CHAPTER 1: INTRODUCTION

1.1. General

In most cases, when a user encounters a bug while using a system, they report the issue in a bug repository. Which typically includes a bug report containing a description of the actions required to reproduce it. If the bug has been fixed, the report may also include the name of the developer who resolved it, referred to as the "owner." This process of bug triangulation involves several actions, including assigning the bug to a developer who may potentially rectify it. In this paper, bug classification refers specifically to the task of allotting developers to bug reports. In systems with large number of incoming bugs and issues, manually going through and classification each report can be time-consuming. Most research on bug classification has focused on using the content of the bug report, including the title and description, to automate this process. However, other sources of information, such as developer profiles on GitHub and component information, have also been explored in the literature.

In our research, we propose a new algorithm for representing bug reports using a recurrent neural network (RNN(using GRU)) model. The proposed model introduces a novel approach to interpret both the syntactic and semantic relationships within data. Unlike traditional bag-of-words (BOW) features, the model utilizes bug representation to train the classifier. enabling representations to learn over longer word sequences.

To ensure a large and diverse dataset for training the learning model, the researchers leverage unfixed bug reports, which account for approximately 70% of all bug reports in an open source bug tracking system. This extensive dataset provides valuable information for the model to learn and generalize from.

By combining these innovative techniques, the model aims to enhance the understanding and interpretation of bug reports by capturing both syntactic

and semantic aspects, thereby improving the overall performance of bug classification and analysis tasks.

In addition, RNNs have been used to identify patterns in software code and make predictions about future software issues. By analyzing the code, the model can recognize patterns and anticipate potential issues. This can be used to inform development decisions and help reduce the time spent debugging and developing software.

To address these limitations, the word2vec model can be used to learn the semantic similarity between words. This model is based on the contextual hypothesis, according to which words appearing in similar contexts in a sentence have close meanings. Ye et al. used word2vec to build a shared word representation for both code language and descriptive language in bug reports. However, word2vec only considers individual word tokens, not the sequence of words in a sentence.

1.2. Bug Assignment

Previous research on bug assignment has focused on selecting developers based on their expertise to work on a single issue report. However, in practice, there are often multiple urgent bug reports that need to be addressed, and there is no reason why one should be prioritized over the others. Therefore, in this work, we define the problem statement as identifying developers to fix a number of B issues reports, assuming that D developers are available. In addition to expertise, there are also operational and economic constraints should also be taken into consideration, such as minimizing the finance of the issue-resolution process and balancing the workload of the developers. We will address this problem incrementally by developing a sequence of models that consider these constraints.

The first model emphasises on assigning only one bug to a developer. Each developer has a predefined wage, and the task is to optimize the assignment of bug reports among the developers to maximize the minimize the total wage and

expertise. The second model considers assigning multiple bug reports to each developer, up to a maximum threshold, and includes an additional cost factor for each additional assignment to a developer to balance workloads.

