Classifying Commits as Bug-Inducing

SENG 474 Progress Report (Undergraduate)

Behl, Vibhu
vibhubehl@uvic.ca

Kaberry, James
jameslkaberry@uvic.ca

Mayall, Lucas
lucasmmayall@uvic.ca

Newcombe, Dave
davenewc@uvic.ca

Wilkes, Brennan
bwilkes@uvic.ca

March 14, 2023

1 The Problem

Śliwerski et al. [1] wrote the seminal paper on automatically identifying *fix-inducing changes*. This term would later evolve into the term *bug-inducing commits* [2]. A bug-inducing commit is a commit which introduces a bug into a software system, such that a fix, patch or revision is induced in the future. Naturally there will be an inherit similarity in the *diff* of a bug-inducing commit and it's corresponding *bug-fixing commit* [3]. The machine learning model we have under development aims to classify the bug-inducing and bug-fixing commit pairs.

Despite widespread usage in industry and numerous citations in research, even the most optimal of the SZZ variants have been shown to be faulty [4], mislabelling up to 50% of the commits studied. We continue to experiment with our proposal that the SZZ algorithm represents an excellent method for creating lists of *candidate* bug-inducing commits, but that each one of these candidates can be critically evaluated by a machine learning model, and classified *in relation* to a bug-fixing commit as either bug-inducing or not bug-inducing.

In order to test implementations of the SZZ algorithm and experiment with our own model, we've initially studied two datasets of bug-inducing and bug-fixing commit pairs. With 106,674 commits (28,239 bug-inducing and 78,435 clean commits) the *ApacheJIT* repository [5] makes up the bulk of our current example set, however pre-processing so far has only yielded approximately 1400 usable examples. We will supplement ApacheJIT [5] with the higher quality *SZZ Developer Oracle* dataset [6], which consists of approximately 1000 examples that were verified manually by a team of researchers. These datasets in order to make up the basis for our training and testing data, and to represent the *positive* training examples. (It is trivial to generate *negative* training examples, as any two random commits can be paired together.) These datasets have been processed and transformed using trivial scripts into a format which our model can train upon. Additionally with the inclusion of an unsupervised representation learning component to our model (more below), we can train the most critical portion of our model on a significantly larger dataset of unlabelled commits, which are in abundant supply and are trivial to acquire.

The problem scope has not changed significantly since the proposal of this project. The most noteworthy change is simply the condensation of effort towards the aforementioned datasets and spending effort in developing an experimental model that produces tangible output.

2 Study Goals

In the proposal, we stated that the overarching goal of this study is to contribute to the industry standard methods of identifying bug-inducing commits by using machine learning techniques to improve upon the results of the SZZ algorithm. We proposed that a classification model with a testing accuracy of *higher than 50%* would be a useful tool if integrated into an SZZ implementation. This still remains our primary goal and focus for our classification model. Beyond that goal, we would also like for our model to be as optimized as it can be. This is to say that we are striving to have a minimal amount of under-fitting or over-fitting on our model.

3 Study Plan and Progress to Date

We mentioned in our proposal that our plan was to implement a neural network with the goal of classifying if a bug-inducing and bug-fixing commit pair is valid. We had a couple of ideas of how to approach this, but we were not locked in to any particular design at that point. We were especially unsure about how exactly we would turn commit information into feature vectors. We now have a much clearer and specific idea of how we attempting to accomplish these things. All of this and more is described in the sections below.

3.1 Work Accomplished To Date

We have made significant progress on our project since the proposal. We have gone from having a concept of what we want to achieve, to having a working model we can now experiment with. This is described in detail below. We have completed the initial research and foundational work required to develop a model which compiles without error. Our goal from here is to refine, expand, and improve upon the dataset, pre-processing, and model implementation.

3.1.1 Dataset Generation and Collection

From here one of our first steps is to acquire positive training examples from a wider range of data sets, and generate higher quality negative training examples. So far we have experiemented with a few approaches for generating negative training examples. Our initial, naive approach, is to simply pair each fix up with a random commit from the same repository that is known to cause a bug. However in order to increase model accuracy we are planning to use a combination of the following techniques for generating negative examples:

1. Commits that are within n hops from the true bug-inducing commit
2. Commits from the same repo but not the same file
3. Commits made after the bug fixing commit
4. Commits that are known to be bug fixing but don't fix this bug
5. Commits that are known to be bug fixing *and* edit the same files but don't fix this bug

We hope that by exposing both our encoder and classifier to a wide range of scenarios, it will best learn the true patterns which differentiate bug-inducing and bug-fixing commit pairs.

