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 Principle 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 better understand the bug 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 ≈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 the 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 really guarantee a system's success once it is 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 given as text, 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. 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 high dimensional word embedding in two-dimensional space that users can percept and understand?
- **RQ$_2$**: How can we visualize similarity calculations between bug report and candidate source codes?

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 contains the tasks that still remain to complete this project. Finally, section V 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.

Word2vec technique proposed by Mikolov et al. [21] is one of the most widely used embedding generation techniques. 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.

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.

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

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

$$P(T \rightarrow S) = \{w\,\epsilon\,T \mid sim(w, S) \neq 0\} \tag{3}$$

$$sim(T \rightarrow S) = \frac{\sum_{w \epsilon P(T \rightarrow S)} sim(w, S)}{|P(T \rightarrow S)|} \tag{4}$$

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

where $sim(w, w')$ denotes the similarity between two words, $sim(w, T)$ denotes the similarity between a word and a document, $sim(T \rightarrow S)$ denotes the asymmetric similarity between two documents, and $sim(T, S)$ denotes the symmetric similarity between two documents.

Finally, a linear regression based learning to rank model takes the similarity scores and produces a final ranking.

III. Progress

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. 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 [22]. We use the 300-dimensional version of their embedding, which is trained on Wikipedia and Gigaword [23] datasets. After the replication, we wrap it in a minimalistic web app using Flask Python framework [24]. 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.

B. Client Server

To visually explain the bug localization, we build a client server using TypeScript [25], Vue JS [26], and D3 [27]. Figure 1 is a screenshot of the home page of the client app. Upon selecting a bug report from the dropdown in the top-left corner, it 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 see the similarities at four levels – word to word similarity, bug report's word to source files similarity, source files' word to bug report similarity, and bug report to source files similarity. *Among them, we have completed the first three levels and visualizing bug report to file similarity remains.*

The reported version of the app is available for public access[1]. Details of the completed visualizations are discussed in the following.

1) Visualizing Word Embedding: Since word embedding is a high dimensional data (e.g. 300 in GloVe [22], 768 in BERT [28]), a human cannot visualize it directly. The use of dimensionality reduction techniques like principal component analysis (PCA) [29] and t-Distributed Stochastic Neighbor Embedding (t-SNE) [30] is common to visualize such data. To answer our **RQ$_1$**, 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.

In figure 1, we see the user can select a particular file to see the word embeddings. In the canvas (marked by the blue square), cyan dots are the words from the bug report, and magenta dots are words from the selected file. Upon selecting the `t-SNE` radio button, the same words will appear using t-SNE (figure 2). The user can hover on the dots to see the corresponding words.

While PCA is purely a mathematical model, it is not always easy to perceive it (e.g. one node is hidden behind another. On the contrary, t-SNE provides better visuals with less accurate information (e.g. two neighbouring nodes in t-SNE might not