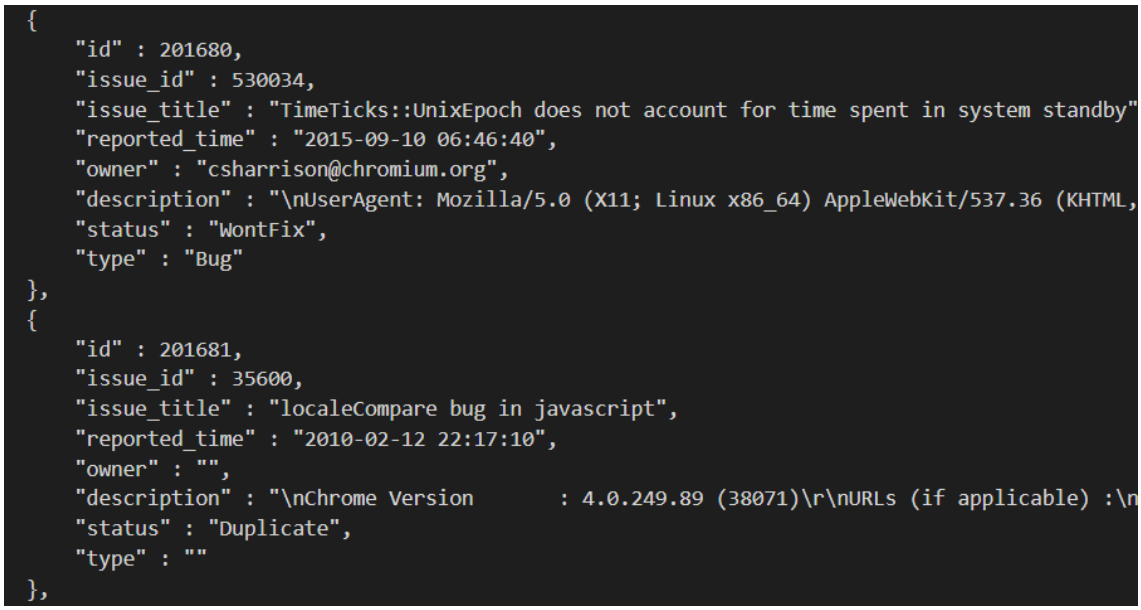

Fig 1.1: Bug Description

Getting bug reports from the chromium bug management website was not straight forward, also, there was no way to download a all the bug reports in bulk. Thus, the only option was to extract the data by look-ing at the requests the browser made while fetching the bug reports, we were then able to extract all the data in bulk by manipulating the requests made to the server and extracting the data.

The data had too many fields like attachment, target, re-view, stability which are not of any concern for bug assign-ment automation, thus we pre-processed the data using the jq' utility used for traversing JSON data. The data after pro-cessing looked alike to the data shown in fig-1.1

This approach has the potential to provide efficient and effective bug assignment solutions, overcoming the challenges associated with NP-hard problems. The use of the assignment problem framework allows us to leverage well-established optimization techniques and algorithms. By applying this

novel mapping and algorithmic adaptation, we aim to find near-optimal solutions for bug assignment.

Further research and experimentation are necessary to validate the effectiveness of this approach. Future work should focus on implementing and evaluating the proposed multidimensional classification algorithm using real-world bug assignment datasets. By comparing its performance with existing methods, we can determine the practicality and efficiency of the proposed solution.

Overall, this research contributes to the field of bug assignment by providing a new perspective and approach. By combining concepts from combinatorial optimization and software engineering, we strive to improve bug assignment processes and facilitate more efficient software development and maintenance practices.

1.3 Machine Learning Vs Deep Learning

Artificial intelligence can broadly be divided into two subfields, these are Machine learning(or ML) and Deep learning(or DL), these are used to develop algorithms and models that can learn and improve from data. Both ML and DL are used to solve complex problems and make predictions, but there are some key differences between the two. Deep learning being a subset of ML.

ML uses models and algorithms that can learn from data, but they require a lot of data to be effective. ML algorithms are designed to find patterns and relationships in data, but they need to be explicitly programmed to do so. This means that ML algorithms require a lot of effort to develop and tune, but they can be effective for a wide range of problems.

DL, on the other hand, learns from data using large and diverse neural networks made up of dense layers of nodes which may be connected to one another, they are trained using large amounts of data. DL algorithms, learn from data provided without explicitly being programmed, which means they

can be more effective for complex problems. However, DL algorithms require a lot of computational power and data to be effective.

In summary, ML and DL are two different approaches to AI that are used to solve different types of problems. ML is a more general approach that is effective for a wide range of problems, but it requires a lot of effort to develop and tune. DL is a more specialized approach that is effective for complex problems, but it requires a lot of data and computational power.

1.4 Model Overview

The model utilises three layers, each with 512 RNN units with GRU (gated recurrent units) for short-term memory re-tention. Each layer is densely connected with the adjacent layers to maximise propagation and back propagation. The first layer of this model involves word vector representation, where discrete context words are mapped to a fixed dimentional vector.
The model is divided into two main layers:

1. The model first uses a closed set of bug reports for initial training by processing the bug reports into word vectors, and then, to focus on layer 1, which involves defect reports and vector representation for whole words, in order to further enhance the confidence of the model, the open bug reports with no developer as- signed are used by extracting information from them in the form of word vectors.

2. The word2vec data processed by the first layer is then fed to the second layer; the function of this layer is to make some meaning of the vectorized data and find crosslinks between the developers and other bug reports. This layer uses GRU in place of LSTM for short-term mem-
ory retention to get a summary of the whole corpus in the form of word vectors.

CHAPTER 2: LITERATURE REVIEW

2.1. Introduction

To further describe the problem, Bug classification is the process of determining the importance and relevance of a issue in a software project. It includes evaluating the bug report, reproducing the problem, and assigning it to the appropriate team or developer for resolution. Bug classification is crucial in the software development life cycle as it helps in prioritizing bugs and check that they are addressed in a timely manner. By classification bugs effectively, developers can focus their efforts on the most critical issues, which can help in improving the overall state of the software. Bug classification is typically performed by a designated team or individual, who are responsible for evaluating the bug reports and assigning them to the appropriate developers.

2.2. Papers reviewed

This literature review will focus on bug classification using recurrent neural networks (RNNs). Bug classification is the process of categorizing and analyzing software issues to prioritize debugging and development tasks. This review will cover various aspects of the topic, including the advantages of RNNs for bug classification, their application in different scenarios, and their limitations.

