxman. European union regulations on algorithmic decision-making and a "right to explanation". *AI magazine*, 38(3):50–57, 2017.

[16] Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, and John C. Grundy. Explainable AI for software engineering. *CoRR*, abs/2012.01614, 2020. URL https://arxiv.org/abs/2012.01614.

[17] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Model-agnostic interpretability of machine learning. *arXiv preprint arXiv:1606.05386*, 2016.

[18] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th international conference on software engineering*, pages 404–415, 2016.

[19] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics

and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, pages 181–190, 2008.

[20] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 964–974, 2019.

[21] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[22] Xin Ye, Razvan Bunescu, and Chang Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 689–699, 2014.

[23] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 433–436, 2007.

[24] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014. URL http://www.aclweb.org/anthology/D14-1162.

[25] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

[26] Robert Parker, David Graff, Junbo Kong, Ke Chen, and Kazuaki Maeda. English Gigaword Fifth Edition, 2011. URL https://catalog.ldc.upenn.edu/LDC2011T07.

[27] Miguel Grinberg. *Flask web development: developing web applications with python*. " O'Reilly Media, Inc.", 2018.

[28] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

[29] Evan You. Vue.js - The Progressive JavaScript Framework, 2021. URL https://vuejs.org/.

[30] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D¡sup¿3¡/sup¿ data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17 (12):2301–2309, dec 2011. ISSN 1077-2626. doi: 10.1109/TVCG.2011.185. URL https://doi.org/10.1109/TVCG.2011.185.

[31] William S. Cleveland and Robert McGill. Graphical perception: The visual decoding of quantitative information on graphical displays of data. *Journal of the Royal Statistical Society: Series A (General)*, 150(3):192–210, 1987. doi: https://doi.org/10.2307/2981473. URL https://rss.onlinelibrary.wiley.com/doi/abs/10.2307/2981473.

[32] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[33] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[34] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[35] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. On the use of stack traces to improve text retrieval-based bug localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 151–160, 2014. doi: 10.1109/ICSME.2014.37.

[36] Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355, 2013. doi: 10.1109/ASE.2013.6693093.

[37] Shaowei Wang and David Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*, ICPC 2014, page 53–63, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328791. doi: 10.1145/2597008.2597148. URL https://doi.org/10.1145/2597008.2597148.

[38] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 14–24, 2012. doi: 10.1109/ICSE.2012.6227210.

[39] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. Automatic query reformulations for text retrieval in software engineering. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 842–851. IEEE, 2013.

[40] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Bug localization with combination of deep learning and information retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 218–229, 2017. doi: 10.1109/ICPC.2017.24.

[41] Trevor Strohman, Donald Metzler, Howard Turtle, and W Bruce Croft. Indri: A language model-based search engine for complex queries. In *Proceedings of the international conference on intelligent analysis*, volume 2, pages 2–6. Citeseer, 2005.

[42] SF Dierk. The smart retrieval system: Experiments in automatic document processing—gerard salton, ed.(englewood cliffs, nj: Prentice-hall, 1971, 556 pp.). *IEEE Transactions on Professional Communication*, (1): 17–17, 1972.

[43] Stephen E Robertson. On term selection for query

expansion. *Journal of documentation*, 1990.

[44] Claudio Carpineto and Giovanni Romano. A survey of automatic query expansion in information retrieval. *Acm Computing Surveys (CSUR)*, 44(1):1–50, 2012.

[45] Joe Ruthruff, Eugene Creswick, Margaret Burnett, Curtis Cook, Shreenivasarao Prabhakararao, M Fisher, and Martin Main. End-user software visualizations for fault localization. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 123–132, 2003.

[46] Michele Risi and Giuseppe Scanniello. Metricattitude: a visualization tool for the reverse engineering of object oriented software. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, pages 449–456, 2012.

[47] Rita Francese, Michele Risi, and Giuseppe Scanniello. Enhancing software visualization with information retrieval. In *2015 19th International Conference on Information Visualisation*, pages 189–194. IEEE, 2015.

[48] Gelareh Meidanipour Lahijany, Manuel Ohrndorf, Johannes Zenkert, Madjid Fathi, and Udo Kelte. Identibug: Model-driven visualization of bug reports by extracting class diagram excerpts. In *2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 3317–3323. IEEE, 2021.

[49] Aron Fiechter, Roberto Minelli, Csaba Nagy, and Michele Lanza. Visualizing github issues. In *2021 Working Conference on Software Visualization (VISSOFT)*, pages 155–159. IEEE, 2021.

[50] Chanathip Pornprasit, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Michael Fu, and Patanamon Thongtanunam. Pyexplainer: Explaining the predictions of just-in-time defect models. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 407–418. IEEE, 2021.

[51] Bang Wong. Points of view: Color blindness. *Nature Methods*, 8(6):441–441, Jun 2011. ISSN 1548-7105. doi: 10.1038/nmeth.1618. URL https://doi.org/10.1038/nmeth.1618.

[52] International Commission on Illumination. *Colorimetry*. CIE technical report. Commission Internationale de l'Eclairage, 2004. ISBN 9783901906336. URL https://books.google.ca/books?id=P1NkAAAACAAJ.