The Journal of Systems & Software 184 (2022) 111133

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Multi-triage: A multi-task learning framework for bug triage[☆]

Thazin Win Win Aung [a,*], Yao Wan [b], Huan Huo [c], Yulei Sui [d]

[a] University of Technology Sydney, Australia
[b] Huazhong University of Science and Technology (HUST), China
[c] Faculty of Engineering & Information Technology, University of Technology Sydney, Australia
[d] School of Computer Science, University of Technology Sydney, Australia

ARTICLE INFO

Article history:
Received 28 December 2020
Received in revised form 18 October 2021
Accepted 19 October 2021
Available online 3 November 2021

Keywords:
Bug triage
Recommendation system
Multi-task learning
Deep learning

ABSTRACT

Assigning developers and allocating issue types are two important tasks in the bug triage process. Existing approaches tackle these two tasks separately, which is time-consuming due to repetition of effort and negating the values of correlated information between tasks. In this paper, a multi-triage model is proposed that resolves both tasks simultaneously via multi-task learning (MTL). First, both tasks can be regarded as a classification problem, based on historical issue reports. Second, performances on both tasks can be improved by jointly interpreting the representations of the issue report information. To do so, a text encoder and abstract syntax tree (AST) encoder are used to extract the feature representation of bug descriptions and code snippets accordingly. Finally, due to the disproportionate ratio of class labels in training datasets, the contextual data augmentation approach is introduced to generate syntactic issue reports to balance the class labels. Experiments were conducted on eleven open-source projects to demonstrate the effectiveness of this model compared with state-of-the-art methods.

1. Introduction

Software issue reports—i.e. feature enhancement requests and bugs that appear during software maintenance—are typically stored in bug repositories or issue tracking systems (Hassan and Xie, 2010). Many open-source software projects predominately use cloud-based issue tracking systems (e.g. Bugzilla, GitHub) to manage requests systematically (Yadav et al., 2019). The process of managing an issue tracking system involves reviewing new issue reports to ensure they are valid (thus eliminating duplicate reports), finding appropriate developers for assignment, and classifying each issue into the relevant issue type (e.g. bug, feature, and product component). The process is also known as bug triaging, and a person who performs these tasks is called a triager or issue tracker (Banerjee et al., 2017). In practice, an issue tracker manually performs this process repeatedly. Bug triaging is thus time-consuming and tedious, since many software projects are maintained by multiple developers and composed of various product components. In some scenarios, if the assigned developers cannot fix the issue, the issue report is reassigned to another developer; this reassignment process is widely known as bug tossing. This tossing process can add to the overall bug fixing time.

Problem. As large numbers of bugs are reported daily in the issue tracking system, manually managing these issue reports on time becomes challenging. For instance, in the `aspnetcore`[1] project, over the course of six months (from Jan 1, 2020 to Jun 30, 2020), 1339 issue reports were reported, with an average of 223 reports per month. The project is maintained by 84 developers, and with each report being classified as one of 197 issue types, an issue tracker needs to spend a lot of time and effort on triaging. As a consequence, this might delay resolving these issue reports. Several automatic triage approaches have been proposed to leverage the candidate developers' prediction process (Anvik et al., 2006; Kagdi et al., 2012; Lee et al., 2017; Xia et al., 2016a; Xi et al., 2018; Mani et al., 2019; Xi et al., 2019) and issue type (Runeson et al., 2007; Wang et al., 2008; Banerjee et al., 2017) labelling process.

In general, existing bug triage approaches mainly fall into two categories: the algebraic model-based approach and the statistical language model-based approach. Both approaches train both developers and issue types prediction tasks with a single task learning model. Studies have used terms frequency (TF) and inverse document frequency (IDF) as the term's weighting factor in

☆ Editor: Alexander Serebrenik.

* Correspondence to: University of Technology Sydney, 15 Broadway, Ultimo, Sydney, NSW, 2007, Australia.
E-mail addresses: thazinwinwin.aung@student.uts.edu.au (T.W.W. Aung), wanyao@hust.edu.cn (Y. Wan), Huan.Huo@uts.edu.au (H. Huo), Yulei.Sui@uts.edu.au (Y. Sui).

1 https://github.com/dotnet/aspnetcore/.

https://doi.org/10.1016/j.jss.2021.111133

algebraic models. Various distance calculation algorithms (e.g. Euclidean distance) are used to calculate the distance between two issue reports and to construct links between a new issue report and potential developers or issue types via matching with existing issue reports (Runeson et al., 2007; Xia et al., 2016a).

The most commonly used algebraic models in these studies are: the vector space model (VSM), latent semantic indexing (LSI), and latent dirichlet allocation (LDA) (Xia et al., 2016b,a; Yadav et al., 2019; Xi et al., 2019; Anvik et al., 2006; Mani et al., 2019; Lee et al., 2017). More recent studies have explored statistical machine learning representation models, such as support vector machine (SVM) (Anvik et al., 2006) and, neural language models, such as convolutional neural networks (CNNs) (Lee et al., 2017), recurrent neural networks (RNNs) to leverage accurate learning representations. However, the existing approaches capture both an issue report's description and code snippet information as continuous distributed vector representations, as code snippets' properties are not captured precisely. In addition, the performance of these learning models can be degraded due to data imbalances in the training data (Lee et al., 2017).

Limitations. To leverage the existing bug triage approaches, the following limitations are addressed in the present study.

Limitation 1: It is time consuming to train multiple single-task learning models individually. Recommending developers and issue types are two important tasks of the bug triage process. Existing methods solve these two tasks separately, which leads to task repetition and ignores the correlating information between tasks. In a software development project, some developers normally work on certain components (e.g. user interface module, API components). Thus, developers and issue types are two closely related attributes. However, this correlation is not adequately considered in existing bug triage models. First, both developers and issue types labelling can be regarded as a classification problem: both rely on historical bug descriptions and code snippet information. Second, these two tasks can benefit each other: developer selection can incorporate additional knowledge from issue types labelling, while learning these two tasks together can be improved by learning textual information and abstract syntax tree (AST) information from the issue reports.

Limitation 2: There is a lack of structural information of code snippets in feature representation. Most issue reports contain code snippets written in the structural language. Code snippets are error-prone, as they cannot parse into AST structure directly without pre-processing. Neither learning the code snippets together with the bug description nor negating them can perform the issue report representation learning effectively.

Limitation 3: There is class imbalance. Each issue report can be linked to multiple developers and issue types. The disproportionate ratio of observations in each label, leads to classification predictive modelling problems. Most studies have addressed the imbalanced labels challenge by using a minimum threshold approach to filter out the inactive labels. However, this approach constrains model prediction to these labels.

Contributions. The main contributions of this papers are summarised as follows:

- A multi-tasking bug triage model is proposed to recommend a list of developers and issue types most relevant to a new issue report.
- A precise issue report feature representation approach is proposed. In this approach, the text description and code snippets context are split into two separate tokens to reduce noise when learning the representations.
- A contextual data augmentation approach is used to generate synthetic issue reports to over-sampled imbalanced datasets, thereby increasing model accuracy.
- Open-source projects from eleven different domains were extensively evaluated. The present multi-triage model is compared with baseline approaches and two single-task learning models (i.e. developers and issue types) to measure this model's benefits in terms of training time and accuracy.

Research questions. This paper focuses on answering the following three research questions to address the significance of the study.

RQ1: Does the multi-triage model outperform two existing approaches in terms of accuracy? First, whether the proposed multi-triage model outperforms the existing approaches is studied.

RQ2: Which component contributes more to the multi-triage model? This question focuses on performing ablation analysis on the multi-triage model to identify which of its components are essential to optimise model performance. Next, the multi-triage model is compared with the conventional single task learning model in terms of time and accuracy.

RQ3: Does increasing the size of training datasets based on the contextual data augmentation approach improve our model's accuracy? *In this paper, a contextual data augmentation approach is introduced to increase the size of the training sets to leverage the multi-triage model's accuracy. To evaluate the effectiveness of the augmentation approach, the multi-triage model is trained with two sets of training data (i.e. with and without augmented data) and the accuracy of the outputs is compared.*

Organisation. The remainder of the paper is organised as follows. Section 2 introduces background information on the bug triage process, and the motivating example. Section 3 presents the overall framework of this study' approach. Section 4 describes the research questions and implementation. Section 5 provides the evaluation results. Section 6 discusses validity threats. Section 7 describes the significance of our findings. Section 8 reports the related work while Section 9 provides conclusions.

2. Background and definitions

This section discusses background information about the correlation between developers and issue types recommendation tasks as well as their usages in the issue report and pull-based development projects. Then, we present our motivating example.

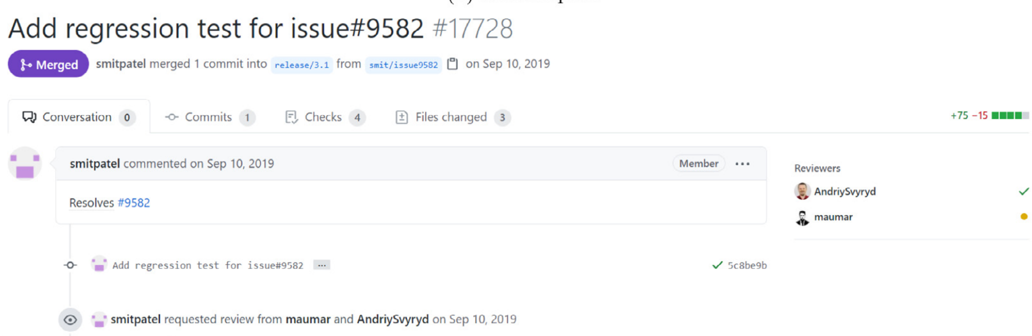
2.1. Developers and issue types recommendation tasks in bug triage

Assigning developers and allocating issue types are two essential tasks in the bug triage process. In the issue tracking system, an issue tracker normally performs these two tasks as the first step in the bug triage process. Our multi-triage recommendation model predicts relevant developers and issue types for a new issue report to leverage the bug triage process. In this context, issue reports include both bug and enhancement-related issues. Our recommendation model performs two tasks, as below.

Developer recommendation task. This task involves predicting the list of potential developers to fix a new issue report. Sometimes, the issue report is fixed by more than one developer, due to its complexity.

Issue type recommendation task. This task involves predicting the list of issue types to categorise a new issue report. For example, GitHub's issue tracking system provides seven generic labels (i.e. bug, duplicate, enhancement, help wanted, invalid, question, and won't fix), but can add a new custom label as needed (Cabot et al., 2015a). Interestingly, most projects create custom labels to track issue priority (e.g. high, low), product version (e.g. 2.1), workflow (e.g. backlog, review), and product components (e.g. area-identity, area-mvc, area-blazer).

(a) Issue report



(b) Pull request

Fig. 1. An example of an issue report and the corresponding pull request.

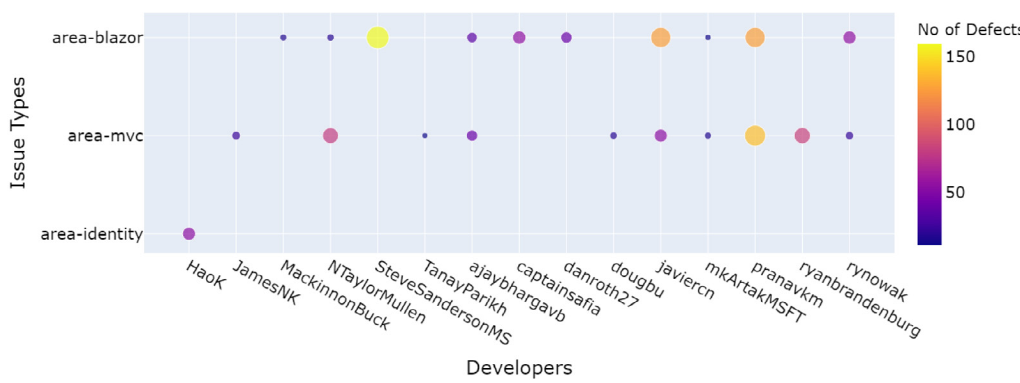


Fig. 2. Developers and issue types correlation example.

Issue report and corresponding pull request. Fig. 1 presents an example of the GitHub issue report 1(a) and its corresponding pull request 1(b). Recent years have seen a growing interest in pull-based development in open-source software projects (Yu et al., 2016; Gousios et al., 2014; Jiang et al., 2020). In a pull-based model, a developer uses a pull request form to submit code for request code review. The reviewers are usually project owners or contributors who make the final decisions on the requested changes (i.e. reject, merge, or reopen). In the GitHub project, the fields contained in the pull request form are similar to those in the issue request form but also include additional sections, such as reviewers, a commits tab, a checks tab, and a files changed tab. In the description field, most projects reference the fixed issue IDs for traceability. The reviewers field contains the list of reviewers who review the changes, while the commits tab contains the commits hierarchy, and the checks tab presents the detailed build outputs. Lastly, the files changed tab displays the list of changed files from all the commits. During initial observations, it was learned that a developer allocated on the issue report may be different from a developer who created the pull request to fix the issue. Therefore, this study considered that the developer information from the pull request is non-trivial in the labels construction process.

Developer and issue type correlation. In existing projects, both developers and issue types recommendation tasks use historical issue reports to train the prediction model. Therefore, there is a common learning representation layer between these two tasks, which can learn together. Also, as a software project involves various components (e.g. user interface, database, application programming interface), an issue report can relate to any part of the system. Consequently, certain issue types are usually assigned to a group of developers with expertise in certain system areas. The recent work of Catolino et al. (2019) highlighted that not all bugs are the same, and the structure of project teams is based on the components of a system. Fig. 2 presents a simple example

Table 1
AST nodes terms and abbreviation.

