# A Light Bug Triage Framework for Applying Large Pre-trained Language Model

Jaehyung Lee
anthi7@postech.ac.kr
Pohang University of Science and Technology (POSTECH)
Republic of Korea

Kisun Han
ksun.han@samsung.com
Samsung Research
Republic of Korea

Hwanjo Yu*
hwanjoyu@postech.ac.kr
Pohang University of Science and Technology (POSTECH)
Republic of Korea

## ABSTRACT

Assigning appropriate developers to the bugs is one of the main challenges in bug triage. Demands for automatic bug triage are increasing in the industry, as manual bug triage is labor-intensive and time-consuming in large projects. The key to the bug triage task is extracting semantic information from a bug report. In recent years, large Pre-trained Language Models (PLMs) including BERT [4] have achieved dramatic progress in the natural language processing (NLP) domain. However, applying large PLMs to the bug triage task for extracting semantic information has several challenges. In this paper, we address the challenges and propose a novel framework for bug triage named **LBT-P**, standing for **L**ight **B**ug **T**riage framework with a **P**re-trained language model. It compresses a large PLM into small and fast models using knowledge distillation techniques and also prevents catastrophic forgetting of PLM by introducing knowledge preservation fine-tuning. We also develop a new loss function exploiting representations of earlier layers as well as deeper layers in order to handle the overthinking problem. We demonstrate our proposed framework on the real-world private dataset and three public real-world datasets [11]: Google Chromium, Mozilla Core, and Mozilla Firefox. The result of the experiments shows the superiority of LBT-P.

## KEYWORDS

Bug triage, Pre-trained language model, BERT, Knowledge distillation, Catastrophic forgetting, Overthinking

**ACM Reference Format:**
Jaehyung Lee, Kisun Han, and Hwanjo Yu. 2022. A Light Bug Triage Framework for Applying Large Pre-trained Language Model. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3551349.3556898

## 1 INTRODUCTION

During the development and maintenance of large projects, numerous bugs are reported. These bugs often cause serious problems in the projects, so it is important to fix them quickly. Bug triage helps to manage all reported issues properly. Triage, which is a term from medicine, is a process of assigning priority to patients' treatments based on their conditions and circumstances. Triage helps to treat as many patients as possible with limited resources. Bug triage is a process of prioritizing the bugs and assigning appropriate developers to the bugs. When the project is small, assigning developers to bugs is not a big deal. However, as the project grows, it becomes more difficult to find suitable developers to fix bugs. This is because each developer's skills are various, and each bug requires different skills. Sometimes a bug even requires multiple skills to fix. Thus, assigning developers to bugs is the primary task in bug triage. In this paper, we aim to deal with developer assignments.

Assigning developers is usually done manually in the industry, and manual assignment is labor-intensive and time-consuming. Nowadays, it is easy to find a project with over 1,000 developers, and new bugs are reported every single day in the project. When a bug report is registered, a human expert first identifies the characteristic of the bug and assigns the developer with the most suitable skills from thousands of developers. If this process is not done quickly, other upcoming bugs cannot be handled. Manual bug triage often significantly delays the development and maintenance of large-scale systems.

To overcome the delay of manual bug triage, automatic bug triage techniques have been studied and these aim to capture useful information from bug reports. Early studies [3, 6, 22] tried to extract traditional textual features such as TF-IDF from a bug report. Word2vec [13] is adopted to get text representations [11], and ELMo [15] is applied to the bug triage task [25].

Recently, Transformer-based large Pre-trained Language Models (PLMs), e.g., BERT [4] and its variants, have achieved state-of-the-art performance on Natural Language Understanding (NLU) tasks. However, applying large PLMs to NLP applications including bug triage is challenging: First, large PLMs require high-end computing resources to be deployed in practice, as their size is typically larger than the size of memory in commodity hardware. Second, during fine-tuning of a PLM on bug triage data, the PLM forgets general language knowledge learned in pre-training. This phenomenon is called *catastrophic forgetting* [12], i.e., a phenomenon that a neural network forgets previously learned information when it learns new information. Finally, when the representation of an input at the network's earlier layer is sufficient to make a correct prediction, progressing into deeper layers could "overthink" the input and deteriorate the prediction performance, which is called the *overthinking*

*Corresponding author.


*ASE '22, October 10–14, 2022, Rochester, MI, USA*

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556898

problem [7]. Likewise, humans, thinking too complex about a simple problem, can also lead to incorrect conclusions. In this paper, we aim to solve these challenges.

This paper proposes a novel framework for bug triage named **LBT-P**, standing for **L**ight **B**ug **T**riage framework with a **P**re-trained language model. **LBT-P** handles the aforementioned problems and captures semantic information from a bug report. First, it compresses a large PLM into small and fast models using knowledge distillation techniques [5]. To handle catastrophic forgetting of PLM, we develop a fine-tuning method preserving language knowledge. To handle the overthinking problem, we propose new loss functions exploiting representations of earlier layers as well as deeper layers. We evaluate our methods on a real-world private dataset and three public datasets: Google Chromium, Mozilla Core, and Mozilla Firefox. The public datasets were used in the previous study [11]. These datasets have 118,607, 128,104, and 24,158 bug reports respectively. Experimental results show that our proposing methods significantly outperform competitors including [11].

The main contributions of our work are summarized as follows:

- We deal with challenges for applying PLM. First, we compress the large PLM into a smaller and faster model using knowledge distillation. Also, we prevent catastrophic forgetting using knowledge preservation fine-tuning.
- We address the overthinking problem by exploiting representations of earlier layers. We propose a new loss function to use earlier representations effectively.
- The empirical result shows the superiority of the proposed framework on real-world datasets.

## 2 PRELIMINARY

### 2.1 Bug Triage Method

There have been various attempts to automate bug triage tasks. Since these studies use text in a bug report as input, bug triage research has been greatly influenced by the development of the NLP technique. Early studies exploited traditional text features, which are about statistics in a document, from the bug report. For example, TF-IDF, the most representative traditional text feature, is multiply of term frequency and inverse document frequency. Xuan et al. [22] extracted traditional text features, reduced train data size by a combination of instance selection with feature selection, and applied Naïve Bayes classifier to assign developer. Dedík et al. [3] extracted TF-IDF from the bug reports and adopted SVM as a classifier. Jonsson et al. [6] used TF-IDF and introduced an ensemble learner called Stacked Generalization to boost performance.

Since knowing the context within a sentence is a great help in understanding the sentence, various techniques to capture context have been introduced in NLP. Recent bug triage studies have utilized these NLP techniques. Mani et al. [11] adopted Word2vec [13], which is a technique for vectorizing words, to get text representations and used bidirectional RNN with attention as a classifier. Although Word2vec learns word representations so that words in a sentence can predict nearby words well, it cannot capture the context change according to sentence, because Word2vec does not generate different word vectors from sentence to sentence. Zaidi et al. [25] applied ELMo [15], an early pre-trained language model, to the bug triage task. In contrast to Word2vec, ELMo takes the entire sentence as input, and reflects the context of the given sentence in the word representation. Thus, it can capture context change according to sentence, and also it is possible to distinguish between homonyms.