We looked into a variety of articles and papers to understand how the problem is being dealt in the real world currently, All of these are mentioned in the references part of this research paper.

2.2.1 Fuzzy Logic Model

The method proposed in the works [2][3] makes thorough use of ’fuzzy sets." Fuzzy implies a mathematical framework for dealing with uncertainty and imprecision. It is a type of logic that allows for reasoning in situations where the truth values of propositions are not clearly defined or when there is incomplete information. As the name suggests, the characteristics of a bug report are stored in the form of a "fuzzy set." In order to identify a relationship between each developer and the bug report to be assigned to resolve it, A membership score is measured, which is then used to sort the bug reports and the profile of each developer. Each bug is then assigned to the acsessible developer with a matching membership score. The fuzzy subset and rules are then adjusted to better optimise the system’s speed and accuracy.

Fuzzy logic helps extract the ambiguous human logic from a body of text with the help of the member ship score without using any kind of machine learning, this also makes it much faster than any machine learning model.

The advantages of using fuzzy logic includes:
1. Simple
2. Light on resources
3. Faster
4. Easier to use and develop
5. Easier to get an understanding of the working of the model, unlike while using neural networks.

2.2.2 Traditional Machine Learning Models(NB, SVM etc.)

The method proposed in the work [4] describes well the ways in which we can automate the bug classification process with little to no sentiment analysis. They mostly only used the Naive Bayes Model in conjunction with SVM to find correlations between the bugs and the developers who were assigned those bugs. Despite using SVM and naive bayes, the highest accuracy achieved was around 30% with average precision and recall values of 28%, which was far from ideal. [5] developed a semi-supervised method of analysing the text and

using the information obtained to further improve the efficiency of supervised learning methods. They made use of the expectation maximisation (EM) algorithm, where there are two steps: In the E-step, the algorithm estimates the values of the missing or unobserved variables by calculating their expected values based on the available data and the current estimate of the model parameters.

These expected values are also known as the "responsibilities" of each data point. In the M-step, the algorithm updates the estimates of the model parameters based on the calculated responsibilities from the E-step. The new parameter estimates are obtained by maximising the likelihood of the observed data given the estimated responsibilities. This repeats until the change in the parameter estimates between iterations falls below a certain threshold; this further increased the accuracy to 36%, which was still not so powerful.

Another approach that got our attention was [6], where the researchers used multi-label KNN to find out about old bugs related to the bug reports being analysed. Then this information is used to determine the similarity between developers, and finally the KNN analysis and the developer similarity are combined to appropriately assign the bug in question. This method was able to achieve a significantly higher accuracy of 89.3%. The paper also compared other traditional machine learning approaches despite their poor efficiency, which include SVM, the Naive Bayes model, and the C4.5 model, of which the SVM model performed the best with an accuracy of 64% on the Firefox dataset.

2.2.3 Tossing Graph Model

The Tossing Graph Model (TGM) is a graph-based model used for defect assignment in software maintenance. It is based on the concept of "tossing," which refers to the process of passing a defect from one developer to another until it is resolved. The TGM represents the entire process of defect resolution as a graph, where each node represents a developer and each edge represents the tossing of a defect from one developer to another. The TGM is

used to predict the shortest path between the initial defect reporter and the final defect resolver, which helps speed up the process of defect resolution.

The TGM has been shown to be effective in improving the accuracy of automatic defect assignment compared to other machine learning-based models. It involves a series of tossing steps, where a defect is passed from one developer to another until it reaches the final fixer.

The model predicts a shorter path by identifying the most efficient path between the initial assignee and the final fixer. [6] showed that the tossing graph models outperform the traditional machine learning models (particularly naive Bayes and Bayesian models) in some or most ways. In the paper [7], the naive Bayes model is combined with the tossing graph model to automate the target assignment, and the accuracy they achieved was far better than solely using the naive Bayes models, i.e., 86%. Overall, the tossing model is a promising approach to automated defect assignment, as it can improve the efficiency of the defect repair process and ultimately lead to better software quality.

2.2.4 Team Network Model

As more and more people are collaborating digitally, it has become far easier to spot people with similar talents and skills. When people comment in chat about a bug, the discussions often hint at some correlation between other bugs or the developers fixing those other bugs. This information can be helpful in identifying the correlation between bugs and the developers of the project. In one such paper [8], the authors used a KNN model along with social network indicators to recommend a suitable defect repairer. They first search for historical bug reports that are similar to the newly reported bugs using the KNN model.

They compare the frequency and distribution of these indicators and find that the out-degree and the frequency indicators may achieve the best

results. In order to automatically assign defects, they used the naive Bayes model and SVM to get developer priority information and then analyse the bug reports to make a selection.

