Knowledge-Based Systems 232 (2021) 107476

Contents lists available at ScienceDirect

Knowledge-Based Systems

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

How well do pre-trained contextual language representations recommend labels for GitHub issues?

Jun Wang [a], Xiaofang Zhang [a,*], Lin Chen [b]

[a] School of Computer Science and Technology, Soochow University, Suzhou, China
[b] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

ARTICLE INFO

Article history:
Received 2 March 2021
Received in revised form 18 August 2021
Accepted 6 September 2021
Available online 10 September 2021

Keywords:
Deep learning
Issue labeling
Data analysis
Language model

ABSTRACT

Motivation: Open-source organizations use issues to collect user feedback, software bugs, and feature requests in GitHub. Many issues do not have labels, which makes labeling time-consuming work for the maintainers. Recently, some researchers used deep learning to improve the performance of automated tagging for software objects. However, these researches use static pre-trained word vectors that cannot represent the semantics of the same word in different contexts. Pre-trained contextual language representations have been shown to achieve outstanding performance on lots of NLP tasks.
Description: In this paper, we study whether the pre-trained contextual language models are really better than other previous language models in the label recommendation for the GitHub labels scenario. We try to give some suggestions in fine-tuning pre-trained contextual language representation models. First, we compared four deep learning models, in which three of them use traditional pre-trained word embedding. Furthermore, we compare the performances when using different corpora for pre-training.
Results: The experimental results show that: (1) When using large training data, the performance of BERT model is better than other deep learning language models such as Bi-LSTM, CNN and RCNN. While with a small size training data, CNN performs better than BERT. (2) Further pre-training on domain-specific data can indeed improve the performance of models.
Conclusions: When recommending labels for issues in GitHub, using pre-trained contextual language representations is better if the training dataset is large enough. Moreover, we discuss the experimental results and provide some implications to improve label recommendation performance for GitHub issues.

1. Introduction

Nowadays, many open source projects are managed in web-based version control systems, such as GitHub, GitLab, and so on. Most web-based version control systems often provide issue-reporting services. Issue reports can improve the efficiency of communication between open source software developers and users. These issue tracking systems often support a tagging mechanism for organizing issues. Maintainers of open source projects make it easy for users to navigate and search by adding labels to issues. However, in some projects with a large number of issues, adding labels to issues is a time-consuming task. In addition, some new participants in these projects may be unfamiliar with the existing labels and assign the wrong labels.

Recently, deep learning has been widely used in various fields of software engineering, such as code summary, defect prediction, and so on. In previous studies, Zhou et al. [1] proposed four different deep learning models to complete tag recommendation tasks. TagCNN and TagRCNN, two of the deep learning models, outperform other traditional methods, including TagMulRec, EnTagRec, and FastTagRec. TagCNN and TagRCNN use Word2vec [2] as the word embedding of input.

However, two identical words often have different meanings depending on the context. This phenomenon is called "polysemy". Traditional word embedding is hard to handle polysemy because a word is only represented as a single vector regardless of the context. In addition, traditional word embedding is only a single layer of weights. The rest layers of the neural network still need to be trained from scratch. Moreover, traditional word embedding can represent the semantic meanings of each word, but they cannot capture some higher-level information such as long-term dependence.

In recent years, the natural language processing (NLP) community has shifted the focus of research on deep learning from traditional pre-trained word embedding to contextual language

* Corresponding author.
E-mail addresses: 20194227028@stu.suda.edu.cn (J. Wang), xfzhang@suda.edu.cn (X. Zhang), lchen@nju.edu.cn (L. Chen).

https://doi.org/10.1016/j.knosys.2021.107476
0950-7051/

representation models. The pre-trained contextual language representations, such as BERT [3], ELMo [4] and GPT2[5], achieved state-of-the-art performance in many natural language processing tasks such as natural language inference, question answering, sentiment analysis and so on.

Although the pre-trained contextual language representation model has been a great success in the NLP community, it still needs to study whether it can improve the effectiveness of label recommendation in the software engineering domain. This question motivates us to explore the pros and cons of applying pre-trained contextual embedding to the label recommendation.

In this paper, we compare different deep learning label recommendation models based on projects selected from the dataset which is collected in GitHub. For this paper, the main contributions are as follows:

- First, we collected a dataset[1] from the most popular projects in GitHub. Researchers could use this dataset to analyze or do something further on these development activities.
- Second, we present a benchmark of label recommendation for issues. This benchmark compares the performance of different deep learning models for multi-label classification in different projects by various evaluation metrics.
- Third, our findings and discussion provide insights to researchers on improving the performance of recommendation approaches based on language models.