Terms	Abbreviation
Method Declaration	Md
Parameter	Para
Block statement	Bst
For statement	Fst
Expression statement	Est
Method call expression	Mce

in which developers focus on fixing particular system areas. This example was extracted from the aspnetcore.[2]

2.2. Motivation

As mentioned earlier, the previous bug triage approaches have considered developers and issue types prediction tasks as independent tasks and trained separately for each. Therefore, it is time-consuming to train the model. In addition, in existing approaches, code snippets are either excluded (Mani et al., 2019; Xi et al., 2018) to reduce noise, or treated as natural language sequence tokens (Lee et al., 2017; Xia et al., 2016a). Thus, these approaches cannot learn the code snippets or representations precisely. In initial observations, the issue reports characteristics of eleven open-source projects from various domains were investigated, including a web application, unit testing, entity development, programming interface, compiler, mobile app, augmented reality, gaming, and search engine to configuration. Further the information is presented in Table 2.

These GitHub projects were selected based on popularity (i.e. rating) and active activity (i.e. recent commits). Projects with a high number of contributors and issue types labels were also considered in order to identify the gap in the existing approaches to leverage the bug triage process. Eclipse issue reports, which are used in baseline studies to compare this study's approach and the-state-of-the-art-approach, were also included. Eclipse issue reports were extracted from the Bugzilla issue management system. However, Bugzilla[3] does not keep developers' tossing sequences as this study did not present the average tossing sequence, value for the eclipse project in Table 2.

Code snippet. The percentage of issue reports include method-level code snippets were analysed to reproduce the problem. Interestingly, as presented in Table 2, 12 to 20 per cent of issue reports contain code snippets. Recent studies (Alon et al., 2019; Kim et al., 2021) have found that learning representations of AST tokens are more effective than simple code-based tokens in various code prediction tasks (i.e. code translation, code captioning, code documentation). Inspired by previous studies, these code snippets are transformed into AST paths and a separate token is created according to this approach. Fig. 3 shows an example of a java code snippet 3(a) and its corresponding AST 3(b), where a node (i.e. Para, Bst, Fst, Est, and Mce) is a terminal node, and the rest are non-terminal nodes. In this approach, the code snippets are complied using Eclipse IDE[4] for java code snippets and Microsoft visual studio IDE[5] for C# code snippets. Table 1 presents the abbreviation for each of the AST node terms. Then, the code snippets are parsed into AST using Java and C# extractor from the code to sequence the representation approach (Alon et al., 2019). Implementation details are presented in Section 4.

Issue reassignment. In the GitHub project, it is noted that a single pull request can include fixes for multiple issue reports, and a developer who fixes the issue may be different from the assigned developers recorded in these issue reports. In the context of bug triage, this process is normally referred to as tossing (Xi et al., 2018). On average, 368 cases in within the training projects are classified into reassigned issue reports. These pull request developers' details are included in the labels construction process, in order to include the issue report's tossing sequence.

2.3. Multi-task learning

In recent years, MTL has been successfully applied in many areas, including computer vision (Kokkinos, 2017; Dvornik et al., 2017; Bilen and Vedaldi, 2016; Zhou et al., 2017), natural language processing (Liu et al., 2015), and facial recognition (Zhang et al., 2014). It seems, however, that MTL has not been applied to modelling the bug triage process. In this paper, the MTL model is adopted to improve the performance of the bug triage process. MTL tackles developer and issue type recommendation tasks simultaneously by sharing learning parameters to enable these tasks to interact with each other. Joint learning of these two tasks significantly improves the performance of each task, compared to learning independently. The multi-task learning model can share parameters between multiple tasks with either hard or soft parameter sharing of hidden layers. The hard parameter sharing model explicitly shares the common learning layers between all tasks while branching the task-specific output layers (Caruana, 1993). The soft parameter sharing model, meanwhile, implicitly shares the parameters by regularising the distance between the parameters of each task. Although both approaches can be viewed as the underlying architecture of the multi-task learning model, hard-parameter sharing is commonly applied in the context of the neural network.

This multi-task learning model uses the hard-parameter sharing approach to learn the issue report representation in the common layer and then branch the two task-specific output layers to predict developers and issue types. In the common layer, the individual issue report are further subdivided into two categories, namely (1) natural language and (2) structural language, to learn the representation effectively. An issue title and description, excluding code snippets are grouped under natural language, whereas code snippets are placed under structural language. Then, two encoders are used, namely (1) context encoder and (2) AST encoder, to extract the essential features of these two contexts. Next, these two features are combined and fed into the task-specific output layers to perform co-responding classification tasks. The detailed implementation of this approach is explained in Section 3.

3. Proposed approach

This section first explains the high-level structure of the multi-triage framework. Next, it presents the integral components of the multi-triage model.

[2] https://github.com/dotnet/aspnetcore the GitHub project. The x-axis represents the developers, whereas the y-axis represents the system areas. The size of the bubble indicates the total issues fixed by developers in the corresponding areas. Referring to the example, a handful of potential developers can fix area-blaze and areas-mvc issues. However, there is one developer (Haok) who is capable of resolving area-identity issues. Based on this observation, we are motivated to seek the effect of the learning developer and issue types jointly in the multi-task learning model.

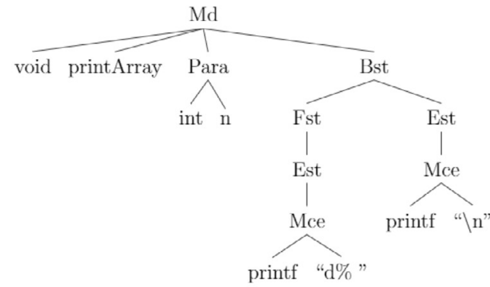
[3] https://bugs.eclipse.org/bugs/.

[4] https://projects.eclipse.org/projects/eclipse.platform/.

[5] https://visualstudio.microsoft.com/.

```
//Utilizing function to print the Array
void printArray(int arr[], int n)
{
  for(int i = 0; i < n; i++)
    printf("%d  ", arr[i]);
  printf("\n");
}
```

(a) Code snippet



(b) AST

Fig. 3. A code snippet and corresponding AST example.

Table 2
Raw datasets information.

Name	Period	#No	#Code	#Dev	#Types	#Tossing	#Days
aspnetcore	10/2014 −10/2020	7151	2520	60	131	62	45
azure-powershell	01/2015 09/2020	2540	312	386	204	128	82
eclipse	10/2001 05/2021	50806	6320	21	621	–	60
efcore	01/2015 - 09/2020	6612	1650	24	57	2293	76
elasticsearch	01/2015 - 10/2020	5190	1504	104	238	178	92
mixedrealitytoolkit-unity	03/2016 - 09/2020	2294	70	55	124	53	71
monogame	01/2015 - 09/2020	1008	110	4	28	22	21
nunit	10/2013 - 09/2020	656	70	27	24	130	63
realm-java	05/2012 -10/2020	1160	340	15	23	400	69
roslyn	02/2015 - -09/2020	5093	1300	79	123	300	100
rxjava	01/2013 - -09/2020	2076	610	5	32	121	44
						Avg (368)	Avg (67)

3.1. Overview

Fig. 4 presents the overall structure of the multi-triage framework. This framework includes three main components: (1) data extraction, (2) a contextual data augmenter, and (3) the multi-triage model. In the *data extraction component*, ground truth links are constructed between issue reports and multi-labels (i.e. developers and issue types).

3.2. Data extraction

The data extraction component includes two sub-components: the text extractor and the AST extractor. The *text extractor* component concatenates each issue report's title and description into one text token, excluding the code snippet information. The *AST extractor* parses each code snippet and constructs the AST paths. An AST or syntax tree has two types of nodes: terminal and non-terminal. The terminal node represents user-defined values (e.g. identifiers), whereas the non-terminal node represents syntactic structures (e.g. variable declarations, a for loop) (Alon et al., 2019). An AST path is the sequences of the terminal and non-terminal nodes.

In this paper, Eclipse and Microsoft visual studio IDE were used to compile the code snippet before passing it to the AST extractor. The AST generator tool from Alon et al. (2019) is used to construct AST paths, using the default parameters settings (max child node = 10, max path length = 1000, and max code length = 1000). In any issue report, a single code snippet can contain multiple methods as the generator is modified (Alon et al., 2019) by adding '⟨*BM*⟩' and '⟨*EM*⟩' separator tags between each method for model learning purposes.

Fig. 5 presents the data extraction steps for a single issue report seen in Fig. 1(a). First, the issue report's title and description are concatenated. Next, the code snippet is compiled and parsed into AST paths. The AST paths are generated by pairing all the dependent nodes and using the ';' separator between a pair to indicate a path. Next, multiple developer labels are created by using the '|' separator. In the developer labelling process, a pull request creator account is included if the developers allocated in the issue report do not include a pull request creator account. Finally, the issue type label is constructed by using bug or enhancement and system components format using the same '|' separator.

3.3. Contextual data augmenter

In the *contextual data augmenter*, synthetic issue reports are created for each project using the approach presented in algorithm 1. The algorithm's input is the list of training issue reports

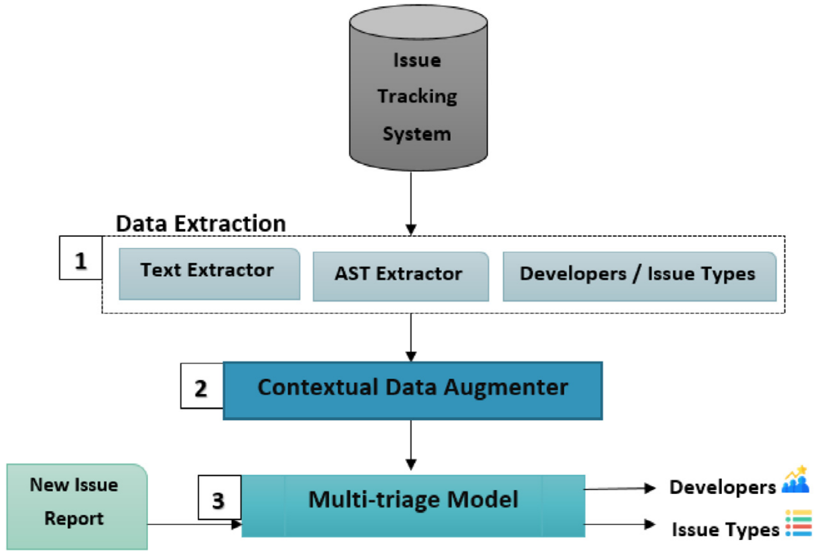


Fig. 4. The multi-triage framework.

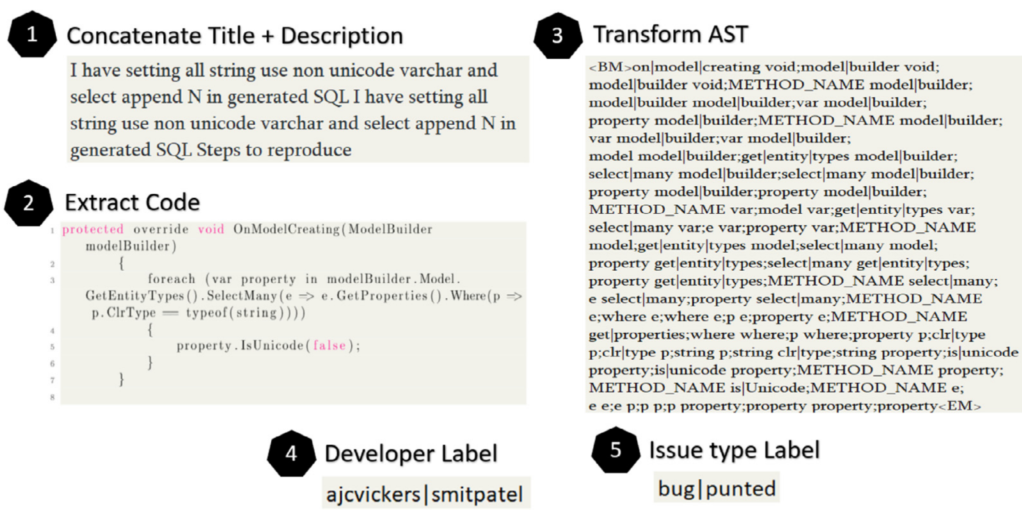


Fig. 5. An example data extraction steps for an issue report.

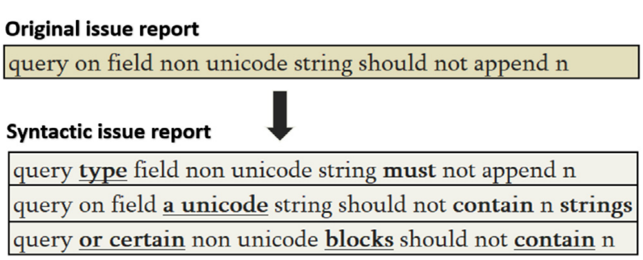


Fig. 6. A synthetic issue report example.

and the *Threshold* to generate synthetic records. In this approach, a new record is created based on the training datasets and the generation of synthetic records is limited by using the *Threshold* parameter. In this experiment, a *Threshold* value of 30,000 is used to control the total number of data augmentation records. The threshold is calculated based on the approximate total number of issue reports from target projects. However, it is a hyper-parameter value and can change as needed. First, it initialises the values with the majority and minority class details (lines 1 to 3). It creates the clusters by grouping with developer and issue type labels. After initialisation, $MinC * MajC$ are multiplied to calculate the estimated number of synthetic records to compare with the *Threshold* amount (line 4). If the estimated value is larger than the *Threshold*, then it calculates the new majority class count value for an adjustment (lines 5 to 7). Next, it iterates through each minority class to generate a synthetic record (lines 8 to 16).