2.2.5 Other Custom Models

[9] developed an approach using the source code vocabulary of the bug reports and then triaged the word vectors with that information to classify the bug reports.

In the paper [10], a tool for automating the bug classification process was developed that used historical information about the changes in the source code of the project and used that information to assign the detected defects. Using this approach, they were able to achieve a significant accuracy of 81.44%. There are quite a few ways in which (custom) expert models are superior to traditional machine learning or even deep learning methodologies, primarily where efficiency is a concern.

Expert systems are computer programmes that mimic the decision-making ability of a human expert in a specific domain. These systems rely on expert knowledge and reasoning to provide advice or make decisions in a particular domain. There are several models of expert systems, each with its own approach to representing and using expert knowledge. The papers we described used the following models with differing efficiencies and accuracies:

1. Neural network expert systems
2. Fuzzy expert systems
3. Bayesian expert systems
4. Rule-based expert systems
5. Case-based reasoning expert systems

CHAPTER 3: PROPOSED WORK

3.1 Methodology

The automated bug classification algorithm is a supervised classification approach that uses the bug summary and description as input data. The major steps involved in this algorithm are shown in Figure 4.

The proposed automated bug classification algorithm has several steps, which are summarized as follows:

1. the information about the bug title, description, reported time, status, and owner of a bug is fetched from an open source bug tracking system.
2. In order to handle unstructured information like URLs, stack traces, code, hex code and snippets in the bug descriptions, the deep learning model must be trained specifically for this task. As a result, these types of content are removed during the preprocessing stage.
3. The algorithm then extracts a set of unique words that appear at least k times in the corpus as the vocabulary for the model.
4. The triaged bugs (D2) are used to train and test the classifier, while the untriaged or open bugs (D1) are used to train the deep learning model.
5. A deep bidirectional recurrent neural network with attention mechanism is used to learn a bug representation by treating the combined bug title and description as a sequence of word tokens.
6. The triaged bugs (D2) are split into train and test data using 10-fold cross validation to prevent training bias.
7. Feature representation for the training bug reports is then extracted using the trained DB-RNN algorithm.
8. A supervised classifier is trained to perform developer assignment as part of the bug triaging process.
9. The feature representation of the testing bugs is extracted using the trained deep learning algorithm.
10. the extracted features and the trained classifier then, helps the algorithm predicts a probability score for each potential developer and computes the accuracy in the test set.

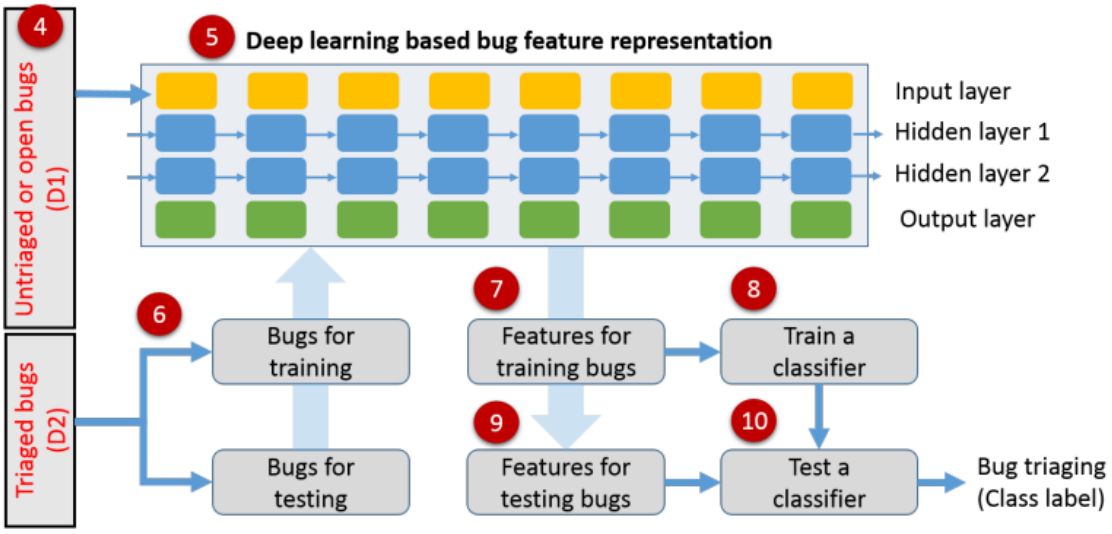

Fig 3.1. Working of the model

3.1.1 Data Preprocesssing

Getting bug reports from the chromium bug management website was not straight forward, also, there was no way to download a large number of bug reports in bulk. Thus, the only option was to extract the data by looking at the requests the browser made while fetching the bug reports, we were then able to extract all the data in bulk by manipulating the requests made to the server and extracting the data.