2. Background

In this section, we introduce labels in GitHub and explain our motivation for issue labeling in GitHub.

2.1. Issue labeling in GitHub

Tagging has proven to be a lightweight and useful mechanism [6] to improve communication between software users and developers in the software development process. The open-source software projects in GitHub use the issue tracking system to track bugs, collect feedbacks, and design new features. In the GitHub issue panel, project participants only need to submit a title and a short description to create an issue. Users or developers can use some labels to mark published issues, which improves the organization of issues. Users often rely on labels to quickly search for specific topics they need. However, the labels assigned to objects are often confusing, and most labels of issues are missing.

Fig. 1 shows an issue in GitHub with the title "backup file truncated after turning on retention policy" at the top of the page, issue description in the middle, and two labels "community" and "working as intended" on the right side.

We collected and analyzed the issue history of projects of which stars are more than 1000 in GitHub. Our dataset contains 9,442,490 issues from 20,340 projects in GitHub until January 2020. From these 20,340 studied projects, 4,305,362 (45.60%) of them contain labeled issues, and the rest 5,137,128 (54.4%) issues do not have any label.

The most frequent issue labels are "bug", "enhancement", and "question". Table 1 shows the most frequent ten labels used in GitHub. In addition, Table 1 gives the number and percentage of issues containing these labels.

We count the label number of each issue. We find that 54% issues do not have any label, 24% issues only have one label, 12% issues have two labels and only 10% issues have three or more than three labels.

[1] https://bitbucket.org/jstzwj/lms4githubissue/src/master/data_view.7z.

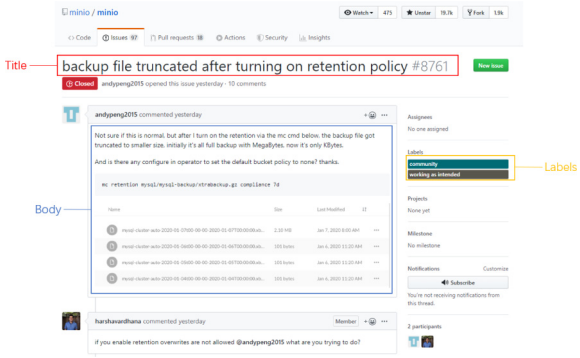

Fig. 1. GitHub issue example.

Table 1
Top ten labels used in GitHub.

Ranking	Label	Number	Percentage
1	Bug	735,400	7.79%
2	Enhancement	394,344	4.18%
3	Question	281,540	2.98%
4	Help wanted	131,627	1.39%
5	Feature	87,366	0.93%
6	Duplicate	81,653	0.86%
7	Stale	79,361	0.84%
8	Type: bug	76,930	0.81%
9	Feature request	64,927	0.69%
10	Invalid	58,434	0.62%

2.2. Problem formulation

Software information sites offer indispensable platforms for software developers to search for solutions, share experience, offer help and learn new techniques. The contents posted on these software information sites are regarded as software objects. Similar to software objects, any new GitHub issue has a title, description, comments, and labels.

We denote a set of issues as $S = \{o_1, \ldots, o_n\}$, where $o_i (1 \leq i \leq n)$ is the ith issue object. All labels of issues are denoted as $L = \{t_1, \ldots, t_m\}$. An issue object o_i consists of an issue title, issue description and a set of labels. For issue $o_i \in S$, we have $L(o_i) = \{y_i^1, y_i^2, \ldots, y_i^m\}$, where $y_i^j (1 \leq j \leq m)$ is 1 means label t_j is assigned to o_i; 0 otherwise.

A number of software engineering problems can be regarded as classification problems in machine learning. For label recommendation for GitHub issues, it can be considered as a multi-label classification problem [7]. Single-label classification models use data with a set of disjoint labels. If the number of labels equals 2, this problem is called a binary classification problem. If the number of labels is greater than 2, it is called a multi-class classification problem. The label recommendation task for issues becomes the optimization of the function $f : o_i \rightarrow L(o_i)$.