### 2.2 Large Pre-trained Language Model

In recent years, studies of large Pre-trained Language Models (PLMs) have achieved great progress in the NLP domain. These studies adopted the Transformer-based [19] model as a language model. The self-attention used in the Transformer can capture semantic information, which is information about the meaning of a sentence, at a higher level than previous NLP methods. Also, the studies of large PLMs applied self-supervised learning to train the model. In self-supervised learning used in these studies, pseudo-labels are generated from unlabeled data, and the model is trained in a supervised manner using pseudo-labels. For example, BERT [4], the most representative PLM, uses Masked Language Model (MLM) and Next Sentence Prediction (NSP) to train the language model. In the case of MLM, random words (or tokens) in the input sentence are masked and the original words are used as labels. During the training, the language model predicts what the masked words are. Using self-supervised learning, PLM can learn general language knowledge from a huge unlabeled corpus.

After BERT is introduced, various BERT-based models are proposed to improve BERT. RoBERTa [10] boosts the performance of BERT by applying dynamic masking to MLM, removing NSP, and increasing the size of training data. ALBERT [9] reduces the size of BERT by factorization of the embedding parameters and sharing all parameters across layers. ELECTRA [2] improves the performance of BERT by introducing a new pre-training task called Replaced Token Detection (RTD).

Before the advent of large PLM, to deal with NLP applications, researchers had to collect large text corpus and train a language model from the scratch with their limited corpus. This process takes a large time and a huge computational cost. Since the large PLM can can capture high-quality semantic information from text using pre-trained general language knowledge, researchers can skip the complex process to get a language model. Thus, a large PLM allows researchers to focus on their target task.

## 3 THE PROPOSED FRAMEWORK

In this section, we propose a novel **L**ight **B**ug **T**riage framework with a large **P**re-trained language model called **LBT-P**. Figure 1 shows the basic architecture of the proposed bug triage framework. The overall model is mainly composed of two parts, which are a text embedding module and a classifier. The text embedding module processes raw text. After the process, the classifier generates a score for each developer using the representations processed by the text embedding module.

In the sections that follow, we present the details of our proposed framework. We define our task in Section 3.1. In Section 3.2, we present detail of the text embedding module. In Section 3.3, we present detail of the classifier. Then, we present challenges and their solutions for our text embedding module in Section 3.4 and Section 3.5. In Section 3.6, we present a solution for dealing with the

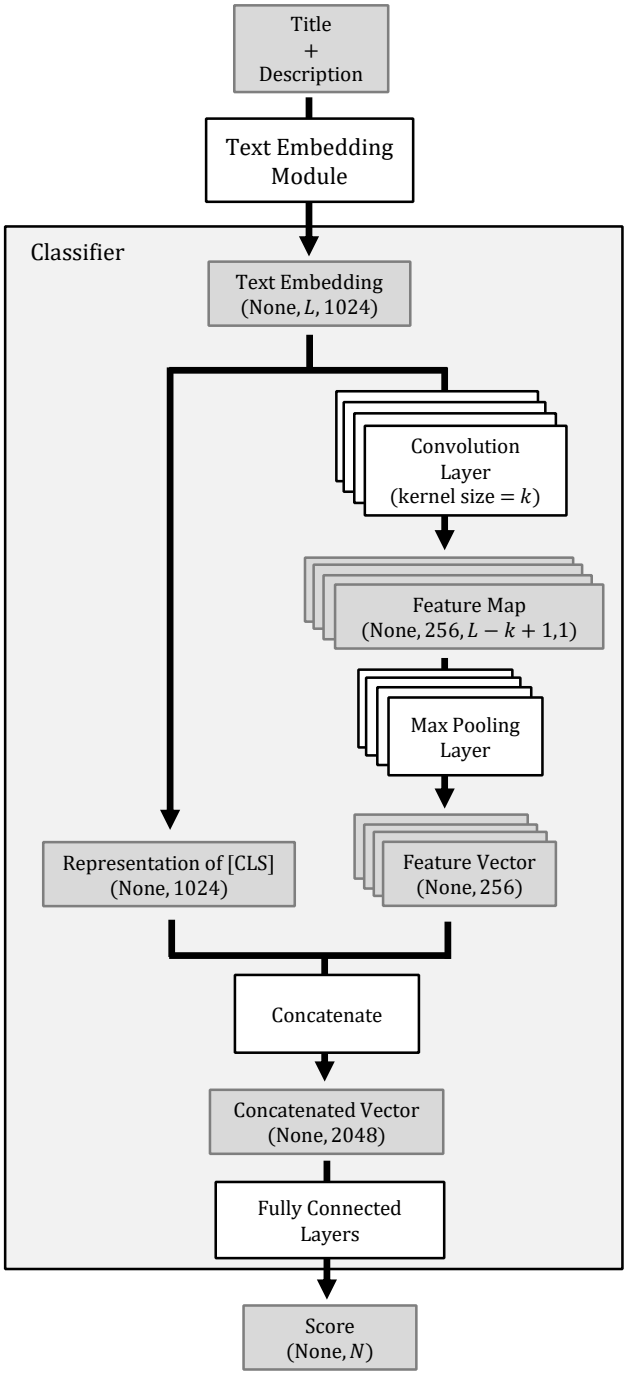



**Figure 1: The basic architecture of the proposed bug triage framework. The framework consists of a text embedding module and classifier. $L$ is the max length of the bug report and $N$ is the total number of developers. Four kernels with $k \in \{3, 4, 5, 6\}$ are used for the convolution layers.**

overthinking problem, a problem separate from the text embedding module.

## 3.1 Problem Formulation

In this work, we focus on the bug triage task, which assigns an appropriate developer to the given bug. Each bug in the dataset $D$ consists of a pair of the bug report and the developer who fixed the bug. The max length of the bug report is $L$ and the total number of developers is $N$. This bug triage task can be considered a classification problem. In this case, the bug report can be regarded as an input and the developer can be regarded as a label. Given a bug report, the bug triage model computes the score for each developer.

In practice, not only one person but also several developers can fix a bug. However, only one developer who actually fixed the bug is recorded in the dataset. If performance is evaluated by accuracy, the case where another developer who is likely to fix the bug gets the highest score is ignored. For this reason, it is inappropriate to use accuracy as an evaluation metric in this task, so we evaluate performance by top-$k$ accuracy (acc@$k$). Acc@$k$ is formulated as follows:

$$acc@k = \frac{\sum_{(x,y)\in D} \mathbf{1}_{rec_x@k}(y)}{|D|}, \tag{1}$$

where $x$ is a bug report, $y$ is label of the bug report, $rec_x@k$ is the $k$ recommended developers for the given bug report $x$, and $\mathbf{1}_A(b)$ is an indicator function that returns 1 if $b$ belongs to A and 0 otherwise. Acc@$k$ counts the number of times a developer who fixed the bug belongs to the $k$ developers with higher scores.

