Embedding Bugs: Leveraging User Q&As to Locate Bugs in Source Code

Caitrin Armstrong
McGill University
caitrin.armstrong@mail.mcgill.ca

ABSTRACT

Many longstanding and emergent problems in software engineering are at their root a question of measuring correspondence between source code and natural language (e.g. query response and feature localization). It has been proposed that word embedding approaches will allow us to bridge this *lexical gap*, as word embeddings allow a representation of relative semantic relatedness in a language-agnostic space. This paper shows an attempt at replication of a well-cited publication addressing the problem of bug localization using word embeddings. We use a novel training dataset as our source for developing word embeddings but test on a common, standardized dataset. We provide insights on the process behind experiment replication, offering advice to those wishing to increase the replicability of their publications. We demonstrate the influence of choices in preprocessing steps, further highlighting the need for extensive experiment reporting as the field of software engineering continues to integrate machine learning tools.

ACM Reference Format:
Caitrin Armstrong. 2018. Embedding Bugs: Leveraging User Q&As to Locate Bugs in Source Code. In *Proceedings of Draft (COMP 762).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The bug localization task can be defined as follows: given the text of a bug report written by a developer or user, return the file or files that will need to be edited to fix the bug. The solution to this task must therefore take natural language as input, and produce a ranking of source code files. Standard information retrieval methods fail to contend with the lexical gap between the language used in the user report and the source code of the target files [2]. As illustrated in the related problem of automatic code search [9], there is usually a mismatch between the high level content found in user queries and implementation code details.

This gap is illustrated in Figures 1 and 2. In Figure 1 the source code file contains code that manages spacing and styling for the eclipse project but it does not contain any of the keywords contained in the bug report. A word-frequency based technique would not allow us to correctly identify the target file, as there are no or very few similar terms. In addition bug reports are often short, which further restricts the ability of frequency-based models to make accurate predictions.

However, even an untrained individual can validate the potential for correspondence between the bug report text and the source code; they appear to be addressing similar issues despite the use of different vocabulary and syntax. Clearly, they share semantic similarity.

According to the distributional hypothesis [4], words that appear in same contexts tend to have similar semantic meanings. The concept of word embeddings takes advantage of this by creating vector representations of words based on their co-occurrences. Words that occur together in similar contexts should have similar meanings and therefore similar vectors. Very large datasets will then allow for a complex representation of the relationships between many words and their contexts.

This paradigm of vectorizing frequently-observed contexts to represent concepts has been applied successfully to NLP tasks, and even to spaces more complex than the textual domain e.g. Doc2Vec or World2Vec employed by Facebook [1]. In general, these vector models are trained and optimized to the task at hand, although large pre-trained general-corpus-based models will have reasonable success on simple NLP tasks.

In such a novel application as this, pre-trained embeddings do not exist, and we find it difficult to believe that they would be relevant. Source code is highly specialized, as are the relevant bug reports. In addition, to bridge the aforementioned lexical gap we must force a correspondence between source code and natural language if we are to later translate from one to the other; this will not occur in conventional word embedding training methods, as these constrain the context surrounding a word to a pre-set window. Code and natural language are often well-separated in space, even within the same file.

In this paper we attempt to replicate Ye et al. (2016) in applying trained embeddings to the problem of bug localization using the exact validation dataset, which was described earlier in Ye et al. (2014) [19] [20]. They trained their word embedding models on project documentation. While we replicate all other model settings where possible, we find that the documentation they used was not always project-specific, possibly leading to embeddings that are two general for the problem at hand.