In each iteration, it randomly retrieves an issue report description (excluding code snippet) of the current minority class. Then, it substitutes 15% of the words in the description with the new words using the contextual data augmentation approach proposed by Kafle et al. (2017) and creates a new issue report. In this experiment, the BERT-base-uncased pre-trained model[6] was used, which trained with a large corpus of English data to predict the substitute words. However, this approach can be generalised to other pre-trained models as well. Lastly, the output of the algorithm is the training datasets, including syntactic records.

Fig. 6 presents an example of synthetic issue reports generated with the contextual data augmenter via comparison with the

6 https://huggingface.co/bert-base-uncased/.

Algorithm 1: An algorithm with which to generate a synthetic issue report with the contextual data augmentation approach

```
input  : list of training issue reports TB, augmentation
         threshold Threshold
output: list of synthetic issue reports TS
MajC ← a majority class samples count;
MinC ← total no of minority classes;
MinClist ← list of minority classes;
EstimateDataAugAmount ← MinC ∗ MajC;
if EstimateDataAugAmount < Threshold then
  | MajC ←(Threshold / EstimateDataAugAmount) ∗ MajC
end
for minclass ∈ MinClist do
    BC ←retrieve total no of issue reports fixed by minclass
    from TB;
    while BC < MajC do
        RC ←retrieve one random record of minclass from
        TB;
        NC ←generate a new synthetic record based on RC
        with contextual data augmentation approach;
        Append NC to TS;
        BC ←BC + 1;
    end
end
return TS
```

original issue report. As shown in Fig. 6, all the syntactic context generated by the data augmenter is underlined. In general, the data augmenter generates the synthetic reports by substituting the main keywords from original issue reports while maintaining the original context. In the next section, the final component of the framework, the multi-triage model, is explained in detail.

3.4. Multi-triage model

As shown in Fig. 7, the multi-triage model has three main components: the context encoder, the AST encoder, and classifiers. The two encoders are used to generate the natural language and structural (code) representation based on the input issue reports. The share layer between the encoders concatenates the outputs of the encoders to construct the overall feature representations of issue reports. Finally, the classifiers analyse these feature representations and recommend the potential developers and issue types as outputs. The main hyper-parameters of the model are $batch = 32$, $max\_seq\_length = 300$, $embedding\_dim = 100$, and $num\_filters = 100$. The batch size can set between 1 and a few hundred; however, a standard batch size (32) was selected to train this model (Bengio, 2012).

3.5. Code representation

Context encoder. As mentioned in Section 1, extracting the representation features of issue reports is non-trivial in the bug-triage process. In this model, a context encoder is used to extract the natural language representations of the issue report. Convolutional neural networks (CNN) are used to generate these representations. In recent years, CNN have been successfully applied in various modelling tasks, including textural classification (Lee et al., 2017; Banerjee et al., 2019; Kim, 2014) and image classification (Mustafa et al., 2019; Lee and Kwon, 2017). The input of this encoder is the concatenated values of issue title and description. The raw input is normalised by removing stop words, stemming, lower-casing, and padding equally to the right with *max_seq_length* range. First, each issue report is transformed into a vector by turning each issue report into a sequence of integers (each integer value being the index of a token in a dictionary). Second, these inputs are fed into a word embedding layer with input dimension ($vocab\_size$ + 1). A dynamic $vocab\_size$ value equal to the size of the vocabulary of each project is used. The next layer is filters, which are the core of CNN's architecture. 1D convolution is applied via *filters*. The standard kernel size of 4×4 is used to extract the important features (Collobert et al., 2011). Then, the max-over-time pooling operation is applied to extract the most relevant information from each feature map. Finally, the pooling output is passed into the joining layer for concatenation.

For example, given an issue report with n words $[b_1, b_2, \ldots, b_n]$, the word vectors corresponding to each word are presented as $[x_1, x_2, \ldots, x_n]$ (i.e. x_i is the word vector representation of word b_i). Let $x_i \epsilon \mathbb{R}$ be k-dimensional (k=1). The inputs of a convolution layer are the concatenation of each word vectors, represented as:

$$x_{1:n} = x_1 \oplus x_2 \oplus \cdots \oplus x_n, \tag{1}$$

where $\oplus$ denotes the concatenation operator. In a convolution layer, a filter $w \; \epsilon \; \mathbb{R}$, slides across inputs by applying a window of h=4 (words) to capture the relevant features. In general, a feature c_i is processed by sliding a window of words $x_{i:i+h-1}$ by

$$c_i = f(w \cdot x_{i:i+h-1} + b), \tag{2}$$

where b denotes bias and f is a non-linear function (i.e. the hyperbolic tangent function)

$$c = [c_1, c_2, \ldots, c_{n-h+1}]. \tag{3}$$

Finally, a max-over-time pooling operation (Alaeddine and Jihene, 2021) is applied to extract the maximum value $\hat{c} = max\{c\}$ to capture the most important feature for each feature map. In general, one feature is extracted from one filter. In this model, 100 filters are used to obtain multiple features from the issue report. Next, the output is flattened to one dimension and fed into the joining layer.

AST encoder. In this approach, each code snippet in an issue report is parsed to construct an AST path using the AST extractor and which is used as input to the AST encoder. In the preprocessing phase, all inputs are first prepared to the same size by padding equally to the right with *max_seq_length* range. Second, an AST path is transformed into a vector by turning each word into a sequence of integers. Next, these inputs are fed into the word-embedding layer with input dimension ($vocab\_size$ + 1).

To learn AST representations, bidirectional recurrent neural networks with long short-term memory (BiLSTM) neurons (Graves and Schmidhuber, 2005; Cai et al., 2019) are used. In general, BiLSTM models combine two separate LSTM layers which operate in opposite directions (i.e. forward and backward) to utilise information from both preceding and succeeding states. In LSTM networks, each memory cell c contains three gates: input gate i, forget gate f, and output gate o. Formally, an input AST sequence vector $[a_1, a_2, \ldots, a_n]$ is given, where n denotes the length of the sequence. The input gate i controls how much of the input a_t is saved to the current cell state c_t. Next, the forget gate f controls how much of the previous cell state c_{t-1} is retrained in the current cell state c_t. Lastly, the output gate controls how much of the current cell state c_t is submitted to the current output h_t. The formal representation of the LSTM network is as follows:

$$\begin{aligned} i_t &= \sigma(W_{ia}a_t + W_{ih}h_{t-1} + b_i), \\ f_t &= \sigma(W_{fa}a_t + W_{fh}h_{t-1} + b_f), \\ o_t &= \sigma(W_{oa}a_t + W_{oh}h_{t-1} + b_o), \\ c_t &= f_t * c_{t-1} + i_t * \tanh .(W_{ca}a_t + W_{ch}h_{t-1} + b_c), \\ h_t &= o_t * \tanh(c_t). \end{aligned} \tag{4}$$

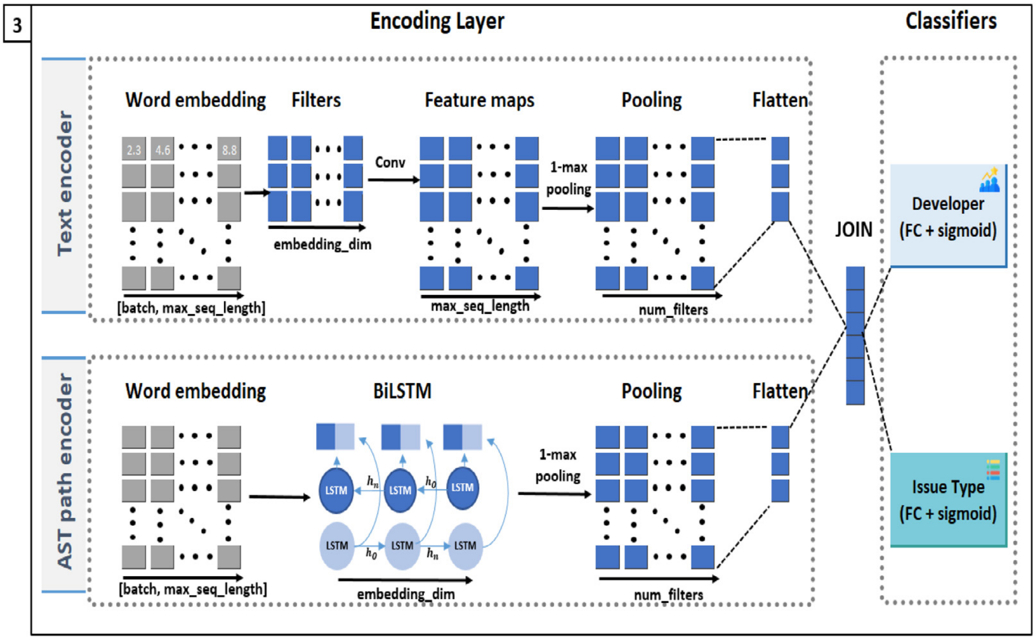


Fig. 7. The multi-triage model.

In Eq. (4), a_t indicates the input word vector of the AST path, h_t indicates the hidden state, W indicates the weight matrix, b indicates the bias vector, and σ indicates the logistic sigmoid function. A BiLSTM network calculates the input AST sequence vector a in a forward direction sequence $\vec{h}_t = [\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_n]$ and a backward direction sequence $\overleftarrow{h}_t = [\overleftarrow{h}_1, \overleftarrow{h}_2, \ldots, \overleftarrow{h}_n]$, then concatenates the outputs $y_t = [\vec{h}_t, \overleftarrow{h}_t]$. The formal representation of the BiLSTM network is as follows:

$$\begin{aligned} \vec{h}_t &= \sigma(W_{\vec{h}a} a_t + W_{\vec{h}} \vec{h}_{t-1} + b_{\vec{h}}), \\ \overleftarrow{h}_t &= \sigma(W_{\overleftarrow{h}a} a_t + W_{\overleftarrow{h}} \overleftarrow{h}_{t+1} + b_{\overleftarrow{h}}), \\ y_t &= W_{y\vec{h}} \vec{h}_t + W_{y\overleftarrow{h}} \overleftarrow{h}_t + b_y \end{aligned} \tag{5}$$

In Eq. (5), y_t is the output sequence of the hidden layer h_t at a time step t. Next, a max-over-time pooling operation (Alaeddine and Jihene, 2021) is applied over BiLSTM outputs to extract the important information. Finally, the output is flattened and fed into the joining layer. In the joining layer, the two encoders are concatenated, output, and fed into the classification layer.

3.6. Task-specific classifiers

The sigmoid function was used to classify the relevant developers and issue types for a new issue report. As illustrated in Fig. 7, both developer and issue type classifiers share the same structure but differ in their input labels (i.e. developer and issue type). Therefore, only illustrate one classification layer is illustrated in this section. The classification layer is composed of two layers: a fully-connected FFN with ReLUs as well as a sigmoid layer.

Label classifier. In the FFN layer, the ReLU is an activation function that outputs the input directly if the input is positive; otherwise, it will output zero (Nair and Hinton, 2010). In Eq. (6), x denotes the concatenated embedding vector with 150 dimensions, W denotes weights, and b denotes bias. Next, the output vectors are fed into the sigmoid layer to predict the appropriate developers or issue types for the input issue report.

$$\mathrm{FFN}(x) = \max(0, xW_i + b_i). \tag{6}$$

The sigmoid exponential activation function is then used to calculate the probability distribution of the output vectors from the FFN layer for each possible class (i.e. developers or issue types):

$$\mathrm{P}(c_j|x_i) = \frac{1}{1 + \exp(-z_j)}. \tag{7}$$

Eq. (7) presents the formal representation of the sigmoid activation function at the final neural network layer to calculate the probability of a class c_j, where x_i is an input issue report and z_j is the output of the FFN layer.

4. Data and evaluation

In this section, the research questions and detailed information on the experimental implementation are presented. The code, data and trained models are available at (Aung, 2021).

Datasets. The issue reports of ten GitHub projects were collected as described in Table 2. In addition, eclipse issue reports were also collected effectively compare the present approach against baseline studies. Following previous studies, only retrieve the issue reports with 'closed' status (Anvik et al., 2006; Mani et al., 2019; Lee et al., 2017; Xi et al., 2018) are retrieved. The issue reports with unassigned developers or issue types are also removed, as the model cannot be trained and validated with unlabelled records. Furthermore, issue reports assigned to 'software bots', which are frequently used in automatic issue assignment processes (Golzadeh et al., 2020), are excluded. As no actual developer is used, these reports are not applicable to use in the developer prediction process. The statistics of datasets such as labels (i.e. developers, issue types) and code snippets are presented Table 2. In terms of issue reports metadata, an issue report title, description, creation date, assignee, and labels are presented, as well as the corresponding pull request's assignee information, to create a tossing sequence.

Single task learning model. The two single-task learning models shown, below are constructed to evaluate the effectiveness of this multi-task learning model.

- **BiLSTM-based triage model -** Two single-task BiLSTM networks are constructed: one for the developers' prediction task and the other one for the issue types prediction task. In these models, architecture similar to the multi-triage model is replicated and used to create the two-word embedding layers to contract textual information and AST paths embedding tokens. Next, these two embedding tokens are concatenated and fed into the BiLSTM network to learn the issue report's representation. Finally, these learned vectors are passed into the classifier to predict labels (i.e. developers or issue types).
- **CNN-based triage model -** Similar to the BiLSTM model, the two single-task networks are constructed using CNN networks to learn the representations of issue reports.

As noted in Section 3, the multi-triage model combines BiLSTM and CNN networks to learn the representations of issue reports. Therefore, single networks are built using these two networks to effectively compare the time and accuracy trade-offs of the model.