## 3.2 Text Embedding Module

The text embedding module converts bug reports into text embedding vectors. Bug reports registered in the bug tracking system usually contain title and description, and the title and description contain rich semantic information. The objective of the text embedding module can be regarded as capturing semantic information from the title and description. Intuitively, the titles and descriptions are closely related, and this relationship should be considered in order to properly extract semantic information. When the title and description are embedded separately, an additional process is required to capture the interaction between them. Thus, we simply concatenate the title and the description into one text and used it as an input for the text embedding module. Then, the text embedding module treats the concatenated title and description as consecutive sentences, so it can consider the correlation between the title and description without extra process. Before putting the concatenated text as input, we remove special characters, extra spaces, line breaks, and URLs from the text.

To extract semantic information from a bug report, we adopt a large PLM as a text embedding module. To find the best performing PLM on our task, we compared the performance of BERT, RoBERTa, ALBERT, and ELECTRA. Detailed results of this comparison are reported in Section 4.4. RoBERTa outperformed other competitors in this comparison, so we adopt RoBERTa-large, increased size of RoBERTa, as the text embedding module. We obtained a pre-trained RoBERTa-large model from the HuggingFace Transformers library [21]. RoBERTa-large, with the same architecture as BERT-large, has 24 layers, 1024-hidden size, 16 heads, and 355M parameters.

RoBERTa adopts byte-level BPE, first used in GPT-2 [16], as a tokenizer. Since byte-level BPE is a universal encoding scheme, it is free from the out-of-vocabulary problem. In particular, bug reports in some domains contain technical terms that are not commonly used and these terms may have important semantic information. Thus the out-of-vocabulary problem may cause performance degradation, so it is better to avoid it for practical use. From this point of view, choosing RoBERTa also has the advantage to avoid the out-of-vocabulary problem.

There are many advantages to using large PLM, but applying large PLM has several challenges. Because of these challenges, applying PLM in a naive way has some limits. We present these challenges and their solutions in Section 3.4 and Section 3.5.

## 3.3 Classifier

In general, a RNN-based model is used to process sequential data, but RNN has a disadvantage in that the longer the sequence, the longer the operation time. And also, recently, it is not difficult to find the case of using a CNN-based model to process sequential data. For example, WaveNet [14] handles audio data using CNN. Since we also observed that the CNN-based model performed better than the RNN-based model in our task, in this work, the CNN-based model is used as a classifier.

We extract features from the text embeddings using four parallel convolution layers of kernel size $k \in \{3, 4, 5, 6\}$. Each convolution layer generates $H_c$ feature maps, and max-pooling layer subsamples the feature map into a $H_c$ dimension feature vector. In this work, $H_c$ is set to 256. In the BERT-based model, the representation of [CLS], which is a special token added in front of every input text, is considered to contain the embedding for the entire text. For residual connection, we concatenate the representation of [CLS] in the text embeddings and the feature vectors. The concatenated vector is used as an input to fully connected layers, and the fully connected layers output a score for each developer as a result.

## 3.4 Compressing Text Embedding Module

The large PLM requires considerable computational cost because of its huge size of the PLM. Since this large PLM is very heavy and slow, practical applications of the large PLM are difficult. To overcome this problem, the size of the PLM should be reduced.

Using small PLM can be one solution to this problem. As mentioned before, ALBERT is a reduced size model of BERT, so applying ALBERT reduces the computational cost and consequently reduces training and inference time. However, the comparison in Section 4.4 shows that ALBERT performs significantly worse than RoBERTa in our task. Thus, ALBERT, despite its lightness, is not suitable for our task.

*3.4.1 Knowledge Distillation.* Addressing this problem requires not only reducing the size of the PLM, but also maintaining the performance of the PLM. One solution to this problem can be found in *knowledge distillation* (KD), a concept first proposed by Hinton et al. [5]. KD is a sort of network compression technique that transfers knowledge from a large model into a small model. In KD, the trained large model is called the teacher and the small model to which knowledge is transmitted is called the student. KD trains the student model to mimic the teacher model through distillation loss. In our task, for a given input, the student model is trained to generate a representation similar to that generated by the teacher model. Thus, the distillation loss $L_{KD}$ in our task is formulated as follows:

$$L_{KD} = \sum_{x \in D_{KD}} MSE\left(F^t(x), F^s(x)\right), \tag{2}$$

where $MSE$ is mean squared error loss, $F^t$ is the teacher model, $F^s$ is the student model, and $D_{KD}$ is the dataset for training the student.

Note that our distillation loss $L_{KD}$ does not use a label. Bug reports in the bug tracking system are divided into open bugs and closed bugs. Open bugs are bugs that have not yet been assigned to a developer, and closed bugs are bugs that have already been fixed by an assigned developer. In other words, open bugs are unlabeled bugs and closed bugs are labeled bugs. While open bugs are not used in many bug triage studies, DBRNN-A [11] exploits open bugs for learning Word2vec. Inspired by DBRNN-A, we use open bugs as datasets for training the student model. This allows the student model to learn from richer data.

We use the RoBERTa architecture with reduced parameters as the student model. Specifically, we set the number of layers of the student model to 3, and all other parameters are set the same as Roberta-large. In this case, since Roberta-large has 24 layers, the size of the student model is 8 times smaller than that of Roberta-large.

*3.4.2 Patient Knowledge Distillation.* In general, the student model performs worse than the teacher model. This is because some information is lost when transferring the knowledge of the teacher to the student. Therefore, it is important for the student to reduce this information loss in order to preserve the performance of the teacher. Patient knowledge distillation (PKD) [18] offers a solution to this problem. In PKD, the student model learns not only to imitate the final outputs of the teacher but also to follow the thinking process of the teacher. To achieve this, a new loss has been introduced. Let $l_t$ be the number of layers in the teacher model and $l_s$ be the number of layers in the student model, where $l_t > l_s$. Then the new loss $L_{PKD}$ [18] is formulated as follows:

$$L_{PKD} = \sum_{x \in D_{KD}} \sum_{i=0}^{l_s-1} \left\| \frac{F^t_{l_t - s \cdot i}(x)}{\|F^t_{l_t - s \cdot i}(x)\|_2} - \frac{F^s_{l_s - i}(x)}{\|F^s_{l_s - i}(x)\|_2} \right\|_2^2, \tag{3}$$

where $s = \lfloor l_t / l_s \rfloor$ and $F_i(x)$ is the output of the $i$-th layer of the model $F$. The above loss is specifically a loss according to the PKD-Skip strategy. When the student model is trained by $L_{PKD}$, the intermediate layers of the student model are trained to imitate the representation of the corresponding intermediate layer of the teacher model, respectively. This makes the embedding process of the student model more similar to that of the teacher model. The experimental results detailed in Section 4.6.1 show that training the student model using $L_{PKD}$ has a lower performance penalty than using $L_{KD}$. Therefore, we adopt $L_{PKD}$ to train the student model.

