Enhancing BugLocator with doc2vec

Quinn Ceplis
Department of Electrical and Software Engineering
University of Calgary
Calgary, Canada
quinn.ceplis1@ucalgary.ca

***Abstract*—Any reasonably advanced software system will have a large backlog of bugs to be fixed, and more coming in all the time. Attempting to manually search for the location of a bug is not the best use of software developers' time, and a meaningful speedup and increase in quality could be achieved if more bugs are found more quickly. Automated fault localization provides one such avenue to achieve this. In this paper, we explore whether a semantic embedder like doc2vec presents a reliable method to improve fault localization results.**

***Keywords*—bug localization, bugLocator, doc2vec**

I. Introduction

Conventional approaches to fault localization can be tedious. They involve one or more developers searching the codebase manually for logical faults. It may take hours or even days before the fault is located, and this is even before a fix can be devised. It is estimated that fault localization costs the software industry X dollars per year. Developers who contribute to open-source projects may be unfamiliar with the different nuances of each codebase.

Fortunately, several automated fault localization systems exist. Through some analysis of the codebase and other development artifacts, these programs help developers zero in on where bugs are likely to be found. Several layers of granularity in localization are possible with these techniques, such as identifying the file, class, or method where the fault might reside. Depending on their inputs and vector approach, models might require more less input artifacts or computation time.

Multiple types of automated fault localization exist. One type is spectrum-based fault localization (SFL), where artifacts associated with test execution traces are analyzed. Another common type is information-retrieval fault localization (IRFL), which is what this paper will focus on. Only source code files and solved bug reports are required for this approach, making its input artifacts more ubiquitous and easier to obtain than other fault localization techniques—especially in an open-source context. Additionally, information-retrieval based techniques use less computational power compared to other techniques (Binkley and Lawrie, 2010).

What then IRFL techniques tick? Although each algorithm varies in what they search for, the core component of all these efforts is the search for textual similarity between bug reports and source code files. These techniques seek to create a ranking of files, classes, or methods where the bug is likely to appear. The sum total of each of these bug locations act like a database, that a bug report acts like a query on.

BugLocator was a seminal model for IRFL proposed by Zhou, Zhang, and Lo (2012). We seek to recreate its results as a baseline, then introduce a new method with the goal of improving its localization properties.

The goal of this paper is to compare the effectiveness of two different textual encoding styles while following the structure of BugLocator. The first encoding style is TF-IDF, used by the original BugLocator paper. The second is doc2vec (Le and Mikolov, 2014), a neural network approach to semantic embedding and encoding. In the following sections, more information is given on the two encoding schemes: TF-IDF to doc2vec.

In Section II, we give a more detailed background on fault localization and the approach used in this paper. In Section III, a rundown of how the new method works is presented. In Section IV, the experimental design is given an overview, and Section V discusses the results. Section VI presents related work in the field. Section VII presents the conclusion of the paper.

II. Background

Some way of finding the similarity between documents must be found for IRFL to work. In their original form, everyday sentences and documents (natural language), are too difficult for computers to effectively parse. To make it easier for computers to reason about sentences, we can use numeric encoding.

The type of encoding used by BugLocator is TF-IDF, or Term Frequency, Inverse Document Frequency. For each word in the corpus, the number of times it appears is multiplied against the inverse of the number of times it appears in a single document. This means that higher weight is given to words that are used often in a single document, and not often globally. A word like "the" may be used often in a single document, but it is also common across all documents giving it a low weight.

Human language is too varied to be effectively grouped and encoded, so some preprocessing is required. One effective tool to make words more legible to machines is stemming. Stemming is the process of converting words that are morphologically similar into their root word. For example, the words "displays", "displaying", or "displayer" would be reduced to the stem word "display".

Additionally, there may be common phrasings between bug reports and source code files, but these may be missed due to the concatenation of words in code. For example, a bug report may contain the text "Error when displaying the graph." and a method name in a related file might be "DisplayGraph". While these appear related to the human eye, a computer will not notice the similarity, even with stemming the word "displaying". Fortunately, most variable, method, and class names follow a case convention (such as Pascal, Camel, Kebab, etc.), which dictates how distinct words are combined. Using regular

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

expressions, these concatenated words can be split apart to be compared against their natural language counterparts.