Thus, we instead train our embeddings on project-specific exchanges taken from *Stackoverflow.com*, finding that even with the new datasets we have similar vocabulary sizes to Ye et al. across all projects tested. We hypothesize that the use of training data from Stackoverflow provides an additional advantage because of the dense relationship between code and natural language; users will often explain short snippets of code with interspersed natural language [3],[14] and [16]. Finally, Stackoverflow focuses, as

COMP 762, April 2018, Montreal, QC, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Bug id: 420238
Summary: Bug 420238 [CSS] Reduce ***whitespace*** usage in the default Eclipse **themes**
Description: One of the major complains I hear is that the new ***styling*** of Eclipse 4 wastes a lot of ***space***. I suggest to reduce this to a minimum in the **default** themes and see if community themes go a different way.

```
.....
public class CleanupAddon {
..

    Display display = Display.getCurrent();
    Assert.isNotNull(display);

    private void subscribeVisibilityChanged(
        @UIEventTopic(UIEvents.UIElement.TOPIC_VISIBLE) Event event) {
        ....
            if (changedObj.isVisible()) {
                // Make all the parents visible
                if (!parent.isVisible()) {
                    parent.setVisible(true);
                }
                ....
}
```

Figure 1: Text from a bug report about the wastage of *spaces* in the *styling* of an Eclipse theme, and the source code of the file containing the eventual bug fix.

Bug id: 420126

Summary: Bug 420126 [FastView] Parts of eclipse ***window*** become a light blue square and don't get ***repainted***.

Description: I have run into this both on 4.4 M2 and Version: 4.4.0 Build id: N20131014-2005 Suddenly (don't recall a fixed pattern of user action preceding the event) a part of eclipse ***window*** becomes a light blue blank square (I am on Windows 64, default out of the box color schemes if that matters) and does not ***refresh*** itself anymore. On one occasion, the size of this area was half the overall screen, at other times, perhaps a quarter. ***Minimize/maximize***, ***resize*** the subwindows, reset perspective etc don't help. I have to shutdown eclipse and restart for it to go away. Somehow some ***widget*** is losing track of its ***size*** or is not responding to ***repaint event*** - may be. If this is a platform/ui issue, please reassign. This makes M2 pretty unusable for me actually.

```
.........
public class CTabRendering extends CTabFolderRenderer implements ICTabRendering {.....
...
static final int INTERNAL_SPACING = 4;
...

@Inject
    public CTabRendering(CTabFolder parent) {
        super(parent);
        parentWrapper = new CTabFolderWrapper(parent);
    }
..
}
```

Figure 2: Eclipse bug report about *repainting windows* and the source code of the file containing the eventual bug fix.

do bug reports, on *issues* that users are having with a language. We hypothesize that the similarity between Stackoverflow **questions** and Bug **reports**, where each leads to a code-based solution would provide increased contextual similarity as compared to the documentation seen in [20]. For a discussion of the importance of context see [5]. While the issue is complex it can be concluded that improved context is important for generating improved word embeddings!

Finally, we note that this paper also takes the form of an interrogation of the *process of experiment replication* for machine learning papers in the field of software engineering. Machine learning offers new challenges for scientific replication, as models can often be poorly understood and dependent on many as-yet unstandardized preprocessing steps. In addition we provide commentary on the importance of reporting null findings.

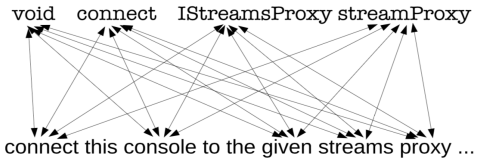

Figure 3: Positive pairs generated from semantically related text and code. Taken from [20]

2 RELATED WORK AND BACKGROUND

Word embeddings are a neural-network-based approach that represent each word with a low- dimensional vector called 'neural embeddings' or 'word embeddings'. Mikolov's Skip-gram model [12] is popular because of its simplicity and efficiency. The Skip-gram model was also shown to significantly out- performs LSA and other traditional frequency-based approaches in information retrieval and natural language processing tasks [13].

To learn word embeddings means to find vector representations such that words that are similar in meaning have similar vectors to those with similar meanings. Given a sample corpus, every word is encoded as a vector using one-hot encoding, then a neural network is trained the maximize the ability to predict a word given a certain context window of the other words that surround it. (i.e. we would expect the word "how" to occur given the surrounding context 'hello, _ are you?'). Negative sampling of unrelated words is often employed as a tool that re-weights the neural network weight vectors away from predicting words that do not occur in the windows surrounding the current target word. See [12] for a more complete explanation.