2.3. Motivation

It turns out that deep learning is effective for many software engineering tasks, such as code cloning detection, code summarization, code completion, and so on. Various deep learning language models have been proposed, including commonly used models such as convolutional neural networks (CNN), recurrent neural networks (RNN), long short-term memory networks (LSTM), gated recursive unit neural networks (GRU), and transformer. In recent years, some language models named contextual embeddings, such as ELMo, GPT, BERT, and GPT2, have been widely used to address the problem of "Polysemy". Polysemy

is known as a phenomenon where two identical words have different meanings depending on the context. Traditional word embedding is hard to handle polysemy because they use a single vector for a word to represent the meaning regardless of the context. However, contextual embeddings represent any "polysemy" depending on the words around it.

The label recommendation which this paper focuses on can be defined as a multi-label classification problem. Polysemy problem also exists in software issue text. As a sequence, we hope to evaluate the feasibility of contextual embedding in this scenario.

3. Related work

3.1. Tag recommendation

Software information sites [8–10] are community-based platforms with developing resources. The content in these sites is regarded as software objects. In the software maintenance process, tagging is an important mechanism to facilitate communication among developers.

Treude et al. [6] chose a large project with 175 developers and studied the tagging usage in the project for two years. They found that the tagging mechanism has become a significant part of many informal processes. Therefore, lightweight informal tool support may improve team-based software development practices.

Xia et al. [8] first introduced the concept of software information site and software object. Then they proposed TagCombine to recommend tags for software objects in software information sites automatically. TagCombine has three various components: multi-label ranking component, similarity-based ranking component, and tag-term-based ranking component.

Later Wang et al. [11] proposed EnTagRec, which outperforms TagCombine in terms of recall. EnTagRec tries to combine the advantages of the two opposite yet complementary lines of thought, the Bayesian inference technique and the frequentist inference technique, in the statistics community to further boost the recommendation performance.

TagMulRec proposed by Zhou et al. [10] can recommend tags and classify software objects automatically in evolving large-scale software information sites. After TagMulRec, FastTagRec [12] introduces neural networks to tag recommendations in software information sites for the first time.

Inspired by deep learning researches, TagDeepRec [13] uses attention-based Bi-LSTM to recommend new tag for software objects in software information sites.

Zhou et al. [1] proposed four deep learning based tag recommendation models, TagRNN, TagHAN, TagCNN, and TagRCNN. They use ten datasets from software information sites to compare the performance. The performance of TagCNN and TagRCNN approaches is better than traditional approaches in tag recommendation tasks.

3.2. GitHub issue classification

Adding labels for an issue in GitHub is similar to tagging a software object in a software information site. However, the difference is that tags for software objects tend to be the topics of the object, whereas the labels of issues are the purposes of issue authors. This also implies that the recommendation model needs to understand the intent of the issue more deeply.

Cabot et al. [14] analyzed more than 3 million GitHub projects and provided some insights on how to use labels in them. Results show that even if the labeling mechanism is rarely used, the use of labels can also help solve the problem.

Antoniol et al. [15] used machine learning to classify issues and showed that linguistic information is sufficient to distinguish bugs from other maintenance activities.

Herzig et al. [16] introduced six different issue categories — bug, feature request, improvement request, documentation request, refactoring request, and others. They examined more than 7000 issue reports from five open-source projects and found that issue report classifications are unreliable.

Kallis et al. [17] introduced a tool, called Ticket Tagger, which leverages machine learning strategies on issue titles and descriptions for automatically labeling GitHub issues.

In this paper, we model issue label recommendation as a multi-label classification problem and try to understand the issue description to attach appropriate labels. As far as we know, we are the first to apply the pre-training contextual language models to issue label recommendations and analyze its pros and cons.

4. Experiment setup

This section introduces the research questions, research data, baselines, experimental settings and evaluation metrics.

4.1. Research questions

We analyze and answer these research questions.

RQ1: How well can deep learning pre-trained classifiers recommend labels for GitHub issues?

Here we use four deep learning methods, CNN, Bi-LSTM, RCNN, and BERT, for multi-label classification. BERT is a pre-trained language model based on the Transformer structure with Multi-Head Attention. It utilizes two unsupervised tasks, Masked Language Model (MLM) and Next Sentence Prediction (NSP), in pre-training procedure.

However, BERT is pre-trained in the corpus for general purposes, BooksCorpus [18] and English Wikipedia, which has a different data distribution from the GitHub issue corpus. BERT are pre-trained in corpora for general purposes and may not be well adapted to the text features of the software engineering domain. A natural idea is utilizing software engineering text to further pre-train BERT and enhance its performance. This motivates our second research question.

RQ2: Does software development-specific data help pre-trained models improve recommendation compared to other pre-trained data set?