Finally, words are made lowercase and punctuation is removed. Now that documents have been preprocessed and use an encoding scheme, we can start to search for the similarity between them. The discussion below highlights how similarity is found using BugLocator.

The first step is to find the direct similarity between bug reports and source code files. These files are first used to train a TF-IDF vectorizer, and then transformed using that same vectorizer into an encoded numeric vector. Each bug report is treated as a query that finds and ranks the relevant source code files. The bug report text and title are combined, then transformed using the vectorizer, then compared against each source code file using the cosine similarity between them. In BugLocator, the cosine similarity is multiplied by the normalized length of the source code file.

Next, we compute an indirectly relevancy between bug reports and source code files. Because the language in code may not match up with what is written in the bug reports, we can instead look to historical bug reports that are like the current query report. Based on the similarity between these two types of reports, we can then rank the files that these historical bug reports have fixed.

The TF-IDF vectorizer used in BugLocator emphasizes frequency of unique words within a document but doesn't capture sentence semantics. It is hoped that an encoding which does remember the sentence semantics will be able to find similarity between historical and querying bug reports more accurately.

This brings us to discussing the doc2vec encoding technique. However, an explanation of the word2vec technique is useful before fully uncovering the main technique. Both algorithms are built using neural networks to embed semantics into text and document encoding. Word2vec is the foundation that doc2vec builds upon.

Word2vec is built on two techniques: continuous bag of words (CBOW) and skip-gram (SG). Given the context of surrounding words, CBOW tries to predict what word will be in the middle. SG takes a single word and tries to predict what words surround it. These predictions can be done by performing operations on a unique word vector. To illustrate this, we will contrast it with one-hot encoding. In one-hot, a one represents the presence of a word in a document, and zero its absence. In the word2vec methods, each number in the word vector represents the likelihood that a word is next to the word we're interested in.

Because this paper is concerned with ranking source code files by similarity, not words, an additional parameter is needed. This is provided by the algorithms in doc2vec.

III. Method

In this section, approach to implementing the novel technique is discussed.

We start with recreating BugLocator's direct similarity results, which will be used in combination with indirect similarity results computed using doc2vec.

Train a TF-IDF vectorizer on the source code files for the current project. Transform source code files into a series of vectors using the vectorizer. Transform query bug reports using TF-IDF vectorizer.

Calculate the cosine similarity of source code vectors and query bug report vectors. BugLocator uses a modified version of cosine similarity that takes into account the normalized length of source code files.

For indirect relevancy, use doc2vec to predict similarity between current and past bug reports. Because source codes operate under the semantics of the language they were written in, they do not offer a reasonable comparison to bug reports using doc2vec. The genism implementation of doc2vec is used for training and testing similarity.

Indirect relevancy is calculated as follows.

$$\sum_{\substack{\text{All } S_i \text{ that} \\ \text{connect to } F_j}} (Similarity(B, S_i) / n_i)$$

Where B is the current query bug, F_j is the file we want to find indirect similarity to, and S_i is one of the historical bug reports connected to S_i.

Once both direct and indirect similarity are found, we linearly combine them.

$$Final\ Score = (1 - \alpha) \times N(Direct\ Similarity) + \alpha \times N(Indirect\ Similarity)$$

Where α is scalar for weighting the importance of each relevance, and $0 \leq \alpha \leq 1$. N() is the normalization function.

IV. Experimental Design

A. Objectives

Recreate the results of the BugLocator paper

- **Motivation**: Have a baseline that we can compare the results of using doc2vec in fault localization to.

Compare BugLocator to a model that also uses doc2vec.

- **Motivation**: Determine if doc2vec presents a more reliable way to locate faults than the conventional approach.

B. Procedure

1. *Gather bug reports and source code files from Bench4BL*. Bugs included here should be guaranteed to be fixed, allowing us to effectively use them to evaluate the model's performance.

2. Split reports and files by project. Split bug reports by current and historical bugs.

Table 1: Number of Bug Reports and Source Code Files in each project analyzed in Bench4BL.