3.1.2 Pre-processing for Commit Data

Before the model can be trained, the training and testing examples need to be processed. The main processing is done on the commit *diff*. As briefly mentioned by our proposal, we follow the steps laid out by Alon et al. [7, 8] and Lozoya et al. [9]. This process can be described as the following algorithm:

1. Create a *pre-commit* and *post-commit snapshot* of each method which has changed in a given commit.
2. Build an *abstract syntax tree* for each snapshot.
3. Compute the set of all unique *paths* (also referred to as *contexts*) between each *terminal* of each tree for each snapshot. Each of these is referred to as a *bag of contextxs*.
4. Calculate the symmetric set difference of the two bags of contexts
5. Encode each context with *one-hot encoding*.

There are a variety of parameters which can be experimented with when performing this pre-processing, which we will describe in more detail below.

3.1.3 Pre-processing for Commit Metadata

Commit metadata has been the subject of lots of internal discussion. Metadata is obviously part of a commit, but its hard to say how much impact including metadata in our model will have on our results. For this reason, we have elected to experiment with including various combinations of commit metadata before drawing conclusions.

To begin, we've decided to include information about the *commit author*, the *commit timestamp*, and the *commit message*. For the commit author, we have elected to compare the authors of each commit (bug-fixing and possibly bug-inducing) and indicate whether or not the author was the same across the two commits. This simplifies our metadata handling as our feature doesn't need to worry about the specifics of who the commit belongs to, just if the two commits have the same author. The timestamp has been handled in a similar manner, where instead of providing both timestamps to the model, we calculate the difference between the two timestamps and provide that to our model. This allows the model to only consider how big the gap between the two commits is.

The commit messages rely on a similar principle as the author names, but instead of being encoded as either a 0 or 1, it can be any continuous value between 0 and 1. This is because we compare the two commit messages and generate a similarity score between them. To accomplish this, we decided to use cosine similarity. In order to prepare the commit messages, they were first encoded as single vectors using a python library called *sentence_transformers*. Specifically, we use their *paraphrase-MiniLM-L6-V2* model, then use the cosine_similarity function from *sklearn*. This allows us to compute a score which indicates if the two given commit messages are similar or not.

3.1.4 Model Design

The majority of our time has been spent designing and iterating on our model design. While we have made significant progress, there is still plenty of work to do, as our results have been inconclusive (more below). As an overview, it's logical to break down our model into three primary components, the *encoder*, the *unsupervised clustering learner*, and the *binary classifier*. The binary classifier is the most trivial, and most high-level. It reads in two bag of contexts (one for each commit), the author similarity feature, the timestamp difference feature, and the commit message similarity feature, then encodes each bag of contexts into a fixed-length vector using the encoder, and concatenates the results. It passes this single concatenated

vector through a series of standard dense layers, and makes a prediction. It is unlikely that this portion of the model will be changed, besides modifications to the hyper-parameters.

The encoder represents the most complex and unpolished component of our model. Experimentation up until this point has been purely investigative, and as such the encoder model doesn't represent the implementation of any one paper or pre-existing design, and will need to be iterated on. The current encoder can be summarized as the following layers:

1. A masking layer to ignore padded values.
2. A 1D convolutional layer to extract features from each individual context.
3. A max pooling layer to summarize the extracted features.
4. A long term short memory layer to capture dependencies with attention.
5. A dropout layer to prevent overfitting.
6. A dense layer to converge to a given fixed length.

It is this portion of our model which we are the least confident of, and are most hoping for feedback on. We are planning to spend a significant portion of time changing components of this encoder in order to see better results (more below).

The final component of our model is an implementation of the unsupervised representation learning model *SimSiam* [10]. The purpose of this component is to allow us to train the weights of our encoder independently of our overall supervised learning problem. (And on a potentially bigger dataset.) SimSiam works by creating two random permutations of the same example, in our case randomly shuffling the order of the bag of contexts, then training the encoder such that both permutations are clustered closely together. We hope that this will allow our encoder to learn and understand commits in bags of context form independently of the final SZZ problem for which it will be used for.

