# Predicting Source File Relevance for Submitted Bug Reports Using Machine Learning

Ben Stringer*
Student, University of Kentucky
bbst225@g.uky.edu

Jeremiah Davis*
Student, University of Kentucky
Jeremiah.Davis@uky.edu

## ABSTRACT

Natural language processing coupled with machine learning holds promise as a way of automating the process of finding source code files relevant to bugs described in user submitted bug reports. We have implemented a set of Python functions for mining and vectorizing natural language data contained in bug reports and source files, and a Python machine learning classifier which attempts to evaluate the relevancy of a particular file to a bug report. Our results indicate that this approach did not improve upon a previous learning-to-rank method. Further work is necessary to determine whether our approach can be refined in such a way as to obtain more accurate results.

**ACM Reference Format:**
Ben Stringer and Jeremiah Davis. 2020. Predicting Source File Relevance for Submitted Bug Reports Using Machine Learning. In *Proceedings of Symposium on Software Engineering (SSE 2020)*. Lexington, KY, USA, 5 pages.

## 1 INTRODUCTION

Bug reports are one of the many tools software engineers employ in the maintenance of code. They can provide developers with an outside perspective and, almost by definition, help to find bugs that test suites and code analysis missed before an application was released. Unfortunately, the process of finding and fixing reported bugs is often laborious and error prone, because the connection between reported behavior and the underlying code is not always obvious. In addition, bug reports, written in natural language, are typically unstructured and may not describe the reported behavior accurately or clearly. Some "bug reports" may in fact be feature requests or simply complaints about one or another aspect of the software. As Ye et al. note, "such a manual process in order to find and understand the cause of a bug can be tedious and time consuming" [2].

Natural language processing and machine learning provide two tools that have the potential to automate this process, at least in part, and to make bug reports into something like crowd-sourced, automated test results. The ability to take unstructured human language and convert it into a structured numeric form allows computers to work with the data in a much quicker and more thorough fashion than any human could, and in some cases to make more accurate predictions. In our research we attempted to improve on the work of Ye et al. [2], by using a machine learning classifier in place of ranking SVM and we applied that approach to small subset of the data from the Eclipse UI repository. We used Python Jupyter notebooks to implement our data mining and vectorization functions, as well as our machine learning classifier, working together in Google Colab.

As our results indicate, the machine learning classifier appears to be substantially less accurate than the learning-to-rank method employed in [2], at least for the smaller dataset we were able to classify. We describe our approach to the problem and the ways that we think we can confirm the results and possibly improve upon them. We believe that the work we have done may provides a basis for further improvements in this area, though this is not yet evident.

In Section 2 of this paper, we briefly describe [2], since it is the main paper that our own attempts to build upon. In Section 3, we describe the specific process we used to extract and vectorize our training data. In Section 4, we discuss the details of our machine learning model. Section 5 presents the details of our results. In Section 6, we cover several threats to the validity of our results. Section 7 includes our thoughts about future directions that we could take in order to create a successful machine learning classifier that will have better results.

## 2 PREVIOUS WORK

Our research is substantially an attempt to build and improve upon the work of Ye et al. [2]. In their approach, a learning-to-rank model was applied to bug reports and source code from six open-source projects: AspectJ, Birt, Eclipse Platform UI, JDT, SWT, and Tomcat. Using vectorized representations of the bug reports, source files, and individual methods in the source files, the authors computed six features to describe the relationship between a bug report and each source file. The weighted sum of these six features following scaling was used to compute a relevancy ranking between a bug report and a source code file.

In the terminology of natural language processing, the bug reports are treated as queries and the associated git commits as documents in a corpus (all of the source files in the git fix repo). The calculation of lexical similarity provides a way of determining which document is the best match for the query.

The first feature was lexical similarity, which was calculated using the maximum of the cosine similarity between a bug report and each corresponding source file and the cosine similarity of a bug report and each method in the source file. The second feature was the lexical similarity between the bug report and the source file concatenated with relevant information from the API documentation. The third feature was a collaborative filtering score calculated using the lexical similarity between a bug report and the set of all bug

*Both authors contributed equally to this research.


*SSE 2020, May 8, 2020, Lexington, Kentucky, USA*

reports for which the source file was previously fixed. The fourth feature was class name similarity, which was defined as the length of a class name explicitly mentioned in a bug report, with 0 being the default similarity. The fifth feature was bug-fixing recency, calculated using the "inverse of the distance in months" [2] between the time of the bug report and the last time the source file was changed to fix a bug. Finally, the sixth feature was a simple count of the number of times the source file was previously fixed. The six features were calculated, and then scaled so that the maximum and minimum values for each feature fall in the same range.