Project	# Bug Reports	# Source Files
COLLECTIONS	92	476
CONFIGURATION	133	224
DATACMNS	158	552
DATAMONGO	271	348
DATAREST	132	348
ELY	25	68
LANG	217	247
SOCIALFB	15	253

3. *Preprocess all text.* Remove punctuation, split concatenated words, make lowercase, remove stop words, and stem words for commonality. In bug reports, titles and summaries are combined to give the best breadth of input possible.
4. *Train an initial model using TF-IDF on source files.* Compute direct and indirect relevancy using this encoding.
5. *Train a secondary model doc2vec on historical bug reports.* Compute indirect relevancy using this encoding.
6. *Combine direct relevancy from TF-IDF with indirect relevancy using doc2vec.* This will be the final results for the novel model.
7. *Evaluate the two models' performance.* Use Top N Rank, Mean Reciprocal Rank, and Mean Average Precision.
8. *Compare evaluation results between the two models.*

C. Measurements

To measure how effective the model is at fault localization, the following techniques are used:

1. *Top N Rank:* Evaluates the percentage of bug reports per list of queries where a fix to the bug exists within the top N ranked files returned by the query. Larger scores of Top N Rank are better.
2. *MRR (Mean Reciprocal Rank):* Given a ranked list of documents returned after a search query, this measure checks how far down the list the first relevant document is. Given several test queries Q, this metric then takes the average of the inverse rank of the relevant documents for every query. Larger scores of MRR are better.
3. *MAP (Mean Average Precision):* When a query should return multiple relevant documents, sum up the number of positive hits divided by the rank they were encountered in. Taking the mean of these average precisions for each bug report query yields the final result. Larger scores of MAP are better.

D. Research Questions

1. How many bugs can be successfully located using this method?
2. How does using doc2vec for indirect relevancy fare in comparison to TF-IDF?

E. Results

Table 2: Direct TF-IDF Results

	Top 1	Top 5	Top 10	MRR	MAP
COLLECTIONS	0.210526	0.578947	0.578947	0.36892	0.299534
CONFIGURATION	0.37037	0.703704	0.851852	0.544123	0.373892
DATACMNS	0.3125	0.625	0.78125	0.463499	0.35038
DATAMONGO	0.181818	0.418182	0.545455	0.302155	0.222569
DATAREST	0.222222	0.481481	0.555556	0.33445	0.260501
ELY	0.2	0.6	0.6	0.366671	0.383333
LANG	0.477273	0.704545	0.772727	0.583146	0.527897
SOCIALFB	0.333333	0.666667	1	0.477778	0.481944

Table 3: Combined TF-IDF Results:

	Top 1	Top 5	Top 10	MRR	MAP
COLLECTIONS	0.421053	0.631579	0.631579	0.512847	0.483484
CONFIGURATION	0.555556	0.851852	0.888889	0.679314	0.552608
DATACMNS	0.34375	0.71875	0.78125	0.49996	0.400845
DATAMONGO	0.381818	0.618182	0.654545	0.483704	0.375377
DATAREST	0.37037	0.666667	0.703704	0.48709	0.376781
ELY	0.2	0.6	0.6	0.400004	0.416667
LANG	0.522727	0.75	0.772727	0.622603	0.581882
SOCIALFB	0.333333	0.666667	0.666667	0.474747	0.480429

Table 4: Combined Doc2vec Results

	Top 1	Top 5	Top 10	MRR	MAP
COLLECTIONS	0.315789	0.631579	0.631579	0.451444	0.433427
CONFIGURATION	0.518519	0.814815	0.888889	0.663962	0.539746
DATACMNS	0.25	0.6875	0.8125	0.437025	0.33955
DATAMONGO	0.272727	0.618182	0.690909	0.42687	0.337376
DATAREST	0.37037	0.62963	0.703704	0.491345	0.366368
ELY	0.2	0.6	0.6	0.400004	0.416667
LANG	0.522727	0.727273	0.772727	0.625092	0.580516
SOCIALFB	0.333333	0.666667	0.666667	0.474747	0.481818