Baselines. The below two baselines approaches were used to evaluate the effective of the present approach.

- SVM+BOW (Anvik et al., 2006): This uses a Tf–IDF weighting matrix to transform textual features of issue reports into vector representations, and applies a support vector machine (SVM) machine learning classifier to automate the bug triage process.
- DeepTriage (Mani et al., 2019) - This uses a recurrent neural network (RNN) to learn the representations of issue reports and a softmax layer to recommend the potential developers and issue types as outputs.

Both of these approaches focus on predicting labels for a new issue report by learning the representation of existing issue reports. The first approach uses a support vector machine, whereas the second utilises a recurrent neural network to automate the bug triage process. As the present approach uses BiLSTM and CNN to learn the representations of issue reports, these approaches have been selected for evaluation. For SVM+BOW, scikit-learn libraries are used to set up SVM+BOW because the source code is not accessible. In addition, the scikit-learn is widely used in various studies [50, 2] to set up machine learning algorithms, including SVM.

Ablation analysis. Parameter analysis plays a crucial role in the supervised learning model since tuning a single parameter can affect the model performance. Ablation analysis is a procedure investigating configuration paths to ascertain which model's parameters contribute most in optimising model performance (Fawcett and Hoos, 2016; Biedenkapp et al., 2017). An ablation analysis procedure is adopted to determine which components of the multi-triage model contribute most in leveraging model performance. In the ablation analysis approach, developers identify a set of candidate parameters, evaluate the training data by running with these parameters, and take the candidate parameter which outperforms at least one other configuration. In this study's ablation analysis experiments, encoder decoupling and parameters tuning are performed to determine which encoder and parameters contribute most to improving model performance.

Evaluation settings. The time-series-based 5-fold cross-validation procedure is followed to split the training (train), development (dev), and test sets (Bhattacharya and Neamtiu, 2010; Tamrawi et al., 2011; Jiang and Wang, 2017; Xia et al., 2016b; Bergmeir and Benítez, 2012). This is a commonly used validation approach to measure the generalisability of a learning model. Fig. 8 presents the validation approach used in the data evaluation process. In this approach, the dev set makes up 10 per cent of the train set, and the test set assigns 20 per cent of the subset of the allocated data sample. The dataset is folded on a rolling basis, based on the issue report creation date in ascending order.

All experiments are run in the google-colab[7] cloud-based platform on tesla v100-sxm2 GPU with 32 GB RAM. Python source code provided by the authors is used to set up the baseline models (i.e. SVM+BOW (Anvik et al., 2006) and DeepTriage (Mani et al., 2019)). Also, the deep learning model is implemented using the TensorFlow Keras[8] deep learning library. In the multi-triage approach, both text input and AST path input are truncated to the length of 300. Each word is embedded into 100 dimensions. The output sizes of the text encoder and the AST encoder are 100 and 50, respectively. After joining the two encoder outputs, batch normalisation is performed on the concatenated output and the drop (rate 0.5) is employed to reduce overfitting (Achille and Soatto, 2018). For the classifier, binary-crossentropy and the Adam optimiser from the Keras library are used with a learning rate of 0.001. The model is tuned with different dimension sizes and learning rates, and results are presented in Section 5. Finally, the vocabulary size is set based on individual project vocab size and the default batch size (32) is used to train the model.

Evaluation metrics. In these experiments, F-scores are used to measure the model's accuracy. In the following equations, *TP* denotes true positives, *TN* denotes true negatives, *FP* denotes false positives, and *FN* denotes false negatives.

- Precision — This is the ratio of the predicted correct labels to the total number of actual labels averaged over all instances. Eq. (8) presents the precision formula:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{8}$$

- Recall — This is the ratio of the predicted correct labels to the total number of predicated labels averaged over all instances. Eq. (9) presents the recall formula:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{9}$$

- F-scores — This is a commonly used metric for the bug triage process. It is calculated from the precision and recall scores. The F1 score is calculated by assigning equal weights to precision and recall, while the F2 score adds more weight to recall. Even though both precision and recall are important, the F2 score is usually preferred in bug triage studies, where measuring the recall is more non-trivial than precision. Eq. (10) presents the F2 score formula:

$$\text{F}_{\beta} = \frac{(1+\beta^2) \times precision \times recall}{\beta^2 \times precision \times recall} \tag{10}$$

- Accuracy —Calculated by the average across all instances, where the accuracy of each instance is the ratio of the predicated correct labels to the total number of (predicated and actual) labels for that instance. Eq. (11) presents the accuracy formula:

$$\text{Accuracy} = \frac{TN + TP}{TN + TP + FN + FP} \tag{11}$$

7 https://github.com/dotnet/aspnetcore/.
8 https://www.tensorflow.org/.

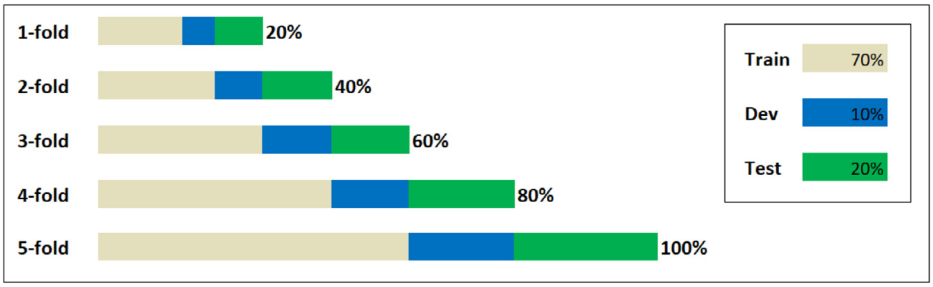


Fig. 8. Time-series-based 5-fold cross-validation.

5. Results

In this section, evaluation results are presented for the three research questions.

5.1. RQ1: How does the multi-triage model compare to other approaches?

The performance of the multi-triage model is compared to that of (SVM + BOW) (Anvik et al., 2006) and DeepTriage (Mani et al., 2019) in the eleven open-source projects. The comparison results are presented in Table 3. The time-series-based 5-fold validation is performed on all approaches, and the average accuracy is presented for both developers and issue types prediction results. Since the Deeptriage (Mani et al., 2019) source code is publicly available, its environment can be replicated. However, the source code of (SVM + BOW) (Anvik et al., 2006) is not accessible, and thus it was manually implemented using sklearn[9] libraries. Both approaches filter out code snippets and stack trace as these features are excluded in these models. Conversely, this approach generates a separate token for each code snippet by parsing it to AST paths and including it in the model's training.

As shown in Table 3, this approach outperforms (SVM + BOW) (Anvik et al., 2006) and DeepTriage (Mani et al., 2019) by an average increase of 10 and 7 percentage points for developers, and 15 and 11 percentage points for issue types, respectively. At its highest, this approach achieves 69% and 57% for developers and issue types, respectively. It was observed that, in both prediction tasks, an accuracy lower than 40% on the projects (i.e. eclipse, elasticsearch, nunit, and Roslyn) has either the higher number of potential issue types or developers' labels, or low sample data compared to the rest of the projects. In summary, this approach achieves the best performance, with DeepTriage (Mani et al., 2019) second by comparison. In the following section, the qualitative analysis test is performed to determine how many bug and enhancement records were correctly predicted with this approach compared to the state-of-the-art approaches.

In the qualitative analysis evaluation, sample data is subdivided into two issue types, namely (1) bug and (2) enhancements group, and the performance is analysed on the prediction results. Table 4 presents the statistics of the prediction results in terms of numbers, whereas the Venn diagram in Fig. 9 illustrates the total numbers of bugs and enhancements found by base1, base2, and the present approach. Notably, the present approach can predict all issue types which are predicated correctly in base1 and base2. In addition, this approach predicts 546 bugs and 46 enhancement records missed by baseline approaches. After inspecting these records, it became clear that these reports provide trivial descriptive text with code snippets to reproduce the issue. Previous studies neglected the code snippets in their approach, as the feature representation of these records cannot provide valuable features for the model to perform the prediction. It was also observed that most of the descriptive information provided in the bug and enhancement reports used similar terms. For example, terms such as 'add', 'improve', 'enhance', 'upgrade' and 'include' are frequently used in both bug and enhancement reports. Thus, the baseline approaches that relied on the issue reports' textual features might wrongly mislabel as enhancement in some scenarios. The present approach uses textual and AST representation of the issue reports to eliminate the mislabelling case by using the additional context from code snippet metadata.

In addition, it was further observed that the reports failed to predict from all three approaches. Interestingly, these records do not include either non-trivial descriptive text or code snippets. These issue reports include either screenshot images, stack trace information, hyperlinks, which are ignored in all three approaches. Stack trace information was neglected by this approach in order to reduce noises in the model training. Screenshots were not covered due to limitations of the model, which supports either natural language or structural context.

5.2. RQ2: Which component contributes more to the multi-triage model?

Ablation analysis is performed on the multi-triage model to ascertain which component contributes more to model performance. To answer this question, the ablation analysis is divided into two sections: (1) system component level ablation analysis, and (2) embedding parameter level ablation analysis.

5.2.1. System component level ablation analysis

This section compares the multi-task learning model with the conventional single task learning model to analyse which model performs better. The two single-task learning models, one with CNN and the other with BiLSTM networks, are implemented by referring to the present approach's encoder architecture. In a single model, the text and the AST path's are concatenated into one token and fed into the CNN, or BiLSTMs layer, respectively. The same classifier components are used in a single model. The outputs of the single task learning model are either developers or issue labels. The comparison results are presented in Table 5 for developers and Table 6 for issue types predictions. Table 5(a) and Table 6(a) present the precision and recall, whereas Table 5(b) and Table 6(b) describe the accuracy and F2 scores. Out of the three models, the present model achieves the best performance in precision, recall, accuracy and F2 score.

In terms of developer precision and recall, the present model outperforms the others by an average increase of 8 percentage points compared to single CNN, and 7 percentage points compared to single BiLSTM. In recall, it improves on both single CNN and single BiLTSM by an average increase of 8 percentage points. In accuracy, on average, it exceeds the others by 10 percentage points compared to single CNN, and by 8 percentage points

[9] https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.

Table 3
Multi-triage v.s. baselines (Base1 - SVM + BOW Anvik et al. (2006), Base2 - DeepTriage Mani et al. (2019)) Accuracy (%).

Project	Developer			Issue type		
	Base1	Base 2	Multi-triage	Base1	Base 2	Multi-triage
aspnetcore	58%	51%	63%	25%	27%	47%
azure-powershell	35%	39%	48%	24%	29%	44%
ecplise	31%	35%	54%	23%	24%	26%
efcore	52%	55%	59%	30%	34%	40%
elasticsearch	46%	53%	58%	13%	21%	31%
mixedrealitytoolkitunity	41%	50%	62%	30%	33%	47%
monogame	62%	65%	69%	53%	55%	57%
nunit	36%	38%	41%	19%	23%	27%
realmjava	59%	60%	62%	24%	25%	50%
roslyn	33%	35%	39%	22%	25%	27%
rxjava	64%	66%	68%	31%	40%	49%
AVG	47%	50%	57%	27%	31%	42%
MAX	64%	66%	69%	53%	55%	57%

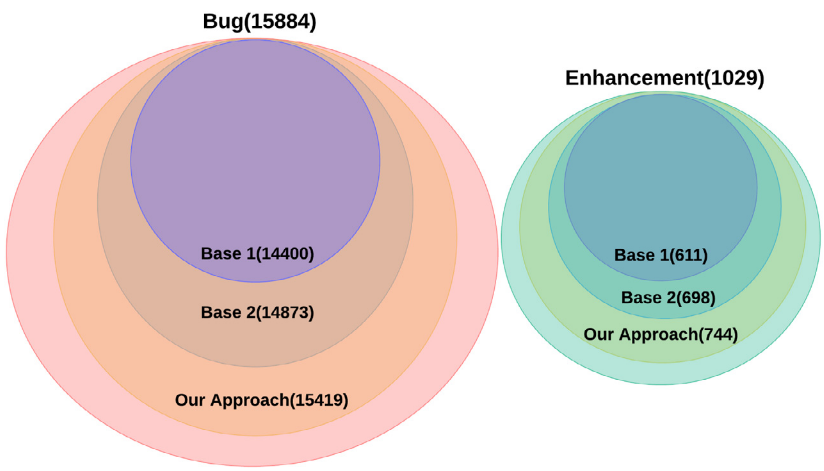


Fig. 9. Qualitative analysis venn diagram.

Table 4
Qualitative analysis (bug and enhancement).

Project	Bug			Enhancement			Total Bug/ Enhancement
	Base1	Base2	Our approach	Base 1	Base 2	Our approach	
aspnetcore	1124	1148	1382	21	25	29	1391/39
azurepowershell	434	448	455	3	4	5	499/8
ecplise	9670	9950	9980	78	91	96	10035/126
efcore	1038	1069	1216	51	65	68	1245/77
elasticsearch	811	820	857	45	66	69	939/99
mixedrealitytoolkitunity	386	413	435	9	13	14	435/23
monogame	122	144	151	10	14	16	180/21
nunit	65	79	97	8	13	16	109/22
realmjava	148	171	190	4	6	7	220/12
roslyn	291	299	303	360	370	390	458/560
rxjava	311	332	353	22	31	34	373/42
AVG	1309	1352	1406	56	63	68	
MAX	9670	9950	9980	360	370	390	
Total	14402	14873	15419	611	698	744	15884/744

compared to single BiLSTM. In F2 scores, this model performs better than the single CNN by 9 percentage points, and the single BiLSTM by 8 percentage points. Therefore, it can be concluded that developers and issue types prediction tasks are compatible with learning in one large network.