## 3.5 Preventing Catastrophic Forgetting

Applying the PLM implies exploiting transfer learning. *Catastrophic forgetting* is one of the main challenges of transfer learning. Catastrophic forgetting, which was first introduced by McCloskey and Cohen in 1989 [12], is a phenomenon in which when a neural

network learns new information, it forgets previously learned information. During fine-tuning for bug triage tasks, the PLM is trained to generate representations from which the train data can be classified well. This implies that the PLM implicitly learns task-specific knowledge contained in the train data during fine-tuning. However, as the fine-tuning progresses, the PLM forgets how to generate representations for general vocabulary which is not included in the train data. In other words, catastrophic forgetting causes the PLM to lose previously learned general language knowledge after fine-tuning.

*3.5.1 Naive Solutions.* Catastrophic forgetting leads the model to overfit. Consequently, the model cannot achieve its maximum potential performance without resolving catastrophic forgetting. One naive solution to avoid catastrophic forgetting is early stopping. If training is stopped before the model's performance deteriorates due to overfitting, the model can achieve somewhat high performance. However, it is hard to decide when to stop the training. Any performance degradation in the previous several epochs during training does not mean that the maximum performance has been reached or that overfitting has occurred. Because of this, for example, early stopping when performance degrades does not guarantee maximum performance. Thus, using this early stopping scheme to avoid catastrophic forgetting relies heavily on heuristics. Also, overfitting can occur before the model's potential maximum performance is reached. Thus, while early stopping mitigates the effect of catastrophic forgetting, it does not completely solve the problem.

Freezing the PLM is another naive solution. In this solution, only the classifier is trained during the fine-tuning. Also, because the PLM is not updated, it does not suffer from catastrophic forgetting and preserves its general language knowledge. Even though the PLM is not fine-tuned, it can generate generally good representations for text. Hence, a model trained in this way has proper performance without overfitting during training. However, despite avoiding catastrophic forgetting by freezing the PLM, the PLM cannot learn task knowledge from train data. Thus, the frozen PLM cannot generate task-specific representations, and consequently, underfitting occurs. Therefore, freezing the PLM guarantees the model's general language knowledge, but the maximum potential performance of the model cannot be achieved.

*3.5.2 Knowledge Preservation Fine-tuning.* With naive solutions, the model suffers from overfitting or underfitting. When the PLM loses its general language knowledge, the bug triage model overfits. In contrast, when the PLM does not learn task knowledge, the model underfits. Thus, the main goal of fine-tuning to address this problem is to have the PLM learn task knowledge while preserving general language knowledge. General language knowledge and task knowledge seem like a trade-off relationship, but actually, they are not. Intuitively, earlier layers of the PLM are strongly associated with the input text, and later layers of the PLM are strongly associated with the output representation. Thus, we can consider the earlier layers are more important to understand the text and the later layers are more important to generate the representation. In other words, general language knowledge is mainly stored in the earlier layers and task knowledge is mainly stored in the later layers. So, we use this intuition to achieve our goal.

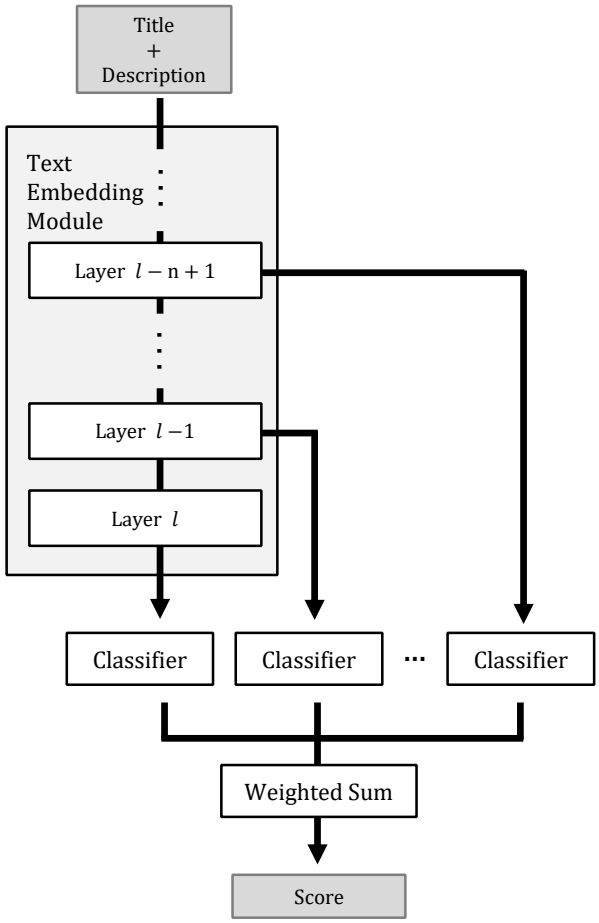



**Figure 2: The architecture of modified bug triage framework for combined thinking.**

To overcome catastrophic forgetting, we propose *knowledge preservation fine-tuning*. It is a fine-tuning that freezes the earlier layers of the PLM and only updates the later layers. The earlier layers are frozen to preserve general language knowledge and later layers are updated to learn task knowledge. So, the PLM trained with knowledge preservation fine-tuning has both general language knowledge and task knowledge. We show the effectiveness of knowledge preservation fine-tuning in Section 4.6.2 with experimental results.

## 3.6 Handling Overthinking

Overthinking of networks is another challenge, which is not related to applying the PLM. When representation of an input at the network's earlier layer is sufficient to make a correct classification, the network "overthinks" on the input. [7] Intuitively, when a person thinks too complexly, it can lead to an incorrect conclusion for a simple problem. Similarly, in the case of the neural network, when the earlier layer generates sufficiently informative representations, some noise can be added to the representations through the later layers. Simply reducing the number of layers of the model cannot

be a solution for overthinking because hard problems still require complex thinking. Thus, although the model shows good performance for complex problems, overthinking may lead to performing badly on simple problems.

*3.6.1 Combined Thinking.* Since hard problems require complex thinking and uncomplicated thinking helps to solve simple problems, one of the intuitive solutions is to combine deep and shallow thinking. So, we exploit the output of the early layers to overcome overthinking. To do this, we select later $k$ layers of the text embedding module including the last layer, and attached the classifier to each selected layer. Then the final output of the model is the weighted sum of each classifier's output. The modified model $F$ is formulated as follows:

$$F(x) = \sum_{i=0}^{k-1} |w_i| \cdot C_i\left(T_{l-i}(x)\right), \tag{4}$$

where $x$ is input text, $T_i(x)$ is the output of the $i$-th layer of the text embedding module, $C_i$ is the classifier attached to the $i$-th layer, and $w$ is weight. Figure 2 shows our combined thinking architecture. In this work, we set $k$ to 3. That is all layers of the text embedding module are used.