[1] http://129.173.67.225:8080/bug-localization

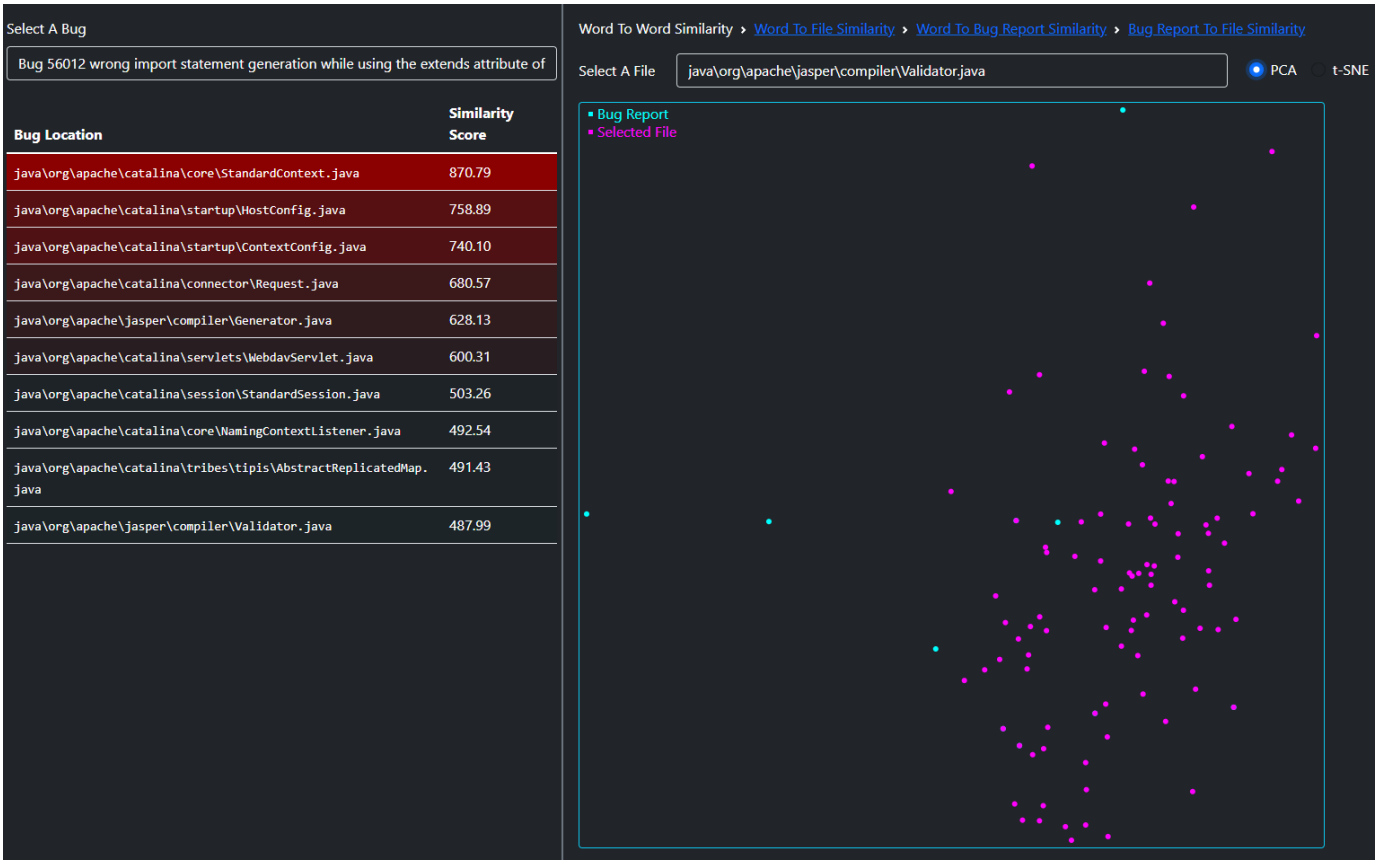

Fig. 1. The home screen[2]

be neighbours in higher dimension). In this app, the user can selectively see information based on his/her preference.

2) Visualizing Bug Report Words To File Similarities: Figure 3 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. Only the edges and source files with the maximum similarity within the limit will be shown. In the figure, we see that some nodes in the left do not have any edge due to the limit on the number of edges. The edges are colour-coded as per their strength of similarity, where red is the most similar and cyan is the least similar. Hovering on edges will show corresponding information, including the nodes and similarities.

3) Visualizing File Words To Bug Report Similarities: Figure 4 shows a visualization of similarities between source file words and the whole bug report (equation 2). 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 (maximum 30) in the

Fig. 2. A sample t-SNE visualization[3]

[2]Better quality: http://129.173.67.225:8080/ss/home.png
[3]Better quality: http://129.173.67.225:8080/ss/tsne.png