In Ye et. al 2016, the authors introduced for the first time a general approach to bridging the lexical gap between natural language and source code. They showed how word embeddings pre-trained using corpus documents had relevance for problems including bug localization and API recommendation. They attributed this to the ability of word embeddings to assign meanings to tokens based on context, irrespective of exact vocabulary match. To force the meaning representations of co-occurring natural language and code tokens even closer than found in conventional word embeddings they supplemented their SkipGram model with the pairings between every code token and every natural language token. This is illustrated in Figure 3, taken from Ye et al 2016 [20].

The paper that we are replicating appears to be the first to consider word embeddings for the task of bug localization, however others have since followed in their footsteps, additionally integrating deep learning and neural machine translation techniques as well as integrated techniques leveraging structural information [6, 7, 17, 18]. Previous approaches to the bug localization task, while varied in terms of model, were based on extracting structural or token frequency information [8].

3 APPROACH

We follow the approach described in [20] as closely as possible on our novel dataset.

3.1 Datasets

Word Embeddings Dataset. To train the model, we created a corpus from Stackoverflow posts tagged according to each of the four chosen projects found in the validation dataset [20]. Using a freely-available Stackexchange (the parent domain of Stackoverflow) data dump on Archive.org we filtered posts (here defined as both Stackoverflow questions and answers, but not comments) according to the appropriate tag for each project. Random observation of the post content confirmed that each was relevant and the tag was appropriate. Tags appear to be heavily curated on Stackoverflow.com and thus we have a high degree of confidence in their specific relevance for each project. For the Eclipse-UI project we simply downloaded all posts with the 'eclipse' tag, meaning this is a broader sample; likely relevant to the other projects as they are related to Eclipse. All possible posts were included, with the aim of increasing the amount of training data. The number of posts per project is illustrated in 1.

Project Name	Number of Documents
Birt	1743
Eclipse	102856
JDT	689
SWT	5013

Table 1: Number of Stackoverflow documents per project

Bug Reports. We downloaded the freely available validation dataset created in [19]. Each dataset is comprised of several thousand bug reports for each of our 4 projects: *Birt, Eclipse SWT, Eclipse UI, Eclipse JDT*. From each, we selected the 30 most-recent bug reports for testing.

Source Code. Source code was downloaded for each of the four repositories as specified in [19]. For each bug report, we cached a before-fix version of the source code, using Git commit history to recreate the repository as it was before the code was fixed. This is necessary to ensure that we are testing and validating the bug localization task as it would be applied in industry use.

3.2 Text Pre-Processing

Text preprocessing was performed similarly across all three datasets, although each was subject to certain variations. We prepared our text for input to the word2Vec algorithm according to the two-vocabulary setting described in [20], which we explain below. They found no significant difference between the two-vocabulary setting and the one-vocabulary setting which did not differentiate between the same token seen in both code and natural language. We chose to implement only the two-vocabulary setting due to this lack of difference.

In the two-vocabulary setting all code tokens are surrounded by @−−@ in order to force a separation between the same token used in code versus the same used in natural language (i.e. "@public@" versus "public").

For all datasets, natural langauge if present was tokenized, stemmed using the porter stemmer, and rendered only in lowercase. If a given natural language word was not present in an English dictionary, it was instead coded as a code token (surrounded by @−-@).

Code was tokenized, and all non-alphanumeric characters were removed. In addition, all compound words were split into their component parts. For example, "CTabRendering" will be split into 'C', 'Tab' and 'Rendering', while we also preserve its original form. All split words that were found in an English dictionary were included in both their original and code-based form, as a single sentence in order to again force the meaning vectors of the two languages closer together. The wording of this pre-processing step was ambiguous in Ye et al. (2016) but we believe we replicated it as they intended.

A block of code surrounded by natural language was considered to be a single line or sentence.

All comments in source code were considered and processed as natural language. The title and description fields of the bug reports were merged to form a single text document representing a bug report.

Finally, we created all pairings between text and code words in a single document (file or sentence), in order to force the meaning representations of the natural language tokens and the code tokens to converge.