After calculation and scaling, the sum of the features can be used as a relevancy rank. All files were sorted in descending order on the basis of this rank. Those files which were relatively higher in the list were assumed to be more relevant to the bug report. The authors carried out an empirical study in which it was demonstrated that their approach was more accurate than simple cosine similarity between bug files and source code, a method based entirely on bug-fixing recency, as well as two "state-of-the-art" systems, BugLocator and BugScout [2].

## 3 DATA COLLECTION

In order to adhere to the time constraints of the semester's software engineering course, we chose to focus on just one of the six datasets used as benchmarks in [2], the Eclipse Platform UI. We were able to obtain a spreadsheet from Xin Ye, the principle author of [2], containing each bug report for Eclipse UI with a corresponding git commit marked "resolved fixed", "verified fixed", "closed fixed", or "resolved wontfix". Each row contains the bug fix ID, the bug fix summary, and bug fix description, together with its corresponding git commit and relevant files. Truncated timestamps are included as well as shortened git commit hashes. The provided README file that accompanied this spreadsheet suggests using the following git command to obtain the repository as it existed before the bug was fixed:

```
git checkout {hash}~1
```

The resulting git checkout can then be used to examine a pre-fix version of the relevant and irrelevant source code files. In our work, we used the PyDriller framework to interact with the git repo and mine data. This allowed us to interact with the repository directly from our Python codebase, without having to leave the Python codebase built for the project as a whole. We discuss our use of PyDriller in the following section.

### 3.1 Data mining

PyDriller is a freely-available Python framework with classes and wrapper functions for working with git repositories. Of particular interest to researchers in the RepositoryMining class, which assists a researcher in quickly iterating over thousands of commits and extracting data of interest to one's research. The GitRepository class is also very helpful for interacting with individual commits, running diffs between two commits, and in general running commands equivalent to those of the git cli command. We found using these wrapper commands more conducive to rapid development of our implementation that directly using git would have been.

After cloning the Eclipse Platform UI repository from https://github.com/eclipse/eclipse.platform.ui, we used the RepositoryMining class to find the git commits preceding the bug fix commits and the current commit at the time of the bug report itself. The latter effort may still prove to be useful if we can continue this work in the future, a matter we explain in Section 6, but we did not end up using these commits in the current study. We also used this step to derive more exact timestamps for all commits directly from the git log, as the spreadsheet rounded them to the nearest ten thousand seconds, and to retrieve the full hash. The more accurate timestamps gave us more confidence that we were really retrieving the commit that immediately preceded the bug fix. The complete hash was less useful, because it seems that the shortened hashes are collision resistant, at least for a dataset of this size, and so can be used in various git commands.

At this point we stored the updated repository information to file and began the preprocessing and vectorization stage.

### 3.2 Data processing

Our first step in preparing the bug reports and source files was to clean, tokenize, remove stopwords, and stem them using the Porter stemmer from the Python Natural Language Toolkit (NLTK), much as in [2]. We had considered using a lemmatizer as a way possibly to improve on the results, as a lemmatizer would provide more accurate "dictionary" forms for the words, but this proved too computationally intensive.

The next step was *tf.idf* vectorization. While the Python scikit-learn library provided a useful count vectorizer for changing the cleaned documents into vectors of simple word frequencies, the *tf.idf* vectorizer and transformer included in scikit-learn appear to use different methods of normalization. Since we wanted to stay as close as possible to [2] in this matter, we needed to write our own function. We were able to determine from [1] that the previous authors' approach was an implementation of "maximum tf normalization." We changed the calculation only slightly to prevent divide by zero errors, by adding one to the numerator and denominator in the *idf* calculation, as :

$$w_{t,d} = nf_{t,d} \times idf_t \tag{1}$$

$$nf_{t,d} = 0.5 + \frac{0.5 \times tf_{t,d}}{max_{t \in d} tf_{t,d}} \quad idf_t = log \frac{N+1}{df_t + 1} \tag{2}$$

In these calculations *tf* represents the simple term frequency, that is, the number of times a term appears in a document. Again, the scikit-learn count vectorizer can be used to produce this value. $N$ is the number of documents in a corpus, in our case the collection of source files in a git commit, and *df* is the document frequency, representing the number of documents in the corpus a term appears in.

Before we began to calculate the vectors for each document in each relevant commit, we needed to build a global vocabulary, $V$, containing all words from all bug reports and source files. Each document will have the same vector length, equal to the length of $V$. This ensures that we can still calculate a textual similarity between, for example, a bug report that contains a certain word that does not exist in a corpus (git commit) and the source files in that corpus.

This is where we encountered our first major problem. The function for building the global vocabulary had to process thousands of files per bug report, and we simply did not have the right combination of time and processing power in order to build the vocabulary for the over 6,000 git commits in question. At this point we made the decision to limit our study to the earliest 250 bug reports and associated commits in the provided spreadsheet. This still ultimately resulted in a full training set with 297,610 rows.