To train the model, first, we used the cross-entropy loss as a loss function without any special changes. The loss function $L_{CE}$ is formulated as follows:

$$L_{CE} = \sum_{(x,y)\in D} CE\left(F(x), y\right), \tag{5}$$

where $CE$ is cross-entropy loss. We observed that when using this loss function, the model is trained to depend too much on the output of a particular layer. Because of this tendency, the classifiers of the remaining layers are not properly trained. So each result of the remaining layers' classifier tends to be far from recommending an appropriate developer.

To combine deep and shallow thinking, each classifier needs to be properly trained. We introduce *combined thinking loss* to tackle this problem. Combined thinking loss $L_{CT}$ is formulated as follows:

$$L_{CT} = L_{CE} + \sum_{(x,y)\in D} \sum_{i=0}^{k-1} CE\left(F_i(x), y\right), \tag{6}$$

where $F_i(x) = C_i\left(T_{l-i}(x)\right)$ is the output of the $i$-th classifier. In combined thinking loss $L_{CT}$, a new term is added, and it forces each classifier to get the classification ability properly. So, using combined thinking loss, the model does not suffer from training failure problems for each classifier, and deep and shallow thinking are properly combined. We demonstrate the effectiveness of combined thinking loss in Section 4.6.3.

# 4 EXPERIMENTS

## 4.1 Research Questions

In this paper, we aim to answer the following research questions:

- ***RQ1****: What is the most effective PLM for the bug triage task?*
- ***RQ2****: How effective is LBT-P for the bug triage task?* LBT-P aims for performance and lightness, so this RQ can be divided into two RQs as follows:
  - **RQ2.1**: How accurate is LBT-P at assigning developers to the given bug?
  - **RQ2.2**: How faster and lighter is LBT-P than a model which naively applies the PLM?
- ***RQ3****: How effective are the components of LBT-P in addressing the challenges?* To answer this RQ, we provide an ablation study that presents an experimental basis for our proposed methods. This RQ is specified as follows:
  - **RQ3.1**: How effective is patient knowledge distillation at reducing information loss?
  - **RQ3.2**: How effective is knowledge preservation fine-tuning at preventing the catastrophic forgetting?
  - **RQ3.3**: How effective are combined thinking architecture and loss at handling the overthinking problem?

## 4.2 Dataset

For a fair comparison, we use 3 public real-world datasets used in the baseline method [11]: Google Chromium, Mozilla Core, and Mozilla Firefox. In preprocessing, we filtered out bug reports of 15 words or less whose text was too short to contain contextual information. Google Chromium dataset has 163,695 open bugs and 118,607 closed bugs. The number of developers in Google Chromium dataset is 2,529. Mozilla Core and Mozilla Firefox datasets have 186,173 and 138,093 open bugs, respectively, and they have 128,104 and 24,158 closed bugs, respectively. The number of developers in Mozilla Core and Mozilla Firefox datasets are 2,548 and 1,314, respectively.

As mentioned earlier, the open bugs, which have no label, are used to pre-train the student model used as a text embedding module. Closed bugs are used to fine-tune the bug triage model. 90% of closed bugs are used for training and 10% of closed bugs are used for the test.

Furthermore, we also validate our framework with real industry private data. The private dataset has 84,267 closed bugs. The number of developers in the dataset is 2,910. In the case of the private data, there are no open bugs so the student model is pre-trained by train data of closed bugs. Table 1 shows a summary of data statistics.

## 4.3 Experimental Setup

To evaluate our proposed framework, LBT-P, we use top-$k$ accuracy (acc@$k$). Specifically, we report results where $k \in \{1, 2, 3, 4, 5, 10, 20\}$. The Adam optimizer [8] is used to train LBT-P, and the learning rate is set to 0.00001. We train LBT-P for 100 epochs. In the case of the student model used as a text embedding module, as mentioned in Section 3.4, the number of layers of the student model is set to 3, and all other parameters are set the same as Roberta-large. The student model is trained for 2 epochs before fine-tuning.

In the experiment, to present the validity of proposed techniques and the superiority of LBT-P, we compare the performance of the following methods:

(1) Baseline methods
   - **TF-IDF**: It is the baseline for traditional text feature. Given bug report, it computes the similarity to the bugs in the train dataset based on the TF-IDF, selects the reports with the top $k$ similarity, and finally recommends the developers of the selected reports.

**Table 1: Data statistics (after preprocessing).**

| Dataset | Train size (KD) | Train size | Test size | Total developers |
|---|---|---|---|---|
| Google Chromium | 163,695 | 106,778 | 11,829 | 2,529 |
| Mozilla Core | 186,173 | 115,393 | 12,711 | 2,548 |
| Mozilla Firefox | 138,093 | 21,792 | 2,366 | 1,314 |
| Private | 75,883 | 75,883 | 8,384 | 2,910 |

**Table 2: Top-$k$ accuracy obtained on Google Chrome dataset.**

| | Model | Acc@1 | Acc@2 | Acc@3 | Acc@4 | Acc@5 | Acc@10 | Acc@20 |
|---|---|---|---|---|---|---|---|---|
| Basic PLM | BERT + CNN | 27.179 | 37.247 | 43.495 | 47.663 | 51.162 | 61.087 | 70.251 |
| | RoBERTa + CNN | 27.754 | 37.763 | 43.799 | 48.229 | 51.644 | 62.093 | 71.173 |
| | ALBERT + CNN | 4.6242 | 6.9828 | 8.5045 | 9.7810 | 10.787 | 14.870 | 20.678 |
| | ELECTRA + CNN | 16.392 | 22.901 | 26.993 | 30.239 | 33.097 | 42.083 | 51.644 |
| Baseline | TF-IDF | 20.044 | 27.796 | 32.750 | 36.402 | 39.150 | 47.975 | 56.176 |
| | DBRNN-A | 11.269 | 16.526 | 20.091 | 22.947 | 25.412 | 33.652 | 42.519 |
| | RoBERTa-large + CNN | 28.887 | 39.445 | 45.752 | 50.402 | 53.994 | 63.716 | 72.652 |
| KD | Student(KD) + CNN | 24.998 | 34.128 | 40.587 | 44.797 | 48.001 | 57.909 | 67.605 |
| | Student(PKD) + CNN | 27.779 | 37.788 | 43.901 | 48.449 | 51.923 | 61.907 | 70.843 |
| Proposed | LBT-P* | 30.070 | 40.350 | 46.741 | 51.315 | 54.417 | 64.350 | 72.686 |
| | LBT-P | **32.395** | **43.376** | **49.894** | **54.400** | **57.773** | **67.132** | **75.230** |