We note here the preprocessing decisions that we made that were not specified in our replication target, and therefore threaten our ability to directly compare our dataset to theirs. [20]

(1) The choice of whether or not to input file-level or sentence-level documents to the word2vec algorithm when training embeddings from the corpus dataset. Literature and precedence supports either option, although we do note that the similarity algorithms require a file-level input for the bug report text and source code.
(2) The choice of English dictionary, stemmer, and whether or not to include numeric characters.
(3) The choice to include natural language and code pairings as a separate sentence after splitting pairings, keeping the original sentence intact, or to integrate both terms into the original sentence. While we chose the former, it is not entirely clear in the replicated paper.

3.3 Generating Word Embeddings

To generate a word embedding model for each project we used the Gensim package Word2Vec on our corpus of Stackoverflow posts [15]. We experimented with integrating all 4 projects into a single corpus or only training on the project-specific corpus, as detailed in our experiments below. We do not report the results of hyperparameter optimization, as we found that our model performed uniformly irregardless, and given that our results found are not comparable in terms of performance to the replicated paper it is more prudent to simply report the results using the hyperparameters reported in Ye et al (2016). Thus, our window size was 10, the dimension was set to 100, and negative sampling was set to 25. We outline experiments below that highlight the influence of the hyperparameters unreported by Ye et al (2016), namely number of epochs and choice of boundary for creating training examples.

3.4 Generating Document Similarities

In order to evaluate on the prediction task we must generate for each bug report a ranking of all the documents in each project. Given that [20] implements a learning to rank algorithm in order to

compare combinations of word embedding models with prior work, we cannot replicate their method exactly. We follow, however, the method from which they devised their modified scoring system and believe them to be equivalent when used only on the embedding task. We follow the scoring function in [10], described in Equation 1 between two input text segments, and where *maxSim* describes the two most similar word in an entire document given an input word *w*. This can easily be measured using the similarity scores provided by the trained embedding. Thus, in order to find the files most relevant given the text of a single bug report, we apply the *sim(T1, T2)* algorithm where *T1* is the preprocessed report text and we iterate over all preprocessed source code documents for *T2*. The resultant scores are then ranked.

$$sim(T1, T2) = \frac{1}{2}(\frac{\sum_{w \in \{T_1\}}(maxSim(w, T_2) * idf(w))}{(\sum_{w \in T_1} idf(w))} + \frac{(\sum_{w \in \{T_2\}}(maxSim(w, T_1) * idf(w))}{(\sum_{w \in T_2} idf(w))})$$

Equation 1: Document similarity function

3.5 Experiments

We conduct 4 experiments. We imitate [20] as closely as possible, and where it is not possible to imitate them we report the influence of the unknown variables.

We found that [20] did not report how they constructed training examples from their corpus dataset: was each example fed into the Word2Vec model generated from an entire document, or from a single line of our document? Previous literature follows either of the two approaches, and the [20] is not entirely clear which approach they chose.

Ye et al. (2016) also did not report the number of epochs used when training their model. The number of epochs describes the number of passes over the training data, and has reported to be significant in some applications [11]. It is a parameter in the Gensim implementation of Word2Vec [15].

Experiment 1: Base Model. Following [20] as closely as possible, we combine Stackoverflow documents from all 4 projects into a single training corpus and report the results from training using each entire document as a complete training example for the Word2Vec model. That is, we do not split on sentence boundaries. We complete only one epoch of training.

Experiment 2: Modulating the Number of Epochs given a Large Training Corpus. For one of the projects, we report the influence of increasing the number of epochs used to train the Word2Vec model.

Experiment 3: Training only on Project-Specific Documents. For one of the projects, we train only on project-specific documents, again keeping the training examples as entire documents.

Experiment 4: Varying Training Example Construction and Number of Epochs. For one of the projects, we additionally report the interaction between the number of epochs and the method of constructing training examples.