V. Discussion of Results

On initial inspection, the number of bugs that can be located with this method seems impressive. Taking the average of each of the stats for combined doc2vec yields the following:

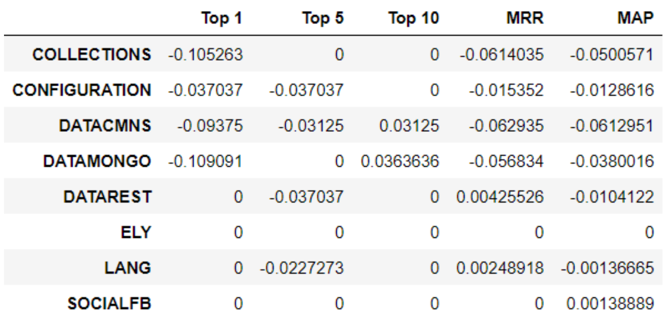
Table 5: Average Results for Combined Doc2vec Model

	Top 1	Top 5	Top 10	MRR	MAP
AVERAGE	0.347933	0.671956	0.720872	0.496311	0.436933

This method can find a bug in the highest rank position 35% of the time, in the top 5 67% of the time, and in the top 10 72% of the time. Scores for MRR and MAP are also reasonable.

However, an issue arises when we compare these results to those generated by combined TF-IDF. For each project and metric in the below table, the results from Combined TF-IDF were subtracted from Combined Doc2Vec. Therefore, any negative number indicates where Combined TF-IDF performed better than Combined Doc2Vec.

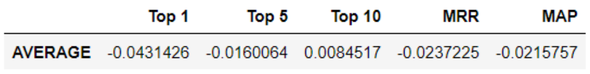
Table 6: Combined Doc2Vec Results, subtract Combined TF-IDF Results



	Top 1	Top 5	Top 10	MRR	MAP
COLLECTIONS	-0.105263	0	0	-0.0614035	-0.0500571
CONFIGURATION	-0.037037	-0.037037	0	-0.015352	-0.0128616
DATACMNS	-0.09375	-0.03125	0.03125	-0.062935	-0.0612951
DATAMONGO	-0.109091	0	0.0363636	-0.056834	-0.0380016
DATAREST	0	-0.037037	0	0.00425526	-0.0104122
ELY	0	0	0	0	0
LANG	0	-0.0227273	0	0.00248918	-0.00136665
SOCIALFB	0	0	0	0	0.00138889

This is further condensed in the table below, which shows the average difference for the two models.

Table 6: Average difference in results between Combined Doc2Vec Results and Combined TF-IDF Results



	Top 1	Top 5	Top 10	MRR	MAP
AVERAGE	-0.0431426	-0.0160064	0.0084517	-0.0237225	-0.0215757

As can be seen, Combined Doc2Vec often performs worse than Combined TF-IDF. It was predicted that the model would perform reasonably well no matter the size of the corpus, because the words trained on would be most similar to those being tested on. However, it seems that more training data is required to effectively learn the semantics and predict results. Perhaps if doc2vec were trained on a larger corpus, better results could be found.

VI. Related Work

The most obvious related research is BugLocator (Zhou et al. 2012). This paper directly builds on top of the results presented there. This paper also draws inspiration from GloBug (Miryeganeh et al. 2020). One motivation for this paper was to see if doc2vec could be effective without relying on a global dataset. DeepFL (Li et al. 2019) presents an extension of where a conditionally relevant paper might be applicable in other domains. IRBFL (Li. et al, 2020) combines spectrum and information retrieval-based fault localization. Perhaps other spectrum based techniques could be integrated with the techniques presented here in future papers. Lal and Sureka's 2012 paper Character n-grams presents a fascinating way to find faults at a more granular level. With the development of more advanced neural network techniques, it would be worth studying how such techniques could support each other in future papers. Rodríguez-Pérez et al (2020) presents a more nuanced look at where bugs form, which could be useful in honing in on where to search for faults using the techniques presented in this paper. Zhang et al (2018) works with similar neural embedding techniques as this paper, but also incorporates user reviews for finding the best spot to make changes. Qianqian et al (2015) provides a synopsis of major fault localization techniques in the community that can be drawn on for future papers.