The data had too many fields like attachment, target, review, stability which are not of any concern for bug assignment automation, thus we pre-processed the data using the 'jq' utility used for traversing JSON data.

After processing the dataset, we pre-process the objects imported from the JSON data. As we made no use of application specific fields like hexcode, URLs, stack traces etc, we remove it from our dataset to simplify the results, we then tokenize all the objects extracted and remove the none words and the punctuations present, as the tokenizer does not make any use of it. Both open issues and the closed issues are treated the same as far as preprocessing is concerned(training and testing data should be of same shape and form).

The Below code block describes in detail how we processed the database:

```
def preprocess(title, desc):
    # Remove \r and repeated sentence
    current_title = title.replace('\r', ' ')
    current_desc = desc.replace('\r', ' ')
    for sent in removal_sent:
        current_desc = current_desc.replace(sent, ' ')
    # “Remove URLs
    current_desc = re.sub(r'http[s]?://(?:[a-zA-Z]|[0-
9]|[$-_@.&+]|[!*\(\),]|(?:%[0-9a-fA-F][0-9a-fA-F]))+',
'', current_desc)
    # Change to lower case”
    current_title = current_title.lower()
    current_desc = current_desc.lower()
    # Remove stack trace
    start_loc = current_desc.find("stack trace")
    current_desc = current_desc[:start_loc]
    # “Remove hex code
    current_title = re.sub(r'(\w+)0x\w+', '',
current_title)
    current_desc = re.sub(r'(\w+)0x\w+', '',
current_desc)
    # Tokenize sentence
    current_title_tokens =
nltk.sent_tokenize(current_title)
    current_desc_tokens =
nltk.sent_tokenize(current_desc)
    current_desc_tokens_list = [desc.split('\n') for
desc in current_desc_tokens]
```

```
    current_desc_tokens = []
    for desc in current_desc_tokens_list:
        current_desc_tokens += desc
    # Remove punctuation”
    def remove_punct(report):
        report_filter = []
        for sent in report:
            for punct in string.punctuation:
                sent = sent.replace(punct, '')
            report_filter.append(sent)
        return report_filter
    current_title_filter =
remove_punct(current_title_tokens)
    current_desc_filter =
remove_punct(current_desc_tokens)
    # “Tokenize word
    current_title_filter = [nltk.word_tokenize(sent)
for sent in current_title_filter]
    current_desc_filter = [nltk.word_tokenize(sent) for
sent in current_desc_filter]
    # Lemmatization
    def get_wordnet_pos(tag):
        if tag.startswith('J'):
            return wordnet.ADJ
        elif tag.startswith('V'):
            return wordnet.VERB
        elif tag.startswith('N'):
            return wordnet.NOUN
        elif tag.startswith('R'):
            return wordnet.ADV
        else:
            return None
    tagged_title = [nltk.pos_tag(title) for title in
current_title_filter]
    tagged_desc = [nltk.pos_tag(desc) for desc in
current_desc_filter]
    current_title_lemm =
[[WordNetLemmatizer().lemmatize(tag[0],
```

3.1.2 Word Tokenize

After removing the stopwords from the corpus during the preprocessing stage. We used word2vec library for word tokenization as it is able to represent the relationships between words in a low dimensional space in which the words which are closely related are more similar to each other compared to the sparsely related ones. This approach also overcomes the limitation of the bag of words(BOW) approach and the skip grams approach as it preserves the context information, then in order to extract the vocabulary for word2vec, we make use of the words having frequency more than k. Usually this threshold(k) is set to 10, resulting in sufficiently diverse vocabulary.

3.2 Classifying Dataset

In this research, a supervised classifier is employed to tackle the problem at hand. To evaluate the performance of the classifier, a 10-fold cross-validation model is utilized, following the approach proposed by Betternburg et al. The fixed bug reports are ordered chronologically and divided into 11 sets.

The cross-validation process starts from the second fold, where each fold serves as a test set while the preceding folds are combined to form the training set. This approach is particularly suited for open source projects where developers may change over time. By using chronological splitting, the train and test sets can be constructed to have significant overlap in terms of the developers involved. This ensures that the classifier is tested on data that simulates real-world scenarios.

By employing the 10-fold cross-validation model and incorporating developer selection criteria, the research aims to provide robust and reliable evaluations of the supervised classifier's performance in bug report assignment. This approach allows for comprehensive testing and assessment of the classifier's ability to accurately assign bug reports to appropriate developers.

3.3 Validating Results

In Order to best utilize the data we have and to get a reliable estimate of our model's perfor-mance, we cross validated using the 10 fold cross validation approach to determine the accuracy(accuracy of the model on he closed bug report database).