4 EVALUATION

4.1 Metrics

- **Accuracy@k:** This is the measure of the percentage of bug reports for which we make at least one correct recommendation in the top k ranked files.
- **Mean Average Precision(MAP):** This is a standard metric widely used in information retrieval. It is defined as the mean of the Average Precision (AvgP) values obtained for all the evaluation queries:

$$MAP = \sum_{q=1}^{|Q|} \frac{AvgP(q)}{|Q|},$$

$$AvgP = \sum_{k \in K} \frac{Prec@k}{|K|}$$

  Here Q is the set of all queries (i.e., bug reports), K is the set of the positions of the relevant documents in the ranked list, as computed by the system. $Prec@k$ is the retrieval precision over the top k documents in the ranked list:

$$Prec@k = \frac{\text{\# of relevant docs in top k}}{k}$$

- **Mean Reciprocal Rank:** This measure is based on the first position ($first_q$) of the first relevant document in the ranked list. For each query q,

$$MRR = \frac{1}{|Q|} \sum_{q=1}^{|Q|} \frac{1}{first_q}$$

4.2 Results

We report the results of each experiment, and additionally compare to the performance on the same projects in [20].

Experiment 1: Base Model. In 2 we report the performance for each model. We note no apparent effect of the number of project files on the resulting scores.

While k=100 is unfeasibly high for prospective application, we find that it allows us to make comparisons between different model implementations, whereas with k=10 or lower we achieved almost null results. We find that many of the correct files are ranked around 100, with some outliers with rank positions in the hundreds, only very rarely in the thousands. It is clear that our model is at least part-way towards the goal of applicable representations, even if it does not approach the performance of Ye et al. (2016). The outliers do not seem upon random observation to be an effect of project file length, although they do seem to be somewhat related to the length of the bug report description.

We posit that these results and the discrepancy with Ye et al. (2016) are due to low vocabulary coverage. Overall, across all reports we cover only 0.22% of the vocabulary, and only 0.24% of the source file coverage. We do not have the vocabulary coverage for the replication paper, however it seems likely that it is much higher. Stackoverflow, while seemingly more appropriate in terms of task, likely contains a less-representative sample than the documents, as certain topics are more likely to be discussed online by developers using but not directly working *with* the project. Bug reports are

Project	Accuracy@k	MAP	MRR	#Files
Birt	0.16	0.05 (0.13)	0.05 (0.17)	4179
Eclipse	0.13	0.04 (0.26)	0.04 (0.31)	6495
JDT	0.16	0.02 (0.22)	0.02 (0.27)	6274
SWT	0.28	0.01 (0.25)	0.01 (0.30)	4151

Table 2: Results for the base models for each project for k=100. Numbers in brackets indicate the performance at k=10 for [20]

Configuration	Accuracy@k	MAP	MRR
SWT - 1 Epoch	0.28	0.01	0.01
SWT - 5 Epochs	0.26	0.01	0.00

Table 3: Results for the effect of the number of epochs on the SWT project

Project	Accuracy@k	MAP	MRR
SWT - All Project Data	0.28	0.01	0.01
SWT - SWT Data Only	0.17	0.01	0.01

Table 4: A comparison between all-project and single-project trained embeddings for the SWT project.

much more likely to be filed by those heavily involved with the project. Finally, our stackoverflow data heavily skews towards the last few years, while the bug report dataset ends in 2013 (all of our sample come from this). Ye et al. therefore had more of an advantage as they picked documentation from the time at which the reports were filed.

A brief literature review turned up no prior work on the necessary amount of vocabulary coverage before generating reasonable word embeddings to make feasible predictions; while perhaps this is due to the wide variety of word embedding tasks and therefore the infeasibility of generalization we still suggest that this would be a fruitful line of inquiry for its potential for evaluating prospective projects.

Experiment 2: Modulating the Number of Epochs given a Large Training Corpus. In Table 3 we compare the results on the swt project when the word embeddings are trained with only one epoch (as in experiment 1) versus 5 epochs. We note a perceptible but small difference, however, in the opposite direction expected. This may be due to overfitting or overtraining. This does not remove, however, from the need to know the number of epochs performed, given that it a parameter in a popular software package.