**Table 3: Top-$k$ accuracy obtained on Mozilla Core dataset.**

| | Model | Acc@1 | Acc@2 | Acc@3 | Acc@4 | Acc@5 | Acc@10 | Acc@20 |
|---|---|---|---|---|---|---|---|---|
| Baseline | TF-IDF | 23.452 | 32.114 | 37.259 | 41.161 | 44.135 | 54.394 | 64.354 |
| | DBRNN-A | 14.401 | 21.318 | 26.334 | 29.965 | 33.313 | 42.884 | 53.520 |
| | RoBERTa-large + CNN | 35.804 | 46.700 | 53.363 | 57.525 | 61.018 | 70.931 | 79.608 |
| Proposed | LBT-P* | 37.235 | 48.556 | 54.795 | 59.091 | 62.340 | 71.780 | 79.931 |
| | LBT-P | **40.492** | **52.010** | **58.571** | **62.930** | **66.368** | **75.439** | **82.669** |

**Table 4: Top-$k$ accuracy obtained on Mozilla Firefox dataset.**

| | Model | Acc@1 | Acc@2 | Acc@3 | Acc@4 | Acc@5 | Acc@10 | Acc@20 |
|---|---|---|---|---|---|---|---|---|
| Baseline | TF-IDF | 22.739 | 33.178 | 38.673 | 43.576 | 47.506 | 58.369 | 67.878 |
| | DBRNN-A | 8.4038 | 14.178 | 18.357 | 22.535 | 25.728 | 36.432 | 49.765 |
| | RoBERTa-large + CNN | 29.670 | 40.828 | 48.267 | 53.043 | 56.932 | 66.822 | 76.078 |
| Proposed | LBT-P* | 33.981 | 45.097 | 50.676 | 55.156 | 58.326 | 68.047 | 76.458 |
| | LBT-P | **34.108** | **46.830** | **53.762** | **58.538** | **61.581** | **70.287** | **78.191** |

- **DBRNN-A** [11]: It embeds text in a bug report using Word2vec, and assigns the bug report to developers using bidirectional RNN with attention mechanism.
- **RoBERTa-large** [10] **+ CNN** : RoBERTa-large is naively applied to it as a text embedding module, and it adopts CNN-based model as a classifier. Because BERT outperforms ELMo [15], we skip to report the results for ELMo-CNN [25].

(2) Basic PLM methods to which a PLM is naively applied

- **BERT** [4] **+ CNN**: It adopts BERT as a text embedding module. BERT is most representative Transformer-based PLM.
- **RoBERTa** [10] **+ CNN**: It adopts RoBERTa as a text embedding module. RoBERTa improves BERT by changing the pre-training approach and increasing training data.

**Table 5: Top-$k$ accuracy obtained on the private dataset.**

| Model | | Acc@1 | Acc@2 | Acc@3 | Acc@4 | Acc@5 | Acc@10 | Acc@20 |
|---|---|---|---|---|---|---|---|---|
| Baseline | TF-IDF | 25.584 | 33.922 | 38.943 | 42.510 | 45.277 | 54.413 | 62.870 |
| | DBRNN-A | 9.5815 | 15.493 | 19.715 | 22.916 | 26.070 | 35.544 | 46.301 |
| | RoBERTa-large + CNN | 32.896 | 43.321 | 49.952 | 54.342 | 57.574 | 67.402 | 76.610 |
| Proposed | LBT-P* | 37.989 | 48.712 | 55.069 | 58.862 | 61.892 | 70.873 | 78.817 |
| | LBT-P | **38.645** | **49.332** | **55.582** | **59.673** | **62.739** | **71.386** | **79.330** |

- **ALBERT** [9] **+ CNN** : It adopts ALBERT as a text embedding module. ALBERT reduces the size of BERT by factorization of the embedding parameters and parameter sharing.
- **ELECTRA** [2] **+ CNN** : It adopts ELECTRA as a text embedding module. ELECTRA improves BERT by introducing RTD.

(3) Methods applying KD
- **Student(KD) + CNN**: It adopts a student model of RoBERTa-large as a text embedding module. The student model is trained by naive KD loss $L_{KD}$.
- **Student(PKD) + CNN** : It adopts a student model, which is trained by patient KD [18] loss $L_{PKD}$, as a text embedding module.

(4) Proposed methods
- **LBT-P** : It is our proposed method. It adopts student(PKD) as a text embedding module and CNN-based model as a classifier. During training, it applies knowledge preservation fine-tuning to prevent catastrophic forgetting problem. It avoids the overthinking problem by using combined thinking.
- **LBT-P*** : It is LBT-P without combined thinking. It only aims to solve challenges related to applying PLM.

## 4.4 Comparison Between Basic PLM Methods (RQ1)

To select the PLM to use for the bug triage task, we compared the performance of basic PLM methods. In the comparison, BERT, RoBERTa, ALBERT, and ELECTRA are used. These PLMs were applied naively for the bug triage task. Since updating all layers in the PLM during fine-tuning suffered out of GPU memory or not a good performance, the result of freezing all layers in the PLM is reported in Table 2. Basically, BERT has higher performance than the baseline methods TF-IDF and DBRNN-A. ALBERT and ELECTRA perform worse than BERT, while RoBERTa outperforms BERT. Based on these results, we set RoBERTa-large as the base model for the text embedding module.

## 4.5 Comparison With The Baseline Methods (RQ2)

*4.5.1 Performance (RQ2.1).* In this section, we compare baseline methods and LBT-P. Table 2 shows the performance of each method for Google Chrome dataset. Table 3, 4, and 5 show the performance for Mozilla Core dataset, Mozilla Firefox dataset, and the private dataset respectively.

In terms of performance, TF-IDF, the traditional method, seems to be superior to DBRNN-A, the existing deep learning method, but as the train dataset increases, the inference time of TF-IDF increases. If the training data is small, this property of TF-IDF is not a big problem. However, in the real world, the number of bug reports that can be used as training data increases over time. Even if a representative bug report is selected to limit the size of the training data for TF-IDF, additional instance selection is required for this. Thus, in practice, TF-IDF is not suitable for real-time service, which requires fast inference time. Furthermore, the performance of TF-IDF is inferior to RoBERTa-large + CNN to which the PLM is naively applied. This shows that the deep learning method applying the latest NLP technique is more effective for the bug triage task than the traditional text feature method.

For all datasets, LBT-P outperforms other methods. One remarkable observation is that LBT-P*, which only solves problems related to PLM, also outperforms RoBERTa-large + CNN, which outperforms other baseline methods. This shows that the maximum potential performance of the bug triage model cannot be derived by just applying the PLM naively.

*4.5.2 Model size and training time (RQ2.2).* Table 6 shows the number of parameters and training time of RoBERTa-large + CNN and LBT-P. Since LBT-P uses the small student model which has fewer layers than RoBERTa-large, the number of parameters in LBT-P is reduced to 33% of the number of parameters in RoBERTa-large + CNN. Also, in the case of training time, LBT-P is about 2.73 times faster than RoBERTa-large, despite the additional training time of the student model required. Therefore, the proposed model not only outperforms RoBERTa-large, but is also faster and smaller.