VII. Conclusion

Bugs can quickly eat up developer time but fixing them is required for successful shipping of a new software release. This process is expensive and tedious. Using automated fault localization techniques allows for faster turnaround time between reporting and fixing bugs. In this paper, we investigated if textual encoding using neural network semantic encoding presents a significant improvement over traditional information retrieval-based fault localization encoding techniques. Evaluating this technique on real-world code repositories shows that it can predict the location of bugs reasonably well, it often slightly detracts from the performance of the algorithm compared to results found in related papers like BugLocator.

In the future, we would like to explore if training doc2vec on different corpus sizes would meaningfully affect the results observed within this paper.

References

[1] Binkley, D., Lawrie, D., 2010. Information retrieval applications in software maintenance and evolution. Encyclopedia Softw. Eng. 454–463.

[2] Fahad, A.H., 2018. Doc2Vec — Computing Similarity between Documents. URL https://medium.com/red-buffer/doc2vec-computing-similarity-between-the-documents-47daf6c828cd

[3] Lal S., Sureka A., 2012. A static technique for fault localization using character n-gram based information retrieval model. In: ISEC '12: Proceedings of the 5th India Software Engineering ConferenceFebruary 2012 Pages 109–118 URL https://doi.org/10.1145/2134254.2134274

[4] Le, Q.V., Mikolov, T., 2014. Distributed representations of sentences and documents. In: Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014. pp. 1188–1196, URL http://jmlr.org/proceedings/papers/v32/le14.html.

[5] Lee, J., Kim, D., Bissyandé, T.F., Jung, W., Le Traon, Y., 2018. Bench4bl: Reproducibility study on the performance of IR-based bug localization. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2018, ACM, New York, NY, USA, pp. 61–72. URL http://dx.doi.org/10.1145/3213846.3213856, http://doi.acm.org/10.1145/ 3213846.3213856.

[6] Li X., Li W., Zhang Y., Zhan L.2019. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. ISSTA 2019: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and AnalysisJuly 2019 Pages 169–180 URL https://doi.org/10.1145/3293882.3330574

[7] Li Z., Bai X., Wang H., Liu Y., 2020. IRBFL: An Information Retrieval Based Fault Localization Approach. 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)

[8] Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J., 2013. Distributed representations of words and phrases and their compositionality. In: Burges, C.J.C., Bottou, L., Welling, M., Ghahramani, Z., Weinberger, K.Q. (Eds.), Advances in Neural Information Processing Systems 26. Curran Associates, Inc., pp. 3111–3119.

[9] Miryeganeh N., Hashtroudi S., Hemmati H., 2020. GloBug: Using global data in Fault Localization. In: The Journal of Systems & Software 177, 2021

[10] Qianqian, Parnin,; Orso, ACM 2015. Evaluating the usefulness of IR-based fault localization techniques. In: Proceedings of the 2015 International Symposium on software testing and analysis, 2015-07-13, p.1-11

[11]

[12] Rodríguez-Pérez G., Robles G., Serebrenik A., Zaidman A., Germán D.M., Gonzalez-Barahona J.M. 2020. How bugs are born: a model to identify how bugs are introduced in software components. In Empirical Software Engineering. URL https://link.springer.com/article/10.1007/s10664-019-09781-y

[13] Tan, R.J., 2019. Breaking Down Mean Average Precision (mAP). URL https://towardsdatascience.com/breaking-down-mean-average-precision-map-ae462f623a52

[14] Zhang, Chen, Zhan, Luo‡ , Lo, and Jiang 2018. Where2Change: Change Request Localization for App Reviews. In IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. , NO. , 2018

[15] Turnbull D., 2021, Compute Mean Reciprocal Rank (MRR) using Pandas. URL https://softwaredoug.com/blog/2021/04/21/compute-mrr-using-pandas.html

[16] Zhou J., Zhang, H., Lo D., 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. (2012). ICSE 2012: 34th International Conference on Software Engineering, Zurich, June 2-9. 14-24. Research Collection School Of Information Systems. Available at: https://ink.library.smu.edu.sg/sis_research/1531