Experiment 3: Training only on Project-Specific Documents. In 4 we report the comparison between SWT trained as the base model versus trained on SWT-project-only Stackoverflow documents. We note that the vocabulary size of this model is 37485, as compared to 92910 when trained on all 4 projects simultaneously.

While one may expect the SWT-only results to be superior given their increased contextual relevancy this is reasonable considering the low vocabulary coverage overall; using less vocabulary may have more of an effect when one has not yet reached the minimum necessary vocabulary for reasonable performance. There is likely a point at which quality overtakes quantity, but it is not found here. Ye et al. (2016) also illustrate this point by comparing with embeddings trained on a large amount of Wikpedia data and obtaining a similar result to when they train on project documentation.

Project	Accuracy@k	MAP	MRR
SWT- 1 Epochs, Document	0.28	0.01	0.01
SWT- 5 Epochs, Document	0.26	0.01	0.01
SWT - 1 Epoch, Sentences	0.09	0.01	0.01
SWT - 5 Epoch, Sentences	0.16	0.00	0.00

Table 5: Varying training example construction and number of epochs for the SWT project

Experiment 4: Varying Training Example Construction and Number of Epochs. In Table 5 we illustrate the difference across all combinations of number of epochs and training example generating method. While it is hard to compare differences in terms of MAP and MRR, we can see a large difference in terms of Accuracy@k score, with the largest difference occurring between that of the *1 Epoch, Document* and *1 Epochs, Sentences*. We also note the improvement between *1 Epoch Sentences*, and *5 Epoch, Sentences*. Perhaps an increased number of epochs makes up for the smaller text sizes when sentence boundaries are used to generate training examples, while it does not do the same and even overfits given document-level training examples. Overall, these two variables do affect the model performance, indicating the need to report them.

4.3 Training Time

We also report training and evaluation time for the all-projects embedding (the time to train the embeddings counts for roughly two minutes of each count below, although it ordinarily would be read from a storage). We have used conventional python standards and packages, integrating multiprocessing while possible. While compute power may vary, we note that this was done maximizing a total of 40 cores and 1TB of RAM on separate machines thus indicating an average time of 22 minutes to rank files for a single bug report with a relatively robust setup. We can therefore question the feasibility of training times in application, a concern not often mentioned in related work. Coupled with low accuracy, this statistic contributed to our decision to not perform cross-validation of the model parameters.

Table 6: Time to Train and Evaluate By Project

Project	Training Time
Birt	3:19
Eclipse	2:35
JDT	1:39
SWT	2:21

5 THREATS TO VALIDITY

Internal. Internal validity concerns the influence of unknown manipulations to the data. We have already outlined how various

choices in the preprocessing steps could influence the data, and have demonstrated the results of this in experiments. We do not know how other papers have performed the processing steps that we did, including, for example, extracting the code data from the html tags of the Stackoverflow data. We do not, however, feel that this is a real threat to our validity, as part of our study design was to examine the impact of unspecified parameters in the case of study replication.

However, the dataset used is very small and thus may not allow us to generalize our results, even within the context of a replication study. We note, however, that 30 bug reports is not a completely insignificant number and we believe it still gives us enough information to characterize performance overall. It is unlikely we would have achieved comparable results to Ye et al. (2016) given more examples, although our model comparison results may change. There is still an effect on the dataset measured, however.

In addition, the corpus dataset used to train our word embeddings varies greatly in size between projects; this does not however have an effect on performance although this is confounded by the generality of the eclipse data; as the projects are all eclipse sub-projects themselves. This does conflict somewhat with the comparison between project-only and all-project training conditions, although the results concerning vocabulary size are still relevant, and show that there is an effect of these parameters on at least one dataset.

Construct. Construct validity concerns the extent to which a measure was fully able to achieve it's intended purpose. We feel that that as accuracy@k, mean reciprocal rank and mean average precision are standard measures in information retrieval tasks they allow us to achieve our goal of comparison to the target paper. Additionally, they provide an easily-interpretable result as to whether or not a developer would be able to use the generated ranked list of documents to aid in the task of bug localization. Our value of k, however, is too high to be practical for actual development use. Is does, however, allow us to make comparisons between conditions which was the point of the paper.