Interestingly, similar improvements were found for issue types prediction results. In issue types precision, the present model outperforms the others on average by 6 percentage points compared to single CNN, and by 5 percentage points compared to single BiLSTM. In recall, it improves on single CNN by 6 percentage points and on single BiLTSM by 2 percentage points, on average. In accuracy, on average, it exceeds the others by 8 percentage points compared to single CNN and by 5 percentage points compared to single BiLSTM. In F2 scores, the model performs slightly better than single CNN by 1 percentage point, and the same for single BiLSTM. Therefore, it is possible to conclude that developers and issue types prediction tasks are compatible with learning in one large network.

Training times for each model are also presented in Fig. 10. On average, the multi-triage model accelerates the training process with the drop of 476 s and 1175 s compared to the single CNN and single BiLSTM models, respectively. Although the accelerated

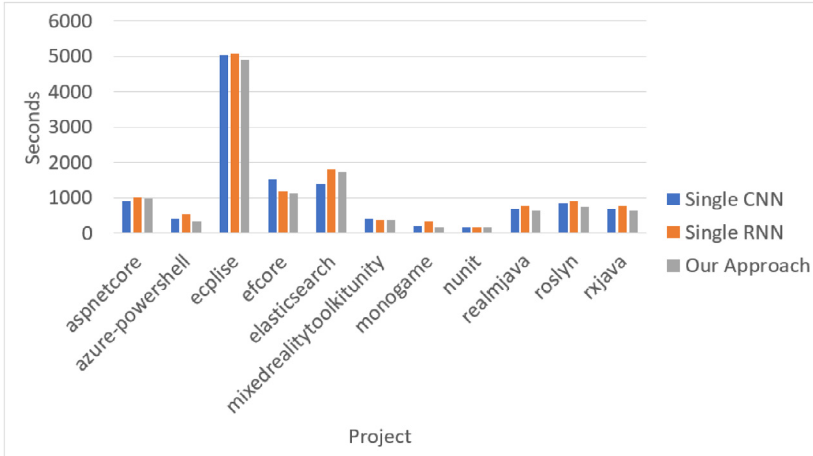


Fig. 10. Training time.

Table 5
Single task prediction model v.s. our approach for developer predictions (precision(P), recall(R), and accuracy(Acc)).

(a) Developers precision and recall results

Project	Single CNN		Single BiLSTM		Multi-triage	
	P	R	P	R	P	R
aspnetcore	57%	52%	57%	51%	66%	61%
azure-powershell	52%	40%	52%	40%	55%	42%
ecplise	43%	24%	50%	30%	52%	38%
efcore	53%	47%	53%	47%	62%	56%
elasticsearch	54%	44%	54%	44%	62%	53%
mixedrealitytoolkitunity	56%	51%	55%	51%	64%	60%
monogame	59%	59%	59%	59%	69%	69%
nunit	49%	42%	51%	45%	59%	52%
realmjava	55%	52%	55%	52%	64%	61%
roslyn	49%	33%	49%	33%	52%	35%
rxjava	58%	58%	58%	58%	68%	68%
AVG	53%	46%	54%	46%	61%	54%
MAX	59%	59%	59%	59%	69%	69%

(b) Developers accuracy and F2 results

Project	Single CNN		Single BiLSTM		Multi-triage	
	Acc	F2	Acc	F2	Acc	F2
aspnetcore	54%	52%	53%	51%	63%	61%
azure-powershell	44%	41%	44%	41%	48%	42%
ecplise	32%	24%	46%	33%	54%	38%
efcore	50%	48%	50%	47%	59%	56%
elasticsearch	47%	44%	48%	45%	58%	53%
mixedrealitytoolkitunity	53%	51%	53%	52%	62%	60%
monogame	59%	59%	59%	59%	69%	69%
nunit	34%	43%	37%	46%	41%	52%
realmjava	53%	52%	53%	53%	62%	61%
roslyn	35%	37%	34%	37%	39%	38%
rxjava	58%	58%	58%	58%	68%	68%
AVG	47%	46%	49%	47%	57%	55%
MAX	59%	59%	59%	59%	69%	69%

Table 6
Single task prediction model v.s. our approach for issue type predictions (precision(P), recall(R), and accuracy(Acc)).

(a) Issue types precision and recall results

Project	Single CNN		Single BiLSTM		Multi-triage	
	P	R	P	R	P	R
aspnetcore	52%	38%	52%	37%	58%	39%
azure-powershell	49%	38%	51%	42%	59%	47%
ecplise	28%	22%	48%	34%	52%	33%
efcore	48%	36%	48%	34%	52%	33%
elasticsearch	44%	21%	43%	20%	48%	25%
mixedrealitytoolkitunity	49%	34%	50%	40%	55%	41%
monogame	53%	46%	55%	49%	60%	53%
nunit	48%	38%	51%	44%	58%	49%
realmjava	46%	35%	50%	43%	56%	45%
roslyn	46%	30%	47%	33%	54%	38%
rxjava	49%	41%	50%	43%	53%	44%
AVG	47%	34%	48%	38%	53%	40%
MAX	53%	46%	55%	49%	60%	53%

(b) Issue types accuracy and F2 results

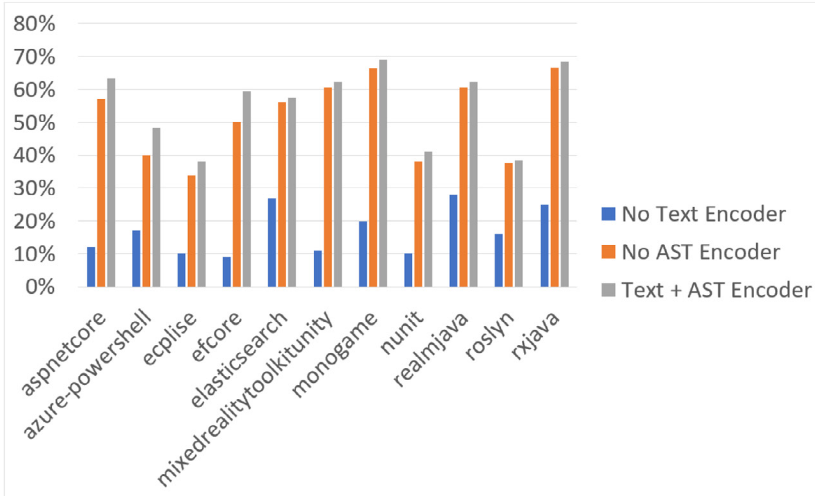
Project	Single CNN		Single BiLSTM		Multi-triage	
	Acc	F2	Acc	F2	Acc	F2
aspnetcore	43%	43%	43%	43%	47%	43%
azure-powershell	39%	40%	39%	40%	44%	49%
ecplise	21%	20%	38%	37%	26%	39%
efcore	38%	38%	38%	37%	40%	38%
elasticsearch	29%	27%	29%	26%	31%	27%
mixedrealitytoolkitunity	40%	38%	45%	43%	47%	38%
monogame	49%	49%	51%	51%	57%	49%
nunit	28%	38%	24%	43%	27%	38%
realmjava	40%	37%	46%	45%	50%	37%
roslyn	22%	32%	25%	34%	27%	32%
rxjava	43%	43%	45%	45%	49%	43%
AVG	36%	37%	37%	39%	42%	39%
MAX	49%	49%	51%	51%	57%	49%

training times are not obvious in the present scenario, imagine a project with N issue reports; the training time complexity of the single model is ($N^2 * t$), where t is the time consumed by the model to learn feature representations of each issue report. However, the multi-triage model only needs ($N * t$) times to learn the feature representation; therefore, the present model is more capable of scaling to train to projects with large amounts of training data. In summary, the multi-triage model outperforms the single task learning model in terms of accuracy and training time.

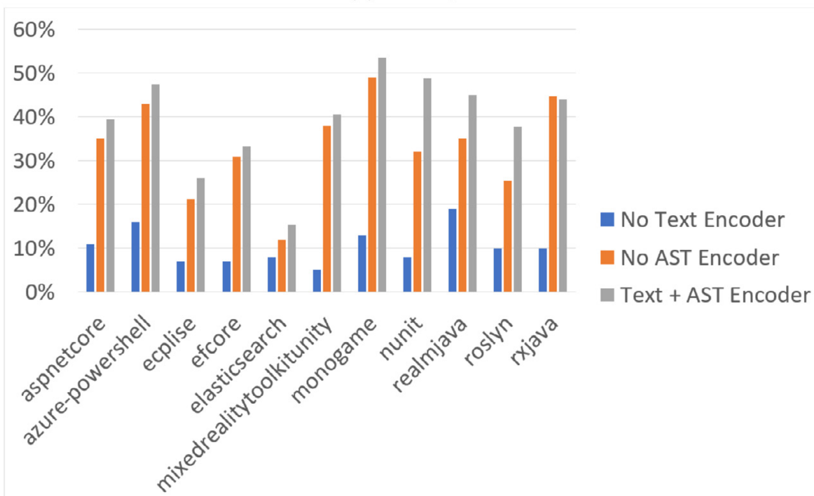
5.2.2. Embedding parameter level ablation analysis

Two types of ablation analysis are performed to evaluate the embedding parameters: encoder decoupling and parameters tuning.

In the *encoder decoupling* experiment, the two encoders, text and AST, are decoupled, and the model's performance is evaluated with three experimental settings: (1) no text encoder, (2) no AST encoder, and (3) both. In the *no text encoder* experiment, the negation effect of the textual input is studied. Similarly, AST paths input is excluded in the *no AST encoder* experiment. The comparison results for prediction accuracy of developers and issue types in Figs. 11(a) and 11(b), respectively. In both predictions, the



(a) Developer



(b) Issue Type

Fig. 11. Multi-triage: ablation analysis.

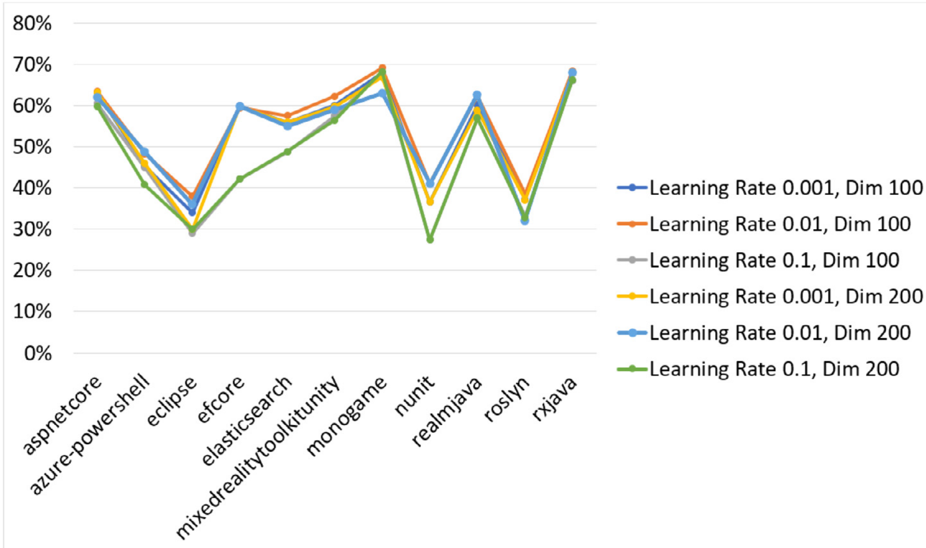
Table 7
Unique word count for Text and AST.

Project	Text	AST
aspnetcore	32959	29559
azure-powershell	20005	5200
ecplise	342103	1234
efcore	24627	51115
elasticsearch	28116	223942
mixedrealitytoolkitunity	11749	3570
monogame	8839	6351
nunit	5231	3197
realmjava	10950	40145
roslyn	21265	25372
rxjava	11225	93517
AVG	47006	43927
MAX	342103	223942

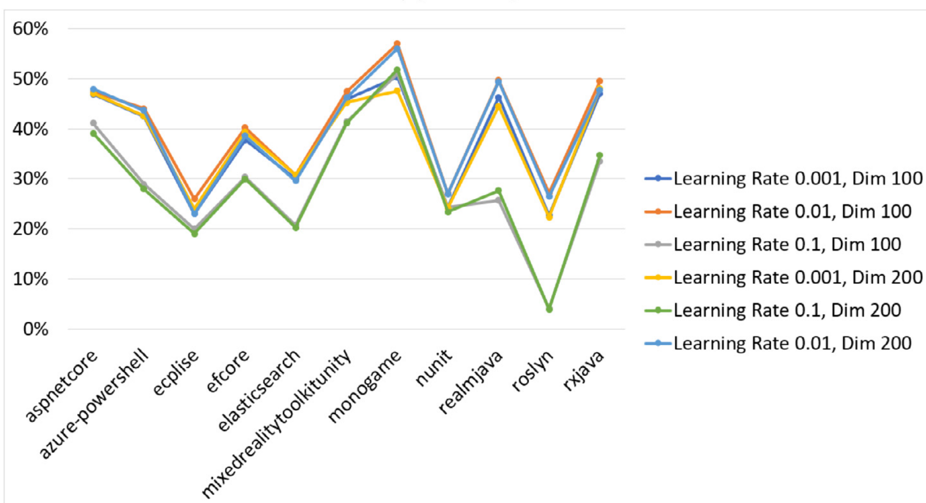
combination of textual and AST path inputs achieves the highest results in all eleven projects, with an average increase of 35 and 3 percentage points for developers and 23 and 6 percentage points for issue types in comparison with no text encoder and no AST encoder, respectively. Therefore, it can be concluded that both the textual encoder and AST encoder are important components of the multi-triage model.