Fig 3.2. k-Folds Cross Validation

3.4 Comparing with current approaches

With the knowledge of the performance of various other models we reviewed during our literature review process, we were able to compile a list of all the approaches and their performance over multiper cross validation iterations.

These when compared with the best case accuracy achieved by our approach, seems to be slightly worse. The GRUs short term memory function did not seem to make a noticeable effect on the performance of our model, although the computation complexity was considerably reduces. Although the model does overfit sometimes.

Classifiers	CV-1	CV-2	CV-3	CV-4	CV-5	CV-6	CV-7	CV-8	CV-9	CV-10	Average
MNB + BOW	22	23	24	26	29	32	34	36	39	38	30.4
Cosine + BOW	18	22	25	28	29	31	34	36	37	38	29.8
SVM + BOW	19	17	15	18	21	19	20	22	23	22	19.6
Softmax + BOW	17	13	13	14	12	12	12	12	13	13	13.1
Softmax + DBRNNA	39	37	40	44	45	47	51	53	54	56	46.6
This Model	51	43	44	42	48	46	55	55	53	59	49.6

Table 3.1. Accuracies compared across other approaches

CHAPTER 4: FINDINGS

4.1 Ten Fold Cross Validation(CV)

10-fold cross-validation is a commonly used technique in machine learning and model evaluation. It involves dividing a dataset into 10 equal-sized subsets or folds. The process can be summarized as follows:

1. The dataset is randomly shuffled to remove any potential ordering effects.
2. The dataset is then divided into 10 equal-sized folds.
3. For each fold, it is held out as a test set, while the remaining 9 folds are combined to form the training set.
4. The machine learning model is trained on the training set and evaluated on the test set.
5. The performance metrics, such as accuracy, precision, recall, or others, are recorded.
6. Steps 3 to 5 are repeated 10 times, each time using a different fold as the test set.
7. The performance metrics from the 10 iterations are averaged to obtain a more robust estimate of the model's performance.

The **benefit of 10-fold cross-validation** is that it provides a more reliable estimate of the model's performance compared to a single train-test split. It helps to assess the generalization ability of the model by evaluating it on multiple independent subsets of the data. Additionally, it allows for better utilization of the available data, as each instance is used in both the training and testing phases.

4.2 Results

Our model was able to achieve a substantial accuracy with

the training data, as can be seen by the output of 10 fold cross validation below:

```
Top10 accuracies for all CVs: [96.24554005838469,
 91.16966793325624, 91.02938788764409,
 96.05220492866408, 95.22690437601297,
 95.74150787075394, 91.23222748815166,
 82.29234263465283, 83.13972513089006,
 79.65756216877293]
Average top10 accuracy: 90.17870704771835
```

Fig 4.1. Achieved Accuracy

but the actual accuracy with the novel test dataset took a sig-nificant hit(like in most other papers we reviewed). Which the figure 2. depicts accurately.

The accuracy of our model in comparison to other models using same or similar dataset can be seen below, we can thus, infer that increasing the number of neurons in the densely linked layers did have an impact on the overall accuracy. Further, our decision to use gated recurrent units instead of LSTM did not seem to have an impact on accuracy, although it did slightly reduce the resources required for computation.