External. External validity concerns the applicability of the work. To this, we believe that we do not face any threat.

6 CONCLUSION

In this paper we replicated a paper close to the state of the art in the bug localization task, outlining the process of doing so and its pitfalls along the way. We reported the results of training word embeddings on a novel corpus using a previously constructed test dataset, achieving inferior results likely due to the low vocabulary coverage of our training dataset.

In addition, we showed how parameter settings can preprocessing steps can affect results. This motivates the need for researchers to thoroughly document their experiments, perhaps even explicitly sharing all model and parameter settings in an open-data format. As it matures into the use of machine learning and natural language processing techniques software engineering must standardize practice.

7 LESSONS LEARNED

A primary lesson learned is that engineering and data manipulation can take much longer than expected - so one should always try and write the most efficient code from the start in order to save on costs of re-writing. In addition one should not necessarily expect a task to be feasible if it has been written in a paper - it could very well be the case that results were left to run for weeks, or a lot of time was put into engineering. I am confidence I could speed up our current slow training times, but it would take a lot more examination of all lines of my code; I feel I wrote what I conventionally write for analyzing social media data yet I had more problems with this "small dataset". I hadn't considered the size of the files; while 11,000 files for eclipse may be considered feasible, it becomes less so if those files are also often very large.

Mutliprocessing can't be done if you're working with git, and continually checking out new repositories - this leads to race conditions and incorrect file repositories!

It is best to exhaustively document code in order to reduce explanation time for partners.

It is best to check vocabulary coverage first, although I may have said at the time that 25% was sufficient!

8 ACKNOWLEDGEMENTS

Thank you to Shruti Bhanderi for being a great partner, and to Jin Guo and my classmates for feedback.

REFERENCES

[1] [n. d.]. What's Wrong With Deep Learning, howpublished = http://www.pamitc.org/cvpr15/files/lecun-20150610-cvpr-keynote.pdf, note = Accessed: 2018-04-14.
[2] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What makes a good bug report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 308–318.
[3] Cong Chen and Kang Zhang. 2014. Who asked what: Integrating crowdsourced faqs into api documentation. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 456–459.
[4] Zellig S Harris. 1954. Distributional structure. *Word* 10, 2-3 (1954), 146–162.
[5] Eric H Huang, Richard Socher, Christopher D Manning, and Andrew Y Ng. 2012. Improving word representations via global context and multiple word prototypes. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers-Volume 1*. Association for Computational Linguistics, 873–882.
[6] Xuan Huo, Ming Li, and Zhi-Hua Zhou. 2016. Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code.. In *IJCAI*. 1606–1612.
[7] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2017. Bug localization with combination of deep learning and information retrieval. In *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on*. IEEE, 218–229.
[8] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P Midkiff. 2005. SOBER: statistical model-based bug localization. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 286–295.
[9] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. 2012. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering* 38, 5 (2012), 1069–1087.
[10] Rada Mihalcea, Courtney Corley, Carlo Strapparava, et al. 2006. Corpus-based and knowledge-based measures of text semantic similarity. In *AAAI*, Vol. 6. 775–780.
[11] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
[12] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.
[13] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. 2013. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 746–751.

[14] Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. 2012. Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow. *Georgia Institute of Technology, Tech. Rep* (2012).

[15] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA, Valletta, Malta, 45–50. http://is.muni.cz/publication/884893/en.

[16] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API documentation. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 643–652.

[17] Yan Xiao, Jacky Keung, Qing Mi, and Kwabena E Bennin. 2017. Improving Bug Localization with an Enhanced Convolutional Neural Network. In *Asia-Pacific Software Engineering Conference (APSEC), 2017 24th*. IEEE, 338–347.

[18] Xinli Yang, David Lo, Xin Xia, Lingfeng Bao, and Jianling Sun. 2016. Combining word embedding with information retrieval to recommend similar bug reports. In *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*. IEEE, 127–137.

[19] Xin Ye, Razvan Bunescu, and Chang Liu. 2014. 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*. ACM, 689–699.

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