In the *parameters tuning* experiment, the effects of embedding dimension and learning rate on the accuracy of our model were analysed. The model was tuned with embedding dimensions (100 and 200) and learning rates (0.1, 0.01, and 0.001), which are the most commonly used hyper-parameters in deep learning models. As previously mentioned, a time-series-based cross-validation approach was adopted, and the model was trained with various learning rates and embedding dimension size incrementally. Fig. 12 presents the accuracy results for the six experiments with developer prediction accuracy in Fig. 12(a) and issue types prediction accuracy in Fig. 12(b).In both prediction tasks, embedding dimension 100 with a learning rate of 0.01 provides the highest average, with an accuracy of 55 percentage points for developers and 41 percentage points for issue types. The embedding dimension 200 with a learning rate of 0.01 follows at second, with an average accuracy of 53 percentage points for developer and 40 percentage points for issue types.

The internal validity of the embedding parameter results are further analysed by validating the total number of unique word counts for both text encoder and AST encoder input for each project. Table 7 presents the word count results for all projects. Stop words and special characters were filtered out before the number of unique words was counted. As shown in Table 7, the average word counts for text encoder input is 47006, whereas the



(a) Developer



(b) Issue type

Fig. 12. Multi-triage parameter analysis.

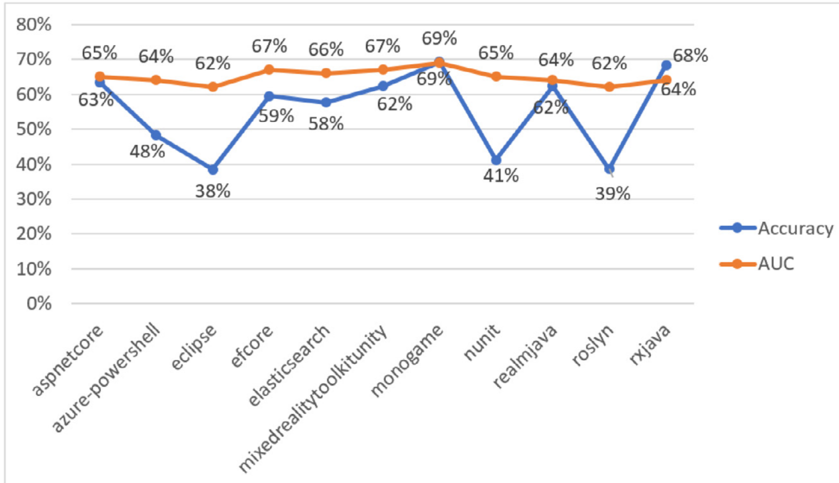
AST encoder input is 43927. The highest word count is 342103 for text encoder and 223942 for AST encoder, respectively. By following previous studies, a word corpus of around 2 million is trained with embedding size 300 or higher (Pennington et al., 2014; Radford et al., 2019; Kenton and Toutanova, 2019). The maximum corpus size of the projects' is lower than 35k, as it is reasonable that both 100 and 200 embeddings provide comparable results in these experiments. However, 100 embedding size was selected as the optimal hyper-parameter to eliminate complex processing. In summary, a learning rate of 0.01 with embedding dimension 100 hyper-parameters were used as optimal parameters to train the model.

Table 8
No data augmentation v.s. data augmentation (accuracy(%)).

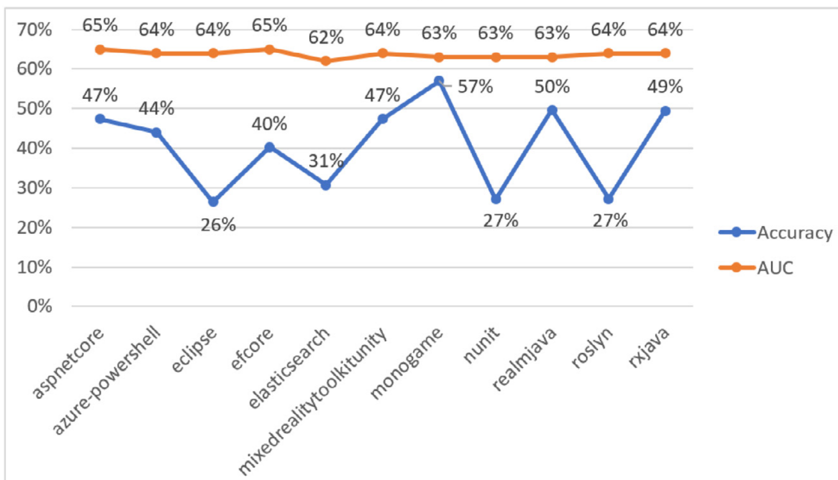
Project	Multi-triage		Multi-triage A+	
	Dev	Issue Type	Dev	Issue Type
aspnetcore	63%	47%	64%	48%
azure-powershell	48%	44%	50%	48%
ecplise	38%	26%	40%	30%
efcore	59%	40%	61%	42%
elasticsearch	58%	31%	60%	33%
mixedrealitytoolkitunity	62%	47%	63%	49%
monogame	69%	57%	70%	58%
nunit	41%	27%	46%	29%
realmjava	62%	50%	63%	51%
roslyn	39%	27%	40%	28%
rxjava	68%	49%	69%	53%
AVG	55%	41%	57%	43%
MAX	69%	57%	70%	58%

5.3. RQ3: Does increasing the size of training datasets (based on the contextual data augmentation approach) improve our model's accuracy?

In this section, the data-imbalanced problem is addressed with the contextual data augmentation approach presented in algorithm 1. First, an Area under the ROC Curve (AUC) analysis is performed to measure classifier performance. Fig. 13 presents the average AUC and accuracy results for the multi-triage model. The line graph in 13(a) illustrates the developers' AUC and accuracy results, whereas the line graph in 13(b) shows the issue types



(a)



(b)

Fig. 13. Multi-triage: AUC v.s. Accuracy.

AUC and accuracy results. In both tasks, AUC fluctuates around 62% and 69%, which indicates that the classifiers perform fairly well.

Therefore, further analysis was performed on the impact of the size of the training data on model accuracy. The training data size was increased by using algorithm 1. Table 8 presents the comparison results. For ease of reference, the model that uses augmented data was named as multi-triage (A). As mentioned earlier, the training data augmentation size was incrementally increased in each cross-fold validation as the average accuracy from the 5-fold validation result was reported. As shown in Table 8, the model accuracy slightly improved in the multi-triage (A) model, with an average increase of 2 percentage points on both developers and issue types. The performance of the prediction model was further analysed using the AUC test. Fig. 14 presents the AUC represented for the multi-triage (A) model. The line graphs in 14(a) and 14(b) illustrate the developers and issue types of AUC and accuracy results. Notably, AUC performance increased an average of 4 percentage points for developers and 3 percentage points for issue types in comparison to the multi-triage model AUC performance, as shown in Fig. 13. The data augmentation approach leveraged the base multi-triage model in both accuracy and AUC performance measure. Therefore, it is concluded that the contextual data augmentation approach effectively increases the issue reports training data.

6. Threats to validity

Threats to external validity. This relates to the quality of the datasets we used to evaluate our model. We used issue reports from eleven open-source projects written in C# and Java languages to generalise our works. All the datasets' programs were collected from GitHub repositories; each dataset contains over 600 training issue reports. However, further studies are needed to validate and generalise our findings to other structural languages. Furthermore, more case studies are needed to confirm and improve the usefulness of our multi-triage recommendation model.

Threats to internal validity. This includes the influence of hyper-parameters settings. Our model's performance would be affected by different learning rates and embedding dimensions, which were set manually in our experiments. Another threat to internal validity relates to the errors in the implementation of the benchmark methods. For DeepTriage (Mani et al., 2019), we directly used their published GitHub repository. For SVM+BOW (Anvik et al., 2006), we implemented it ourselves using scikit-learn

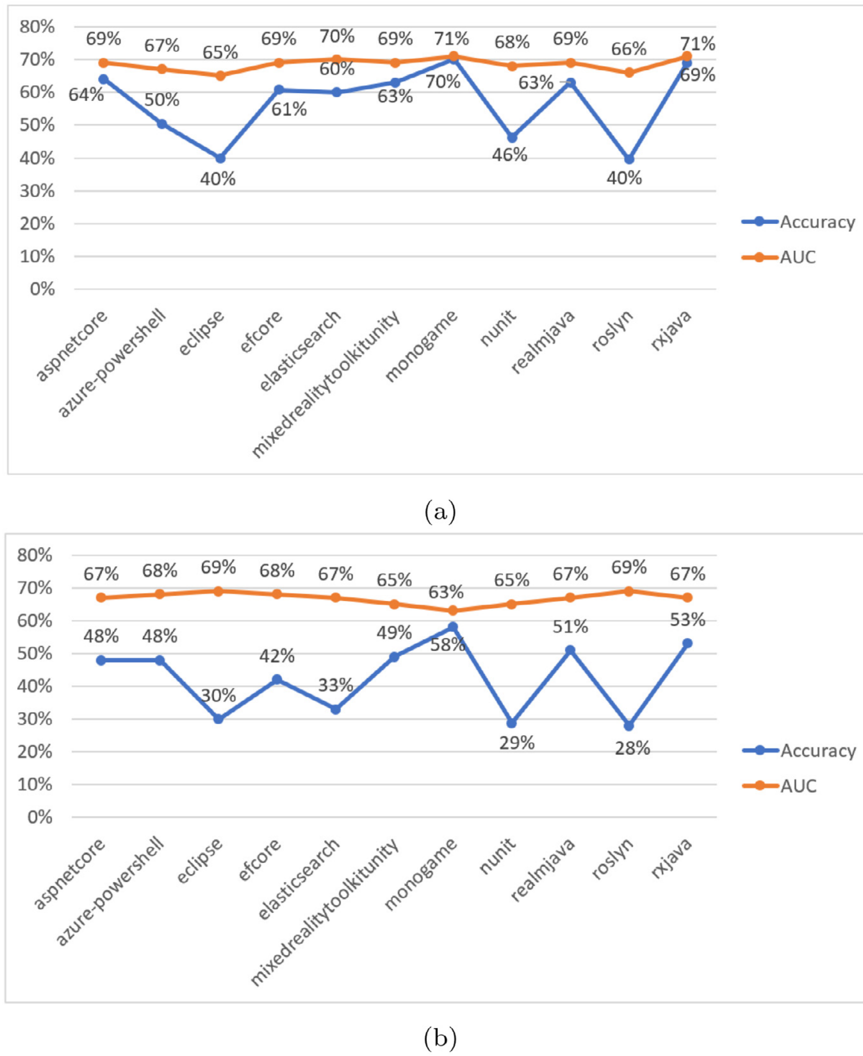


Fig. 14. Multi-triage with Data Augmentation: AUC v.s. Accuracy.

libraries, because the source code is not accessible. Nonetheless, the scikit-learn is widely used in various studies (Liang et al., 2020; Yadav et al., 2019) to set up machine learning algorithms, including SVM. Thus, there is little threat to baselines implementation. In terms of the contextual data augmentation approach, we calculated the threshold amount (30,000) using the approximate total number of issue reports from targeted projects, based on the assumption that synthetic records should not be larger than the total. Thus, the threshold value can change based on the targeted project.

Threats to construct validity. This relates to the applicability of our evaluation measurement. We use accuracy and the F2 score as the evaluation metrics that evaluate the performance of the model. They represent standard evaluation metrics for bug triage models used in previous studies (Anvik et al., 2006; Mani et al., 2019).

7. Discussion

This section discusses implications of the accuracy, precision, and recall rates we achieved on our eleven experimental projects. We also report various alternatives we have considered in implementing our model and in choosing a time-series-based cross-validation approach. Then, we further discuss the decision to use the contextual data augmentation approach in generating synthetic issue reports. Lastly, we also review the lessons we have learned in applying a deep learning approach to an issue report contextual and structural information.

7.1. Accessing the significance of our approach

Our approach achieves an average accuracy of 57% and 47% for developers and issue types, respectively. Also, our approach compromises precision and recall for both developers and issue types prediction results, with an average of 61—54% and 53—40% respectively. The only way to ensure these prediction rates are good enough for the bug triage process is by either performing a direct observation with human triagers, or by statistical analysis of the qualitative data. Our study performs qualitative analysis by categorising the results into two generic issue report types (i.e. bug and enhancements) and observing the prediction results in terms of numbers. However, we envision our approach will be evaluated with human triagers in the future. Notably, all the issue reports predicted correctly in baseline approaches are covered by our approach. In addition, our approach can correctly predict issue reports, which are missed by state-of-the-art approaches, due to our model capability to comprehend the structural context of code snippets. Therefore, we believe that the prediction rates we report in this paper for the eleven open-source projects are sufficient to assist human triagers in assigning a developer and an issue type for a new issue report. As previously mentioned, there is an average of 67 days to fix a new issue report in these projects, due to the delay in triagers becoming acquainted with the problem and finding the relevant developers. Our approach can reduce the time spent on issue report allocation tasks and regaining the time to resolve the issues.

Furthermore, our multi-triage learning model takes advantage of the multi-task learning approach to train the developers and issue types classification tasks in one model. It reduces the training time substantially, compared to a single-task learning model. However, the multi-task learning model is pruned to encounter a negative transfer learning problem if prediction tasks are not compatible for learning together. We eliminate the problem by comparing our approach with two single-task learning models. To evaluate the two single models effectively, we designed these models in the same manner as the two neural networks used in our encoder layers (i.e. BiLSTM and CNN). Notably, our model outperforms the single models in both developers and issue types prediction in precision, recall, and accuracy. Therefore, we considered a relatedness between developers and issuer types prediction tasks, as it is compatible to learn in one single prediction network.