## 4.6 Ablation Study (RQ3)

*4.6.1 The Effectiveness of Patient Knowledge Distillation (RQ3.1).* In this section, we present the effectiveness of PKD. To do this, we compare the student model trained by naive KD loss $L_{KD}$ and the student model trained by PKD loss $L_{PKD}$. As aforementioned, we use the open bugs to pre-train the student model. Thus, train size (KD) in Table 1 corresponds to the size of the open bugs. The table shows that using the open bugs can train students with richer data. The comparison result is reported in Table 2. Since, information loss occurs when a teacher's knowledge is transferred to a student, both students have lower performance than the teacher, RoBERTa-large. The performance degradation of the student trained by $L_{PKD}$ is lower than that of the student trained by $L_{KD}$, and this implies that using PKD, which mimics the thinking process of the teacher, reduces information loss.

**Table 6: The number of parameters and training time of RoBERTa-large-based methods on Google Chrome dataset.**

| Model | Total parameters | Training time | | | Speedup |
|---|---|---|---|---|---|
| | | Student | Fine-tuning | Total | |
| RoBERTa-large + CNN | 364,769,761 | - | 277,273.7 s | 277,273.7 s | 1.00 |
| LBT-P* | 100,249,057 | 11,420.69 s | 70,144.60 s | 81,565.29 s | 3.40× |
| LBT-P | 119,069,094 | 11,393.61 s | 90,313.64 s | 101,707.4 s | 2.73× |

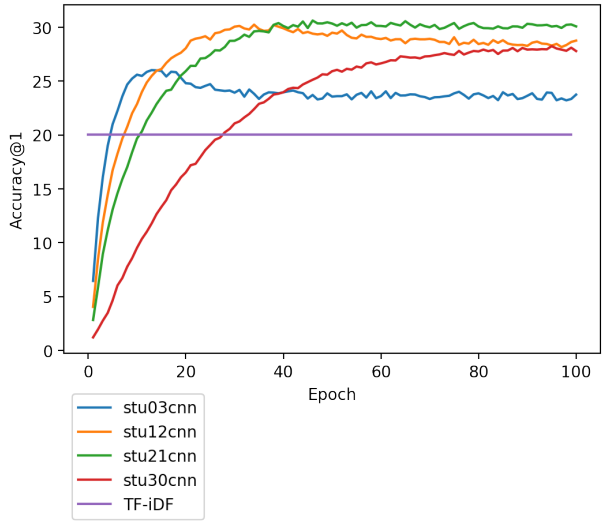



**Figure 3: Test accuracy per epoch according to the number of frozen layers in the text embedding module. The results are obtained on Google Chrome dataset.**

*4.6.2 The Effectiveness of Knowledge Preservation Fine-tuning (RQ3.2).* In this section, we present the effectiveness of knowledge preservation fine-tuning. We compared the test rank 1 accuracy per epoch by varying the number of frozen layers in the text embedding module. Figure 3 shows the result of this comparison. In this experiment, we use Student(PKD) + CNN as the base model. In the figure, for example, stu21cnn indicates that the first 2 layers of the text embedding module are frozen and the remaining 1 layer is updated during fine-tuning. The full layer freezing (stu30cnn) is exactly the same as Student(PKD) + CNN.

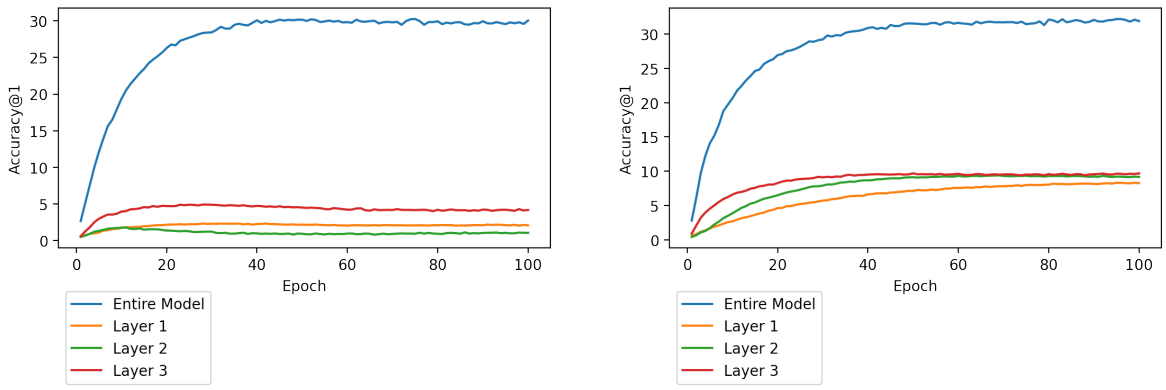
When all layers are updated without any freezing, performance degradation due to overfitting is observed at the fine-tuning midpoint. When all layers are frozen, overfitting is not observed but the model does not achieve the best performance. Note that the highest accuracy of the no layer freezing (stu03cnn) is lower than the highest accuracy of the full layer freezing (stu30cnn). This shows that overfitting occurs before reaching the maximum potential performance of the model in full layer update. So, this implies that early stopping cannot completely solve catastrophic forgetting.

In the case of stu12cnn, it updates the earlier layer, so, similar to full layer update, overfitting occurs. Since stu12cnn does not modify the first layer which contains the most essential knowledge about language, it outperforms stu30cnn. The highest performance is achieved when only the last layer is updated and the remaining layers in the text embedding module are frozen (stu21cnn). Also, in this case, the model did not suffer performance degradation during fine-tuning.

*4.6.3 The Effectiveness of Combined Thinking Loss (RQ3.3).* To handle the overthinking problem, we modified our framework to do combined thinking. We also introduce combined thinking loss to fine-tune the modified framework. In this section, we present the effectiveness of the combined thinking loss. Figure 4 shows the accuracy per epoch according to the loss. In the case of naive cross-entropy loss $L_{CE}$, only the final layer is trained properly, while the rest of the layers are not properly trained. That is the model relies heavily on layer 3. On the other hand, in the case of combined thinking loss $L_{CT}$, all layers are properly trained. Moreover, all layers with $L_{CT}$ achieve higher performance than the last layer with $L_{CE}$. Figure 5 shows performance comparison between $L_{CE}$ and $L_{CT}$. No CT in the figure stands for the model which is not modified for combined thinking and this is identical to LBT-P*. In this comparison, the performance of the models with $L_{CE}$ is not significantly different from that of LBT-P*. This is because, as shown in Figure 4, the models with $L_{CE}$ heavily depend on the last layer, so there is no big difference from LBT-P*, which uses only the last layer. The model with $L_{CT}$, in which all layers are properly trained, shows higher performance than LBT-P*. This shows that both shallow and deep thinking can be used with the proposed loss $L_{CT}$.