3.2 Experiments Run

At this stage we have focused on model design and preliminary implementation over specific hyperparameter experiments. Most of our focus has been on developing our pre-processing techniques for the commit diffs and metadata, and designing our model. Experiments to date have been informal, with the aim of creating a model which compiles without error, as opposed to improving accuracy. Moving forward however, we have a variety of experiments planned, which we will discuss below.

3.3 Planned Experiments

There are a number of experiments we have planned. These include:

- Testing the impact commit metadata has on our model and if it is worth including.
- Experimenting with the size of each bag. (The k value defined by code2seq [8].
- Experimenting with size of the contexts themselves, which represents focusing more on longer term, or shorter term dependencies in the abstract syntax trees [8].
- Experimenting with the size of the fixed-length vector governing encoder output size.
- Using one-hot encoding, or other numerical encoding and scaling techniques.
- General analysis and corresponding experminents regarding encoder design.
- Experimenting with negative example ratio (Lucas)

- Experimenting with negative example techniques (Lucas)

More experiments will like be done as model design iteration continues. Once a reasonable model has been developed and hyperparameter experiments begin, we'll commence observation of training/test error scores and other relevant score metrics.

4 Task Breakdown

In Figure 1 we show a list of reasonable deliverables. The included deadlines not set by the instructor are tentative and subject to change. While we expect that every group member will contribute in some way to every portion of the project, we assign each deliverable a list of *primary* contributors. Our current model was trained on 1400 commits, which needs to be scaled up significantly by including other data sources. Another change will be to perform data processing in a standalone script (and then loaded from disk by the model experimentation notebook) so that the testing and training of the model is more efficient. We also need to determine an appropriate final size for the dataset, which can be supported by our existing hardware and trains in a reasonable timeframe.

Task	**Deadline**	**Primary Contributor(s)**
Adding more data sources	March 15th	Lucas, Dave, Vibhu
Adding Negative examples	March 16th	Lucas, Dave
Final Pre-processed Dataset	March 16th	Lucas
Experiment with encoder	March 18th	Brennan, Vibhu
Model Implementation	March 20th	Brennan, Vibhu
Tuned Hyper-parameters	March 28th	James, Lucas, Dave
Presentation	**March 31st**	All Group Members
Final Trained Model	April 7th	Brennan, Dave
Final Report	**April 10th**	All Group Members

Figure 1: Tentative schedule of deliverables. Bold dates represent deadlines set by the instructor.

5 Initial Results

Very recently, we achieved the first successful run though of our work in progress model. This first successful run through was achieved with a very reduced subset of the data-set that we would like to train with, but it did result in an accuracy score of 46.154%. This is currently lower than choosing at random, which is not ideal. *However*, this was training using a tiny fraction of our available data, only using 300 positive and 900 naively generated negative examples. Given that each sample has around 2^{20} features, this is not nearly enough data to properly train the model. Indeed, Alon et al. [7, 8] and Lozoya et al. [9] train on datasets on the scale of 2^{24} training examples. We are however optimistic about creating a significantly larger dataset, due to our decision to utilize unsupervised learning when training our encoder. It is also a possibility that our model is fundamentally flawed in some way, and no amount of tuning and data will allow our model to work, but we do not have enough data to determine if this is the case or not.

References

[1] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[2] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2012, pp. 104–113.

[3] M. Wen, R. Wu, Y. Liu, Y. Tian, X. Xie, S.-C. Cheung, and Z. Su, "Exploring and exploiting the correlations between bug-inducing and bug-fixing commits," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 326–337.

[4] S. Herbold, A. Trautsch, F. Trautsch, and B. Ledel, "Problems with szz and features: An empirical study of the state of practice of defect prediction data collection," *Empirical Software Engineering*, vol. 27, no. 2, p. 42, 2022.

[5] H. Keshavarz and M. Nagappan, "Apachejit: a large dataset for just-in-time defect prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 191–195.

[6] G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto, "Evaluating szz implementations through a developer-informed oracle," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 436–447.

[7] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[8] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[9] R. Cabrera Lozoya, A. Baumann, A. Sabetta, and M. Bezzi, "Commit2vec: Learning distributed representations of code changes," *SN Computer Science*, vol. 2, no. 3, p. 150, Mar 2021. [Online]. Available: https://doi.org/10.1007/s42979-021-00566-z

[10] X. Chen and K. He, "Exploring simple siamese representation learning. arxiv. org," 2020.