Once $V$ was built for the 250 bug reports and pre-fix commits, we were ready to vectorize the documents. This was a time consuming process that we did not want to have to repeat, so as we iterated through the bug reports and commits, we used Python pickles to save the results for each bug report to file. The count vectors were first calculated using a scikit-learn function, and then the *tf.idf* vectors calculated on the basis of these simple term frequencies. The resulting pickle files each contain a data structure that includes the full hash for the git, and, for the bug report and each Java source file, an identifying name, the count vector for the document, the *tf.idf* vector for the document, and the entire contents of the document stored as a string. Whenever possible, we reused the count vectors from previous iterations for every file that did not change between commits, which seems to have provided a modest increase in the speed of this process.

### 3.3 Computing features

Our implementation calculates the same features as those in [2], with two exceptions. In this step, we read data from the Python pickles mentioned in the previous subsection. We were not able to implement functions for parsing the individual Java files into methods. For the first feature, lexical similarity, therefore, the calculation does not include similarities between the bug reports and individual methods. We were also unable to calculate the API-enhanced similarities, again because we did not manage to implement Java source file parsing.

## 4 MACHINE LEARNING MODEL

Instead of a ranking SVM as used by Ye, et al., we chose to use a machine learning classifier to classify source files as relevant or irrelevant for a given bug report. In practice this requires an output layer with 1 cell and a sigmoid activation function, which outputs values between 0 and 1. These outputs are interpreted as "probability of relevance," and hence can be used to suggest an ordering of source files by decreasing order of probability. We opted not to use a learning-to-rank method because the exact ranking of files was not known–relevant files were not measurably more or less relevant than other relevant files.

However, the issue with choosing a deep learning classifier is its tendency to focus on learning classes that make up the biggest population. In our case, the class of irrelevant source code files vastly outnumbers the class of relevant source files. A neural network trained simply on accuracy would achieve over 99% accuracy if it simply predicted the majority label (irrelevant).

It becomes necessary to penalize false negatives (relevant bug reports that were incorrectly labeled as irrelevant) so that the model cannot simply predict that every source file it sees is irrelevant with 100% probability.

The alternative that we employed was to arbitrarily turn the problem into a regression problem and give fewer irrelevant training examples so as not to overwhelm the model with a single class of data. Rather than the 300 irrelevant files that each bug report was paired with for training in the original paper, we used only 50 for a total relevant file count of 440 and irrelevant file count of $50 \times 250 = 12500$. Hence, our training dataset size consisted of only 12,940 samples of the 297,610.

The architecture for this model was very simple: it consisted of 2 hidden layers, each with 10 neurons. After each hidden layer, batch normalization and a leaky ReLU activation was performed. The simplicity of the model helps prevent overfitting and helps the model to generalize to new data. Batch normalization and ReLU help the model converge more quickly and also help it not to get stuck.

We trained the model using Tensorflow's Sequential API and compiled it using the Adam optimizer with default parameters and the mean squared error loss function. The Adam optimizer makes use of a momentum and an adaptive learning rate. The MSE loss function computes the average squared error for each prediction in a batch compared with its ground truth value.

We allowed the model to train for 50 epochs with a batch size of 50. Due to the model's simplicity and our small dataset, the training finished in a matter of seconds.