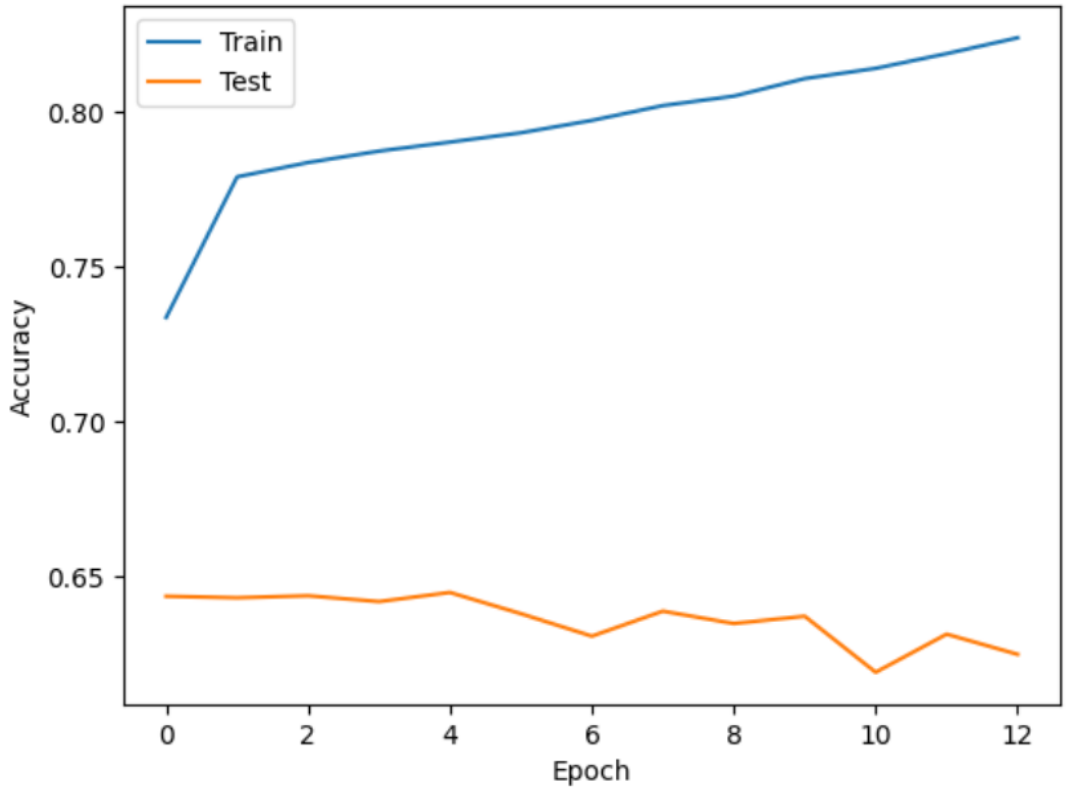

Fig 4.2. Training vs Testing accuracy through epochs

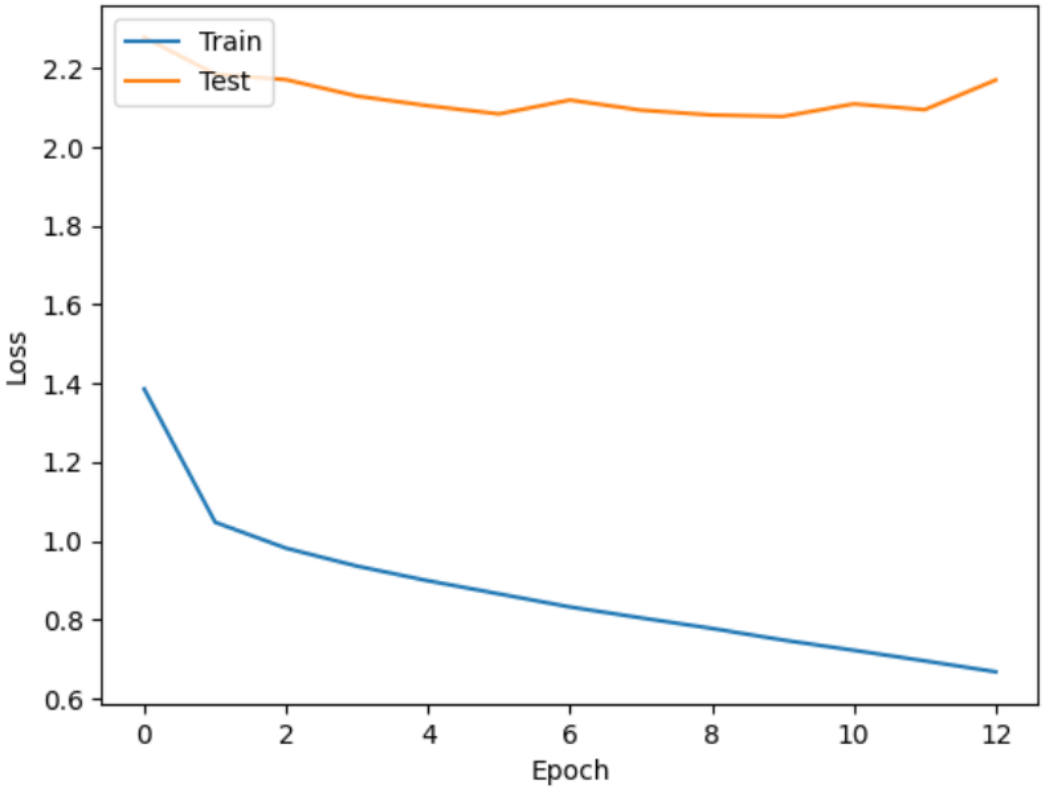

Fig 4.3. Training vs Testing loss through epochs

4.3 Conclusion

In this paper we discussed an approach to the problem of bug assignment to developers,
one solution of which was proposed by [1],
we work upon this solution to further enhance the accuracy of the model.

The major modifications made by us include :

1. Using gated recurrent units in place of LSTM for faster execution.
2. Increasing the number of neurons in the densely linked layers to increase the performance ceiling of the model.

The GRUs short term memory function did not seem to make a noticeable effect on the performance of our model, although the computation complexity was considerably reduces.