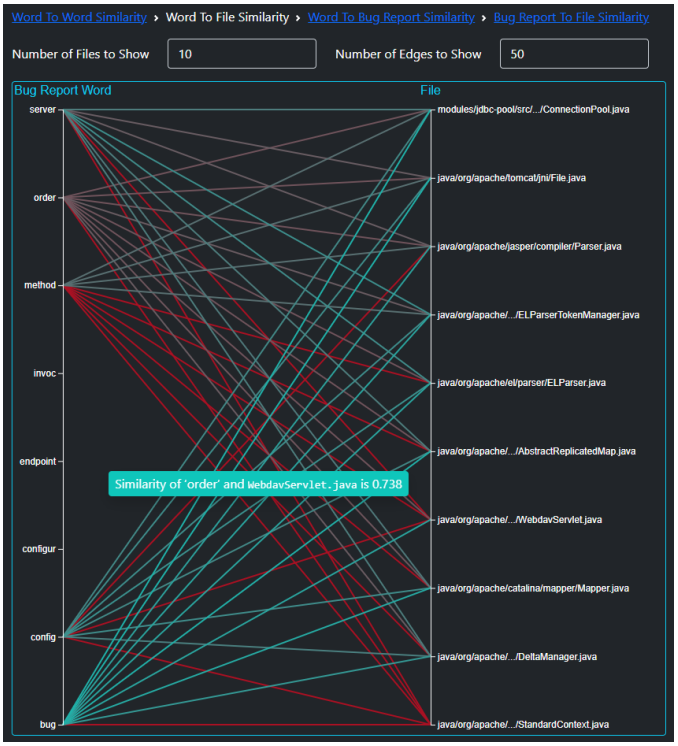

Fig. 3. Visualizing Bug Report Words to File Similarities[4]

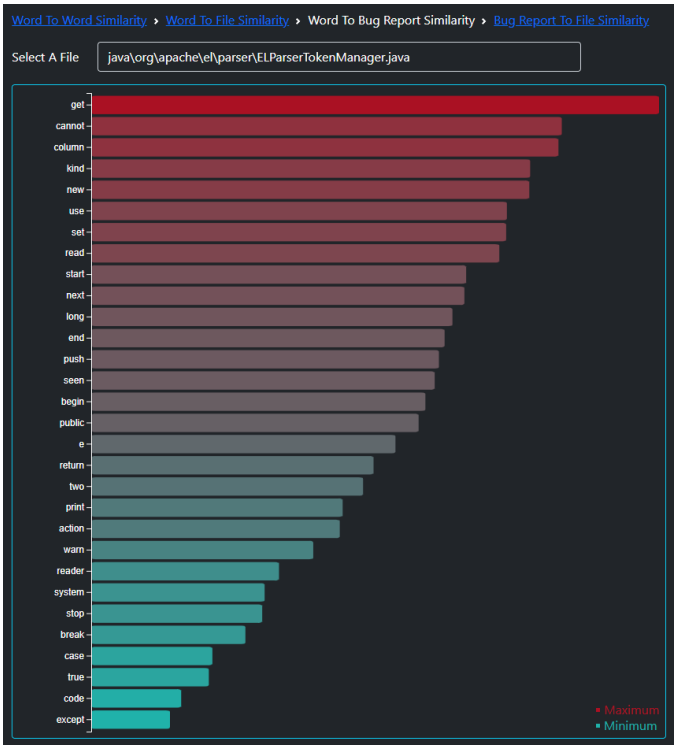

Fig. 4. Visualizing Source File Words to Bug Report Similarities[5]

source file will be shown. The bars are colour-coded as per the similarity strength. We do not show the similarity score in this graph either on the axis or on hover. The rationale is that the exact similarity scores merely express any meaning; instead, they help compare different words, and the bars can aid such comparison pretty well.

IV. Upcoming Tasks

In the project proposal, we estimated to complete visualizing word embeddings in 2-dimensional space within week eleven. Nonetheless, within week nine, we have completed visualizing word embeddings and further visualized half of the similarity scores. Therefore, compared to the timeline submitted in the project proposal, we are already quite ahead of our schedule. In the upcoming days, we will visualize –

1) Asymmetric similarity between source files to the bug report (equation 4)
2) Asymmetric similarity between bug report to source files (equation 4)
3) Symmetric similarity between the bug report and source files (equation 5)

As these three visualization are essentially three more bar plots, we are certain to complete the whole project within the allocated time.

[4]Better quality: http://129.173.67.225:8080/ss/word2file.png
[5]Better quality: http://129.173.67.225:8080/ss/word2bug.png

V. Conclusion

In recent years, explainability has become a serious concern of 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 Grundy. Explainable ai for software engineering. *arXiv preprint arXiv:2012.01614*, 2020.

[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] 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.

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

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

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

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

[27] 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.

[28] 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.

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

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