## 5 RELATED WORK

Rich semantic information in the bug report can be used in various ways. Therefore, bug reports are used not only in the developer assignment task but also in other various tasks. In this section, we introduce several studies that exploit the bug report. Chen et al. [1] study generating the title from the description of the bug report. They regard title generation as a one-sentence summarization task. To do this, they build a Seq2Seq model to summarize the description. Yanqi el al. [23] aim to deal with bug component assignment task. In particular, they focus on solving the bug tossing phenomenon. To solve the problem, they construct a bug tossing knowledge graph and exploit derived features from the graph. They adopt a learning-to-rank model to assign the bug component. This bug component assignment task is similar to our task. The main difference between these tasks is the number of classes. Their study covers 186 components, but in the case of the developer assignment task, the number of developers exceeds a thousand. Therefore, although their study is effective in a small class size task, it is hard




**(a) Cross-entropy loss $L_{CE}$.** **(b) Combined thinking loss $L_{CT}$.**

**Figure 4: Test accuracy per epoch of each layer of combined thinking architecture when trained by cross-entropy loss $L_{CE}$ and combined thinking loss $L_{CT}$. The results are obtained on Google Chrome dataset.**

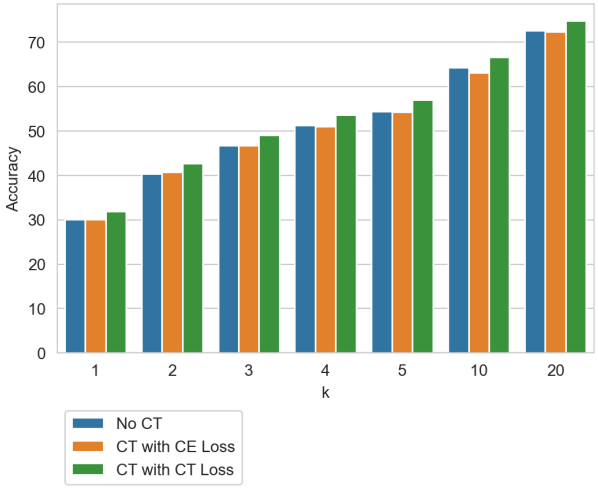



**Figure 5: Comparison of combined thinking model performance according to loss. The results are obtained on Google Chrome dataset.**

to extend their study to our task because the size of the knowledge graph increases exponentially. Yang et al. [17] introduce a tool that automatically analyzes bug reports. The tool uses the sentence in the bug report as input, and it identifies whether the sentence corresponds to three types: the observed behavior, the expected behavior, and the steps to reproduce the bug. To do this, they extract *n*-grams and part-of-speech tags from the sentence and use SVM as a classifier. Ye et al. [24] and Wang et al. [20] studied information retrieval-based bug localization technique using bug reports.

## 6 CONCLUSION & DISCUSSION

Assigning an appropriate developer to the bug, which is called bug triage, is important to maintaining a large project. In this work, we propose a novel light bug triage framework with a pre-trained language model called LBT-P that solves the challenges of applying large PLM and the overthinking problem. Applying large PLM to bug triage tasks has problems with the size of the large PLM and problem of catastrophic forgetting. We compress the size of the PLM using patient knowledge distillation [18] and prevent catastrophic forgetting by introducing knowledge preservation fine-tuning. Furthermore, we propose combined thinking architecture and combined thinking loss to handle the overthinking problem. The experimental result on three real-world datasets [11] and the private industry dataset shows the effectiveness of our proposed framework.

Because the framework we propose aims to solve the task of classification, it can be extended easily to other tasks on bugs in addition to developer assignment task such as department assignment and bug component assignment. In addition, as the PLM can process various text data, proposed knowledge preservation fine-tuning can also be applied with the PLM to various tasks that need text embeddings.

## ACKNOWLEDGMENTS

This work was supported by SAMSUNG Research, Samsung Electronics Co.,Ltd., Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2018-0-00584, (SW starlab) Development of Decision Support System Software based on Next-Generation Machine Learning), the NRF grant funded by the MSIT (South Korea, No.2020R1A2B5B03097210), and Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2019-0-01906, Artificial Intelligence Graduate School Program(POSTECH)).

## REFERENCES

[1] Songqiang Chen, Xiaoyuan Xie, Bangguo Yin, Yuanxiang Ji, Lin Chen, and Baowen Xu. 2020. Stay professional and efficient: automatically generate titles for your bug reports. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 385–397.

[2] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2020. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555* (2020).

[3] Václav Dedík and Bruno Rossi. 2016. Automated Bug Triaging in an Industrial Context. In *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 363–367. https://doi.org/10.1109/SEAA.2016.20

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[5] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

[6] Leif Jonsson, Markus Borg, David Broman, Kristian Sandahl, Sigrid Eldh, and Per Runeson. 2016. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. *Empirical Software Engineering* 21, 4 (2016), 1533–1578.

[7] Yigitcan Kaya, Sanghyun Hong, and Tudor Dumitras. 2019. Shallow-deep networks: Understanding and mitigating network overthinking. In *International Conference on Machine Learning*. PMLR, 3301–3310.

[8] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[9] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942* (2019).

[10] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[11] Senthil Mani, Anush Sankaran, and Rahul Aralikatte. 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*. 171–179.

[12] Michael McCloskey and Neal J Cohen. 1989. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*. Vol. 24. Elsevier, 109–165.

[13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[14] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. 2016. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499* (2016).

[15] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep Contextualized Word Representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. Association for Computational Linguistics, New Orleans, Louisiana, 2227–2237. https://doi.org/10.18653/v1/N18-1202

[16] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[17] Yang Song and Oscar Chaparro. 2020. Bee: a tool for structuring and analyzing bug reports. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1551–1555.

[18] Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. 2019. Patient knowledge distillation for bert model compression. *arXiv preprint arXiv:1908.09355* (2019).

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[20] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the 2015 international symposium on software testing and analysis*. 1–11.

[21] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771* (2019).

[22] Jifeng Xuan, He Jiang, Yan Hu, Zhilei Ren, Weiqin Zou, Zhongxuan Luo, and Xindong Wu. 2015. Towards Effective Bug Triage with Software Data Reduction Techniques. *IEEE Transactions on Knowledge and Data Engineering* 27, 1 (2015), 264–280. https://doi.org/10.1109/TKDE.2014.2324590

[23] Xin Peng Xin Xia Chong Wang Xiwei Xu Liming Zhu Yanqi Su, Zhenchang Xing. 2021. Reducing Bug Triaging Confusion by Learning from Mistakes with a Bug Tossing Knowledge Graph. In *36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE.

[24] Xin Ye, Razvan Bunescu, and Chang Liu. 2014. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 689–699.

[25] Syed Farhan Alam Zaidi, Faraz Malik Awan, Minsoo Lee, Honguk Woo, and Chan-Gun Lee. 2020. Applying Convolutional Neural Networks With Different Word Representation Techniques to Recommend Bug Fixers. *IEEE Access* 8 (2020), 213729–213747. https://doi.org/10.1109/ACCESS.2020.3040065