Further, the increased number of densely linked neurons were able to achieve slightly better performance, so much so that it outpaced our reference model, few times we ran the model.
the one mentioned in the table 1 is the best we were able to achieve through our ten fold cross validation method
In addition to improving the efficiency of the bug fixing process, using RNNs for bug classification has several other benefits. First, RNNs can handle large amounts of data and can be easily scaled to handle the increasing number of bugs reported in modern software systems. This allows developers to triage more bugs in less time, and to focus their efforts on the most important bugs. Second, RNNs can learn from past classification decisions and can improve their performance over time. This means that the model can become more accurate and reliable as it is exposed to more data.

Third, RNNs can provide more consistent and objective classification decisions compared to manual classification. This can help to reduce subjectivity and bias in the classification process, and can improve the overall quality of the triaged bugs.

Furthermore, using deep RNNs for bug classification can also provide valuable insights and information about the bugs and the software system. For example, the model can identify common patterns and trends in the bugs, which can help developers to better understand the root causes of the bugs and to prevent similar bugs in the future.

Overall, the use of deep RNNs for bug classification is a promising approach that has the potential to improve the efficiency and reliability of the bug fixing process in modern software systems.

REFERENCES

[1] S. Mani, A. Sankaran, and R. Aralikatte, Deeptriage: Exploring the effectiveness of deep learning for bug classification, *Corr, vol. abs/1801.01275, 2018, Available*

[2] R. R. Panda and N. K. Nagwani, Classification and in-tuitionistic fuzzy set based software bug classification tech-niques, *Journal of king saud university - computer and information sciences, vol. 34, no. 8, Part B, pp. 63036323, 2022,*

[3] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, Fuzzy set and cache-based approach for bug classification, in *Proceedings of the 19th acm sigsoft sympo-sium and the 13th european conference on foundations of software engineering, in Esec/fse 11. Szeged, Hun-gary: Association for Computing Machinery, 2011, pp. 365375.*

[4] D. Cubranic and G. C. Murphy, Automatic bug triage using text categorization, *in International conference on software engineering and knowledge engineering, 2004.*

[5] J. Xuan, H. Jiang, Z. Ren, J. Yan, and Z. Luo, Auto-matic bug triage using semi-supervised text classifica- tion, *Arxiv, vol. abs/1704.04769, 2017.*

[6] G. Jeong, S. Kim, and T. Zimmermann, Improving bug triage with bug tossing graphs, *in Esec/fse 09, 2009.*

[7] P. Bhattacharya and I. Neamtiu, Fine-grained incre-mental learning and multi-feature tossing graphs to im-prove bug classification, *2010 ieee international conference on software maintenance, pp. 110, 2010.*

[8] *W. Wu, W. Zhang, Y. Yang, and Q. Wang,* Drex: Devel-oper recommendation with k-nearest-neighbor search and expertise ranking, *2011 18th asia-pacific Software engineering conference, pp. 389396, 2011.*

[9] D. Matter, A. Kuhn, and O. Nierstrasz, Assigning bug reports using a vocabulary-based expertise model of developers, in 2009 6th ieee international working conference on

mining software repositories, 2009, pp. 131140.

[10] John Anvik, Lyndon Hiew, and Gail C Murphy. Who should fix this bug? In *International Conference on Software Engineering, pages 361–370, 2006.*

[11] John Anvik and Gail C Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology, 20(3):10, 2011.*

[12] Ali Sajedi Badashian, Abram Hindle, and Eleni Stroulia. Crowdsourced bug classification. In *International Conference on Software Maintenance and Evolution, pages 506–510. IEEE, 2015.*

[13] Pamela Bhattacharya and Iulian Neamtiu. Fine-grained incremental learning and multi-feature tossing graphs to improve bug classification. *In International Conference on Software Maintenance, pages 1–10, 2010.*

[14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078, 2014.*

[15] Alan Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. *In International Conference on Acoustics, Speech and Signal Processing, pages 6645–6649, 2013.*

[16] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Mur- phy. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. *In International Conference on Software Engineering, volume 1, pages 495–504, 2010.*

[17] Ashraf M Kibriya, Eibe Frank, Bernhard Pfahringer, and Geoffrey Holmes. Multi-nomial naive bayes for text categorization revisited. *In AI 2004: Advances in Artificial Intelligence, pages 488–499. Springer, 2004.*

[18] Quoc V Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International Conference on Machine Learning, volume 14, pages 1188–1196, 2014.*

[19] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025, 2015*.

[20] Rada Mihalcea, Courtney Corley, and Carlo Strapparava. Corpus-based and knowledge-based measures of text semantic similarity. *In AAAI, volume 6, pages 775–780, 2006.*