7.2. Evaluation using time-series based cross validation

The standard method for evaluating the machine learning model is the K-fold validation approach. In the K-fold validation approach, the original sample is randomly partitioned into k equal sized sub-samples and trains the model k times repeatably. However, the standard K-fold validation approach is inappropriate in a time-ordered dataset, where the future issue reports will be used to predict past bug reports. Therefore, we followed a time-series 5-fold validation approach and trained all our models, including the baselines approach. When we used the time-series approach, we noticed that the first one-or two-fold accuracy results are relatively lower than the later folds, due to smaller data size. The neural networks-based approach generally produces better results when there is more data available to learn. However, our time-series approach statically generalised the results based on how the issue report information flows and alters an issue tracking system.

7.3. Alternative considerations on model building

We choose to use CNN in-text encoder and BiLSTM for AST encoder by referring to previous studies in similar areas (Liu and Guo, 2019; Alon et al., 2019; Mani et al., 2019). Both of the networks are commonly used in natural language and structural language processing. Alternatively, we could incorporate the BiLSTM model for text encoder or CNN for AST encoder. However, in our preliminary test, the CNN model performs better than does the BiLSTM in the text encoder layer, whereas the BiLSTM model performs better than does the CNN in the AST encoder. Therefore, we choose the combination, which produces the best results.

7.4. Applicability of contextual data augmentation approach

We adopted a supervised machine-learning approach, as our triage model required a ground truth label for each report to train the model. Therefore, we faced an imbalanced class problem in our model training. When we evaluated our model with the AUC test, we observed that our model performance is slightly low, with an average of 65% for developers and 64% for issue types. Thus, we adopted the contextual augmentation approach to generate synthetic issue reports to balance developers and issue types label distribution on training samples. In general, there are two ways to develop the synthetic reports with the contextual augmentation approach: (1) random word substitution, and (2) random word removal (Kafle et al., 2017). We selected the substitution approach, as we do not want to lose the important information of the issue report. We incrementally generated the synthetic reports using a time-series cross-validation approach and trained the model. Since we are interested in the performance of our model, we statistically evaluated the improvement of the data augmentation approach using the AUC test. Notably, our model performance rose on average 69% for developers and 67% for issue types. Therefore, we considered that the contextual augmentation approach is reasonable for smoothing label distributions in the supervised learning approach.

7.5. Lessons learned

Our approach uses textual and code snippets information from issue reports. The accuracy of our approach might be improved by incorporating additional information.

The screenshot image is a valuable asset of issue reports, providing additional information about user requirements. Also, the execution stack trace from issue reports can be used as the pointer to identify the code area in recommending issue types. As mentioned in Section 2, the GitHub projects issue types label includes project areas or components information. Identifying the project areas or components can assist in finding potential developers by looking into the list of developers who are actively working on these areas, either using the code ownership information or previous issue assignments history. However, as explained in Section 3, stack trace introduces noise into the model training, as we neglected this information. Also, correlating code ownership information to issue reports is challenging, especially for large projects evolving throughout time.

8. Related work

This section introduces previous studies related to the semi-automatic bug triage process and multi-task learning model. Moreover, other studies related to bug resolution (e.g. bug localisation) are discussed.

8.1. Semi-automatic bug triage

In an early work of Murphy and Cubranic (2004), the authors proposed an automatic bug triage approach that used a native Bayes (NB) classifier to recommend candidate developers to fix a new bug. Later, (Anvik et al., 2006) extended this by comparing the work of Murphy and Cubranic (2004) with three machine learning classifiers: NB, SVM, and C4.5. Their preliminary results found that SVM outperforms the other classifiers. In Matter et al. (2009), the authors proposed an approach to modelling developers' profiles using the vocabularies from their changed source code files, compared with terms from issue reports to rank the relevant developers.

A comparison of different machine learning algorithms (i.e. NB, SVM, EM, conjunction rules, and nearest neighbours) to recommend potential developers can be found in Anvik and Murphy (2011). In general, the authors used project-specific heuristics to construct a label for each issue report rather than using the assigned-to field, in order to eliminate default assignee assignment and duplicate reports with unchanged assigned-to field problems. In our approach, we alternatively address these problems by filtering out issues assigned to *software bots* and including the corresponding pull request's developer information as the tossing sequence in our labelling process.

Similarly, Zhang and Lee (2012) proposed a concept profile and social network-based bug triage model to rank expert developers to fix a bug. In their work, a concept profiling first defines the topic terms to cluster the issue reports. Then, the social network feature captures a set of developers' collaborative relationships, extracted from the concept profiles, to rank the candidate developers based on the level of expertise (i.e. a fixer of a bug, a contributor of a bug). In Park et al. (2016), the authors proposed CosTriage to assist triagers in finding the candidate developers who can fix the bug in the shortest time frame. CosTriage adopts content-boosted collaborative filtering (CBCF), which combines issue report similarity scores with each developer's bug fixing time to recommend relevant developers for a new issue report.

Aside from developer recommendation studies, other studies have focused on automating issue type prediction in the bug triage process. In Xia et al. (2013), the authors proposed TagCombine, an automatic tag recommendation method, which is based on a composite ranking approach to analysing information in software forum sites (i.e. Stack Overflow, free-code). TagCombine consists of three ranking components: multi-label ranking, similarity-based ranking, and tag-terms-based ranking. In their approach, multi-nominal NB classifier, Euclidean distance algorithm, and latent semantic indexing (LSI) are used to calculate three component scores separately. The linear combination score of these three components is then used to recommend the list of relevant tags for a new issue report. In Xia et al. (2014), the authors proposed MLL-GA, a composite method to classify crash reports and failures. MLL-GA adopts various multi-label learning algorithms and generic algorithms to identify faults from crash reports automatically.

In the work of Zhang et al. (2016), the author adopted a BM25-based textual similarity algorithm and KNN to predict severity levels and developers for a new issue report. In Alonso-Abad et al. (2019), the authors adopted machine learning classifiers, such as NB and SVM, to predict an issue label (e.g. bug, enhancement) for a new issue report. Their approach uses the bag-of-words model to represent issue reports in text classification. In this representation, every word in the training corpus is considered a feature; therefore, each issue report presents as a sparse representation with a high number of features. These features are used by machine learning classifiers to predict issue labels for new issue reports. Recently, in Mani et al. (2019), the author used the BiLSTM model to recommend potential developers.

Our work is closely related to that of Mani et al. (2019). However, our multi-triage model adopts a multi-task learning approach and recommends both developers and issue types from one learning model. As such, it reduces a considerable amount of training time in comparison to the single task learning model. In addition, our model uses both textual and structural information (i.e. code snippets) to learn the representation of issue reports, as doing so provides a more accurate representation. In comparison, previous studies have neglected the code snippet information in order to reduce noises in the model training. In our approach, we transform the code snippet to AST paths and learn the representation in a separate encoder, which eliminates the risk of introducing noises in the model.

There are several techniques to parse AST from partial programs. Some of the well known approaches are fuzzy parsers (Koppler, 1997), island grammars (Moonen, 2001), partial program analysis (PPA) (Dagenais and Hendren, 2008) and pairwise paths (Alon et al., 2019). Fuzzy parsers scan the code keywords and extract the coarse-grained structure out of code snippets (Koppler, 1997). Similarly, island grammars extract part of the code snippets that describes some details of the function (island) and ignores the rest of the trivial lexical information (water). In contrast, PPA parsers trace the defined type of a class or method and extract a typed AST (Dagenais and Hendren, 2008). PPA recovers the declared type of expressions by resolving declaration ambiguities in partial java programs. Declaration ambiguity refers to the fields whose declarations are undeclared, or to the unqualified external references. These approaches are more suitable for situations where a sound analysis is required, such as code cloning, code representation and code summarisation.

Lastly, in the pairwise paths parser (Alon et al., 2019), the AST paths are extracted using modern integrated development environments (IDE) (e.g., Eclipse), which generate the pairwise paths between terminal nodes (e.g. variable declaration) by neglecting the non-terminal nodes (e.g. do-while loop). In the pairwise paths approach, two programs that have similar terminal nodes are likely to parse as similar format. As it is our intention to compare similar code snippet between issue reports, we have adopted this pairwise paths approach in our study. Next, we discuss the related work of the multi-task learning model.

8.2. Multi-task learning

The multi-task learning model has been successfully applied in computer vision applications as well as in many natural language problems which require solving multiple tasks simultaneously (Lu et al., 2017; Shinohara, 2016). In the recent work of Kokkinos (2017), the authors used hard parameter sharing to address seven computer vision tasks. Similar works are presented in Dvornik et al. (2017), Bilen and Vedaldi (2016), Zhou et al. (2017). In the work of Standley et al. (2020), the authors proposed a framework by which to evaluate which tasks are compatible with learning jointly in the multi-task learning network. Their preliminary results revealed that multi-task learning networks' prediction quality depends on the relationship between the jointly trained tasks. Their framework incrementally increases the number of tasks assigning to the model by starting with three or fewer networks. They used predefined inference time, and the lowest total lost value to identify the compatible pairing tasks.

In Deng et al. (2019), the authors used a multi-task learning approach to tackle two types of question-answering tasks: answer selection and knowledge-based question answering. In their approach, the CNN network is used to model the shared learning layer in order to learn the contextual information of historical question and answer data to predict answers to a new question automatically. In a similar line of work, the authors of Zhang et al. (2014) used the CNN network to identify facial landmarks and attributes (i.e. emotions). In Liu et al. (2015), the authors adopted a multi-task learning network to learn query classification tasks and ranking of web search tasks together. Our work is similar to that of Deng et al. (2019), but we tackle a problem in a different domain. We adopted multi-task learning with a hard-parameter sharing approach to recommend potential developers and issue types for a new issue report.

8.3. Other tasks in the bug resolution process

In Cabot et al. (2015b), the authors reported the usage of GitHub's label in over 3 million GitHub projects. Their preliminary results revealed that most projects use four generic types of labelling strategies: priority labels, versioning labels, workflow labels, and architecture labels to categorise the issue reports. In Polisetty et al. (2019), the authors proposed a CNN-based bug localisation model to assist developers in identifying code smell areas. In the work of Deshmukh et al. (2017), the researchers leverage deep neural networks to detect duplicate issue reports automatically.

Likewise, Tufano et al. (2018) relied on recurrent neural networks (RNN) and graph embedding to detect similarities in source code components. The work proposed in White et al. (2019) used deep learning neural networks to identify similar code components in generating bug-fixing patches for program repair. In Wan et al. (2018), an LSTM encoder–decoder was used to generate a code summary that provided a high-level description of code functionality changes. Despite different strategies, these approaches use AST tokens as embedding input to learn the representation of source code components. Instead, the work in Sui et al. (2020) and Cheng et al. used the control flow graph (CFG) representation of a program to embed the code to support a variety of program analysis tasks (e.g. code summarisation and semantic labelling).

9. Conclusion

In this paper, we have presented an approach to recommend potential developers and issue types of an issue report to resolve the issue. Our approach uses the multi-task learning approach to simultaneously resolve the developer's assignment and issue types allocation tasks.

We use a text encoder and AST encoder to learn the precise representation of issue reports. The experiments are conducted on eleven widely-used open-source projects and achieve accuracy on average of 57% for developer and 42% for issue types, respectively. Furthermore, we present the effectiveness of the contextual data augmentation approach in balancing the disproportional ratio of class labels. In addition, we introduced a qualitative analysis of our machine learning model against state-of-the-art approaches. We have reported on lessons learned in processing the issue report data from the issue tracking system.

We believe that our approach is promising for the leveraging bug assignment and the tossing process for open-source software developments. An interesting future direction includes experiments using our approach with human bug triagers and investigating the additional information of issue reports (e.g. screenshots and comments).

CRediT authorship contribution statement

Yulei Sui: Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We thank the anonymous reviewers for their reviews and suggestions. This research is partially supported by Australian Research Grants DP200101328 and DP210101348.

References

Achille, A., Soatto, S., 2018. Information dropout: Learning optimal representations through noisy computation. IEEE Trans. Pattern Anal. Mach. Intell. 40 (12), 2897–2905.

Alaeddine, H., Jihene, M., 2021. Deep network in network. Neural Comput. Appl. 33 (5), 1453–1465. http://dx.doi.org/10.1007/s00521-020-05008-0.

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019. Code2vec: Learning distributed representations of code. Proc. ACM Program. Lang. 3 (POPL), http://dx.doi.org/10.1145/3290353.

Alonso-Abad, J.M., López-Nozal, C., Maudes-Raedo, J.M., Marticorena-Sánchez, R., 2019. Label prediction on issue tracking systems using text mining. Prog. Artif. Intell. 8 (3), 325–342.

Anvik, J., Hiew, L., Murphy, G.C., 2006. Who should fix this bug? In: Proceedings of the 28th International Conference on Software Engineering, pp. 361–370.

Anvik, J., Murphy, G.C., 2011. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. ACM Trans. Softw. Eng. Methodol. 20 (3), 1–35.

Aung, T.W.W.A., 2021. MultiTriage. http://dx.doi.org/10.5281/zenodo.5532458, URL https://github.com/thazin31086/MultiTriage.

Banerjee, I., Ling, Y., Chen, M.C., Hasan, S.A., Langlotz, C.P., Moradzadeh, N., Chapman, B., Amrhein, T., Mong, D., Rubin, D.L., Farri, O., Lungren, M.P., 2019. Comparative effectiveness of convolutional neural network (CNN) and recurrent neural network (RNN) architectures for radiology text report classification. Artif. Intell. Med. 97, 79–88. http://dx.doi.org/10.1016/j.artmed.2018.11.004, URL http://www.sciencedirect.com/science/article/pii/S0933365717306255.

Banerjee, S., Syed, Z., Helmick, J., Culp, M., Ryan, K., Cukic, B., 2017. Automated triaging of very large bug repositories. Inf. Softw. Technol. 89, 1–13.