**Figure 1: Architecture of our neural network.**

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_48 (Dense)             (None, 10)                60
_________________________________________________________________
batch_normalization_20 (Batc (None, 10)                40
_________________________________________________________________
leaky_re_lu_20 (LeakyReLU)   (None, 10)                0
_________________________________________________________________
dense_49 (Dense)             (None, 10)                110
_________________________________________________________________
batch_normalization_21 (Batc (None, 10)                40
_________________________________________________________________
leaky_re_lu_21 (LeakyReLU)   (None, 10)                0
_________________________________________________________________
dense_50 (Dense)             (None, 1)                 11
=================================================================
Total params: 261
Trainable params: 221
Non-trainable params: 40
```

## 5 RESULTS

To properly evaluate a model, especially with such a small dataset, it is necessary to withhold some data from training to ensure that the model generalizes well to data it hasn't yet seen and doesn't simply memorize the entire dataset. Of the 250 bug reports that we processed, a random 25 of them were withheld from training.

The following sections describe the results from our model when evaluated on the withheld bug reports.

These metrics show our method to be inferior to the learning-to-rank method we built upon.

### 5.1 Mean Reciprocal Rank

MRR computes identifies the best predicted ranking of a relevant file for a given bug report. It does this for each bug report, takes the inverse, and averages the result. The validation set achieved an MRR of 0.08 with a maximum possible MRR of 1.0, which occurs when a relevant file is the number one prediction for each bug report.

### 5.2 Mean Average Precision

MAP@k measures the mean of the average precisions across all bug reports. This is computed as follows

$$MAP = \sum_{q=1}^{|Q|} \frac{AvgP(q)}{|Q|}, AvgP = \sum_{k \in K} \frac{Prec@k}{|K|}$$

where Q is the set of all bug reports and K is set of predictions of the relevant files. We also have

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

The MAP@10 for the validation set was 0.177 of a maximum possible value of 1.0. Map@5 was naturally slightly better, but still only achieved 0.217.

### 5.3 Accuracy

Accuracy@k represents the portion of bug reports that had at least one relevant file in our top k predicted files.

Accuracy@10 for our validation set was 32%, and for k=20 it was 36%.

### 5.4 Training Metrics

Because a validation size of 25 is small (despite being 10% of our processed bug reports), it's also important to see how the model evaluated on the rest of the data. Note that, although the model saw samples for most bug reports, it only saw 50 of the thousands of irrelevant files in each bug report's commit. This being said, the training data resulted in the following metrics after the model was trained:

- MRR = 0.08
- MAP20 = 0.15
- Accuracy20 = 26.4%

## 6 THREATS TO VALIDITY

The largest threat to validity of our project is of course that we did not precisely compute the same features as the authors in the original learning to rank paper, omitting arguably the most important features of API enriching and individual method parsing.

Additionally, it is clear that with training on the entire eclipse project of 6000+ bug reports our results would improve, as their accuracy metrics of their model when trained individually on each of the features that we *did* compute was comparable or better than our accuracy scores.

During the data processing stage, we also noticed some limitations of the Porter stemmer provided in the NLTK library. This stemmer is somewhat naive and can produce erroneous roots for certain words. If two words are supposed to share a root and the Porter stemmer does not match them properly, they end up being treated as completely different words, since each unique word occupies a single dimension in the tfidf vectors.

Another observation that was made during the data collection and preprocessing stage was the author's assumption that the state of code when a bug report was submitted was equal to the code base of the commit that immediately preceded the one that claimed to fix the given bug report.

However, some bug reports were not fixed until years after they were initially reported, and hence the "prefix" commit for the bug report did **not** represent the state of the code when the bug report was submitted. A better method perhaps would have been to compare commit timestamps and take the most recent commit that occurred before the bug report was submitted. Even this is not guaranteed to be the state of the code that was present for that bug, because developers may be running old versions of the Eclipse software. However, it would much more closely represent that original buggy code state.

Lastly, late into the project we noticed that some of the bug reports in the original spreadsheet were marked "resolved wontfix". This would seem to indicate that these reports actually were not fixed in the corresponding commits, but this is unclear. We need to seek clarification from the authors of [2]. There were four of these bug reports in the 250 commits we used as our dataset. If they do not actually represent fixed bugs, our results may have been affected, though probably only slightly.

## 7 CONCLUSIONS AND FUTURE WORK

Although our neural network did not perform as well as the author's method of learning to rank, further data processing would have to be done to make a fair comparison. We have however shown that this problem of identifying relevant files for fixing bug reports can indeed be considered from a machine learning classification and/or regression perspective.

The question of how significantly typos in bug reports affect the ability of such algorithms to suggest relevant source files has yet to be answered, as our sample size of processed bug reports was simply too small.

The features we computed and those that the authors of the learning to rank paper suggested only relied on information that was available prior to a given bug report submission. This makes sense because when a bug report was submitted there was no knowledge of future commits or bug report solutions. However, bugs are rarely fixed immediately, and there is information to be gained from information that comes in *after* a bug report has been submitted but *before* a solution has been found.

Another point of interest in the future, particularly if one has high computational power, is to use a *lemmatizer* to perform word tokenization and vectorization. Lemmatizers are more robust (and time-expensive) to run, so they can do a better job at creating a representative vocabulary of the set of bug reports and source code files.

Lastly, the current algorithms for performing this source file suggestion reliant on natural language processing are easily generalizable to projects that the models are not trained on. A shining achievement of these methods would be if they could offer accurate

source file suggestions even without first training on years worth of bug reports and commits.

## ACKNOWLEDGMENTS

We would like to thank Dr. Xin Ye for kindly providing us with the files that helped us start our work. We would also like to thank our instructor, Dr. Tingting Yu, for invaluable advice this semester and in particular during the course of our project. We are grateful for the comments of our fellow computer science students during our midterm and final presentations.

## REFERENCES

[1] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2009. *Introduction to Information Retrieval* (online ed.). Cambridge University Press, Cambridge, England. https://nlp.stanford.edu/IR-book/pdf/irbookonlinereading.pdf

[2] 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* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 689–699. https://doi.org/10.1145/2635868.2635874