Bengio, Y., 2012. Practical recommendations for gradient-based training of deep architectures. Arxiv.

Bergmeir, C., Benítez, J.M., 2012. On the use of cross-validation for time series predictor evaluation. Inform. Sci. 191, 192–213. http://dx.doi.org/10.1016/j.ins.2011.12.028, Data Mining for Software Trustworthiness. URL https://www.sciencedirect.com/science/article/pii/S0020025511006773.

Bhattacharya, P., Neamtiu, I., 2010. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In: 2010 IEEE International Conference on Software Maintenance. IEEE, pp. 1–10.

Biedenkapp, A., Lindauer, M., Eggensperger, K., Hutter, F., Fawcett, C., Hoos, H., 2017. Efficient parameter importance analysis via ablation with surrogates. In: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 31.

Bilen, H., Vedaldi, A., 2016. Integrated perception with recurrent multi-task neural networks. In: Advances in Neural Information Processing Systems. pp. 235–243.

Cabot, J., Izquierdo, J.L.C., Cosentino, V., Rolandi, B., 2015a. Exploring the use of labels to categorize issues in open-source software projects. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering. SANER, IEEE, pp. 550–554.

Cabot, J., Izquierdo, J.L.C., Cosentino, V., Rolandi, B., 2015b. Exploring the use of labels to categorize issues in open-source software projects. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering. SANER, IEEE, pp. 550–554.

Cai, L., Zhou, S., Yan, X., Yuan, R., 2019. A stacked BiLSTM neural network based on coattention mechanism for question answering. Comput. Intell. Neurosci. 2019.

Caruana, R., 1993. Multitask learning: A knowledge-based source of inductive bias. In: Proceedings of the Tenth International Conference on Machine Learning. Morgan Kaufmann, pp. 41–48.

Catolino, G., Palomba, F., Zaidman, A., Ferrucci, F., 2019. Not all bugs are the same: Understanding, characterizing, and classifying bug types. J. Syst. Softw. 152, 165–181. http://dx.doi.org/10.1016/j.jss.2019.03.002, URL https://www.sciencedirect.com/science/article/pii/S0164121219300536.

Cheng, X., Wang, H., Hua, J., Xu, G., Sui, Y., 2021. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network, 30, 3, Association for Computing, New York, NY, USA Machinery, 38, https://doi.org/10.1145/3436877.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., Kuksa, P., 2011. Natural language processing (almost) from scratch. J. Mach. Learn. Res. 12 (ARTICLE), 2493–2537.

Dagenais, B., Hendren, L., 2008. Enabling static analysis for partial java programs. SIGPLAN Not. 43 (10), 313–328. http://dx.doi.org/10.1145/1449955.1449790, URL https://doi-org.ezproxy.lib.uts.edu.au/10.1145/1449955.1449790.

Deng, Y., Xie, Y., Li, Y., Yang, M., Du, N., Fan, W., Lei, K., Shen, Y., 2019. Multi-task learning with multi-view attention for answer selection and knowledge base question answering. In: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 33, pp. 6318–6325.

Deshmukh, J., Podder, S., Sengupta, S., Dubash, N., et al., 2017. Towards accurate duplicate bug retrieval using deep learning techniques. In: 2017 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 115–124.

Dvornik, N., Shmelkov, K., Mairal, J., Schmid, C., 2017. Blitznet: A real-time deep network for scene understanding. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 4154–4162.

Fawcett, C., Hoos, H.H., 2016. Analysing differences between algorithm configurations through ablation. J. Heuristics 22 (4), 431–458.

Golzadeh, M., Legay, D., Decan, A., Mens, T., 2020. Bot or not? Detecting bots in GitHub pull request activity based on comment similarity. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, pp. 31–35.

Gousios, G., Pinzger, M., Deursen, A.v., 2014. An exploratory study of the pull-based software development model. In: Proceedings of the 36th International Conference on Software Engineering, pp. 345–355.

Graves, A., Schmidhuber, J., 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. Neural Netw. 18 (5), 602–610. http://dx.doi.org/10.1016/j.neunet.2005.06.042, IJCNN 2005. URL http://www.sciencedirect.com/science/article/pii/S0893608005001206.

Hassan, A.E., Xie, T., 2010. Software intelligence: the future of mining software engineering data. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, pp. 161–166.

Jiang, G., Wang, W., 2017. Error estimation based on variance analysis of k-fold cross-validation. Pattern Recognit. 69, 94–106.

Jiang, J., Wu, Q., Cao, J., Xia, X., Zhang, L., 2020. Recommending tags for pull requests in GitHub. Inf. Softw. Technol. 106394.

Kafle, K., Yousefhussien, M., Kanan, C., 2017. Data augmentation for visual question answering. In: Proceedings of the 10th International Conference on Natural Language Generation, pp. 198–202.

Kagdi, H., Gethers, M., Poshyvanyk, D., Hammad, M., 2012. Assigning change requests to software developers. J. Softw. Evol. Process 24 (1), 3–33.

Kenton, J.D.M.-W.C., Toutanova, L.K., 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of NAACL-HLT, pp. 4171–4186.

Kim, Y., 2014. Convolutional neural networks for sentence classification. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing. EMNLP, Association for Computational Linguistics, Doha, Qatar, pp. 1746–1751. http://dx.doi.org/10.3115/v1/D14-1181, URL https://www.aclweb.org/anthology/D14-1181.

Kim, S., Zhao, J., Tian, Y., Chandra, S., 2021. Code prediction by feeding trees to transformers. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering. ICSE, pp. 150–162. http://dx.doi.org/10.1109/ICSE43902.2021.00026.

Kokkinos, I., 2017. Ubernet: Training a universal convolutional neural network for low-, mid-, and high-level vision using diverse datasets and limited memory. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 6129–6138.

Koppler, R., 1997. A systematic approach to fuzzy parsing. Softw. - Pract. Exp. 27 (6), 637–649.

Lee, S.-R., Heo, M.-J., Lee, C.-G., Kim, M., Jeong, G., 2017. Applying deep learning based automatic bug triager to industrial projects. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 926–931.

Lee, H., Kwon, H., 2017. Going deeper with contextual CNN for hyperspectral image classification. IEEE Trans. Image Process. 26 (10), 4843–4855. http://dx.doi.org/10.1109/TIP.2017.2725580.

Liang, X., Jiang, A., Li, T., Xue, Y., Wang, G., 2020. LR-SMOTE–An improved unbalanced data set oversampling based on K-means and SVM. Knowl.-Based Syst. 105845.

Liu, X., Gao, J., He, X., Deng, L., Duh, K., Wang, Y.-Y., 2015. Representation learning using multi-task deep neural networks for semantic classification and information retrieval.

Liu, G., Guo, J., 2019. Bidirectional LSTM with attention mechanism and convolutional layer for text classification. Neurocomputing 337, 325–338.

Lu, Y., Kumar, A., Zhai, S., Cheng, Y., Javidi, T., Feris, R., 2017. Fully-adaptive feature sharing in multi-task networks with applications in person attribute classification. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 5334–5343.

Mani, S., Sankaran, A., Aralikatte, R., 2019. Deeptriage: Exploring the effectiveness of deep learning for bug triaging. In: Proceedings of the ACM India Joint International Conference on Data Science and Management of Data, pp. 171–179.

Matter, D., Kuhn, A., Nierstrasz, O., 2009. Assigning bug reports using a vocabulary-based expertise model of developers. In: 2009 6th IEEE International Working Conference on Mining Software Repositories. IEEE, pp. 131–140.

Moonen, L., 2001. Generating robust parsers using island grammars. In: Proceedings Eighth Working Conference on Reverse Engineering. IEEE, pp. 13–22.

Murphy, G., Cubranic, D., 2004. Automatic bug triage using text categorization. In: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering. Citeseer, pp. 1–6.

Mustafa, H.T., Yang, J., Zareapoor, M., 2019. Multi-scale convolutional neural network for multi-focus image fusion. Image Vis. Comput. 85, 26–35. http://dx.doi.org/10.1016/j.imavis.2019.03.001, URL http://www.sciencedirect.com/science/article/pii/S026288561930023X.

Nair, V., Hinton, G.E., 2010. Rectified linear units improve restricted Boltzmann machines. In: Proceedings of the 27th International Conference on International Conference on Machine Learning. ICML'10, Omni Press, USA, pp. 807–814, URL http://dl.acm.org/citation.cfm?id=3104322.3104425.

Park, J.-w., Lee, M.-W., Kim, J., Hwang, S.-w., Kim, S., 2016. Cost-aware triage ranking algorithms for bug reporting systems. Knowl. Inf. Syst. 48 (3), 679–705.

Pennington, J., Socher, R., Manning, C.D., 2014. Glove: Global vectors for word representation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP, pp. 1532–1543.

Polisetty, S., Miranskyy, A., Başar, A., 2019. On usefulness of the deep-learning-based bug localization models to practitioners. In: Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering. ACM, pp. 16–25.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., 2019. Language models are unsupervised multitask learners. OpenAI Blog 1 (8), 9.

Runeson, P., Alexandersson, M., Nyholm, O., 2007. Detection of duplicate defect reports using natural language processing. In: 29th International Conference on Software Engineering. ICSE'07, IEEE, pp. 499–510.

Shinohara, Y., 2016. Adversarial multi-task learning of deep neural networks for robust speech recognition. In: Interspeech. San Francisco, CA, USA, pp. 2369–2372.

Standley, T., Zamir, A.R., Chen, D., Guibas, L., Malik, J., Savarese, S., 2020. Which tasks should be learned together in multi-task learning? URL https://openreview.net/forum?id=HJlTpCEKvS.

Sui, Y., Cheng, X., Zhang, G., Wang, H., 2020. Flow2Vec: Value-flow-based precise code embedding. Proc. ACM Program. Lang. 4 (OOPSLA), http://dx.doi.org/10.1145/3428301.

Tamrawi, A., Nguyen, T.T., Al-Kofahi, J.M., Nguyen, T.N., 2011. Fuzzy set and cache-based approach for bug triaging. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ESEC/FSE '11, Association for Computing Machinery, New York, NY, USA, pp. 365–375. http://dx.doi.org/10.1145/2025113.2025163.

Tufano, M., Watson, C., Bavota, G., Di Penta, M., White, M., Poshyvanyk, D., 2018. Deep learning similarities from different representations of source code. In: Proceedings of the 15th International Conference on Mining Software Repositories. MSR '18, ACM, New York, NY, USA, pp. 542–553. http://dx.doi.org/10.1145/3196398.3196431, URL http://doi.acm.org/10.1145/3196398.3196431.

Wan, Y., Zhao, Z., Yang, M., Xu, G., Ying, H., Wu, J., Yu, P.S., 2018. Improving automatic source code summarization via deep reinforcement learning. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, pp. 397–407.

Wang, X., Zhang, L., Xie, T., Anvik, J., Sun, J., 2008. An approach to detecting duplicate bug reports using natural language and execution information. In: Proceedings of the 30th International Conference on Software Engineering, pp. 461–470.

White, M., Tufano, M., Martinez, M., Monperrus, M., Poshyvanyk, D., 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 479–490.

Xi, S., Yao, Y., Xiao, X., Xu, F., Lu, J., 2018. An effective approach for routing the bug reports to the right fixers. In: Proceedings of the Tenth Asia-Pacific Symposium on Internetware, pp. 1–10.

Xi, S.-Q., Yao, Y., Xiao, X.-S., Xu, F., Lv, J., 2019. Bug triaging based on tossing sequence modeling. J. Comput. Sci. Tech. 34 (5), 942–956.

Xia, X., Feng, Y., Lo, D., Chen, Z., Wang, X., 2014. Towards more accurate multi-label software behavior learning. In: 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering. CSMR-WCRE, IEEE, pp. 134–143.

Xia, X., Lo, D., Ding, Y., Al-Kofahi, J.M., Nguyen, T.N., Wang, X., 2016a. Improving automated bug triaging with specialized topic model. IEEE Trans. Softw. Eng. 43 (3), 272–297.

Xia, X., Lo, D., Ding, Y., Al-Kofahi, J.M., Nguyen, T.N., Wang, X., 2016b. Improving automated bug triaging with specialized topic model. IEEE Trans. Softw. Eng. 43 (3), 272–297.

Xia, X., Lo, D., Wang, X., Zhou, B., 2013. Tag recommendation in software information sites. In: 2013 10th Working Conference on Mining Software Repositories. MSR, IEEE, pp. 287–296.

Yadav, A., Singh, S.K., Suri, J.S., 2019. Ranking of software developers based on expertise score for bug triaging. Inf. Softw. Technol. 112, 1–17.

Yu, Y., Wang, H., Yin, G., Wang, T., 2016. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? Inf. Softw. Technol. 74, 204–218. http://dx.doi.org/10.1016/j.infsof.2016.01.004, URL http://www.sciencedirect.com/science/article/pii/S0950584916000069.

Zhang, T., Chen, J., Yang, G., Lee, B., Luo, X., 2016. Towards more accurate severity prediction and fixer recommendation of software bugs. J. Syst. Softw. 117, 166–184.

Zhang, T., Lee, B., 2012. An automated bug triage approach: A concept profile and social network based developer recommendation. In: International Conference on Intelligent Computing. Springer, pp. 505–512.

Zhang, Z., Luo, P., Loy, C.C., Tang, X., 2014. Facial landmark detection by deep multi-task learning. In: European Conference on Computer Vision. Springer, pp. 94–108.

Zhou, D., Wang, J., Jiang, B., Guo, H., Li, Y., 2017. Multi-task multi-view learning based on cooperative multi-objective optimization. IEEE Access 6, 19465–19477.