4.2. Research data

In order to obtain the dataset for research, we first crawled the issue history in projects from GitHub. The stars metric has been used by previous work [19,20] to select open-source projects. We only use the projects of which stars are more than 1000 to ensure a large enough collection of diverse and realistic projects. Then we selected the most 100 popular projects from the dataset by the number of stars. However, not all projects are software development projects. There are some tutorials, books, and other projects. The projects which do not aim to develop open-source software are removed. Then we used the left 73 projects to evaluate the performance of the different deep learning classifiers.

The following approach is used to preprocess data. Here, both text and labels need to be preprocessed. For text, we only extracted text from HTML and removed all links, codes, and images. Then all words are converted to lower case. For labels, all labels are converted to lower case first. Then, we removed issues without any label. We randomly divided 80% of issues into a training set and 20% into a validation set.

Because different projects have varying label sets and label rules, the recommendation models trained in one project cannot be used in other projects directly. To address this problem, we can use the following two strategies depending on the size of the project or the number of issues. Some projects with a large number of issues can use existing data to train models for label recommendation. In practice, some other small open source projects can use issues from other projects to predict some common labels in GitHub, such as "bug", "feature". In this paper, in order to study the performance of different models, the models are trained separately for each project.

4.3. Baselines

BiLSTM [21], CNN [22] and RCNN [23] are commonly used deep learning language models. Meanwhile, BERT [3] is one of the most famous pre-trained language models. In this paper, we trained these four different deep learning models and compared the performance in the issue label prediction dataset.

Recurrent neural network (RNN) is one of the most popular neural network architectures used to process sequence data. Compared to other neural networks, RNN is able to process variable-length data. Long short-term memory (LSTM) is a kind of special RNN, which overcomes gradient vanishing and gradient exploding problems for long sequence data. With different gates to control the data to keep or discard, LSTM can perform better in longer sequences than basic recurrent networks. Bi-directional LSTM (BiLSTM) concatenates the outputs of two LSTMs to predict each element in the sequence. The two LSTMs have different data processing directions. One processing the sequence from left to right and the other one from right to left. So any word in a sentence is predicted by preceding and following words.

Convolutional Neural Network (CNN) is a class of deep learning models that use convolutional layers. CNN is widely used in the field of image processing, such as image inpainting [24,25], image annotation [26] and image super-resolution [27]. In recent years, CNN is also used in natural language processing for fast inference speed and excellent classification performance. Kim et al. [22] first proposed a slight variant of the CNN architecture of Collobert et al. [28], named TextCNN.

Recurrent convolutional neural networks (RCNN) is proposed by Lai et al. [23]. RCNN combines a bi-directional recurrent neural network for word representation learning and a convolutional neural network for text representation learning.

BERT is a pre-trained multi-layer bidirectional Transformer encoder. The most commonly used BERT models are BERT-Base and BERT-Large. BERT-Base has 110 million parameters and BERT-Large has 340 million parameters. BERT uses the WordPiece [29] algorithm to tokenize input text. The first token in the input is always a special token ([CLS]). Token [CLS] is used for classification tasks. Token [SEP] is another special token to distinguish the previous sentence from the next sentence.

During pre-training procedure, BERT uses Masked Language Model (MLM) and Next Sentence Prediction (NSP) as an unsupervised task to learn semantic information from pre-training corpus. In the MLM procedure, BERT predicts the input tokens, which are masked at random. The training data generator chooses 15% of the token positions randomly for prediction. For tokens are chosen, 80% of them are replaced with the [MASK] token, and 10% of them are replaced with a random token, and the left 10% remain unchanged. In the NSP procedure, the training data generator of BERT chooses two sentences A and B as the inputs. 50% of the time B is the actual next sentence that follows A, and 50% of the time it is a random sentence from the corpus. BERT is trained to predict whether sentence B is the next sentence of A.

Table 2
Neural network hyperparameters settings.

	BiLSTM	CNN	RCNN	BERT
Delta	0	0	0	0
Patience	5	5	5	5
Batch size	8	16	16	24
Optimizer	SGD	SGD	SGD	Adam
Learning rate	0.01	0.1	0.1	2e−5
Truncation	600	600	600	128

4.4. Experiment settings

In our experiments, all CNNs refer to TextCNN. We use GLoVe [30] as the initial weights of word embedding for BiLSTM, CNN, and RCNN. GLoVe relies on the global co-occurrence matrix of different words in a corpus. The embedding size of GLoVe is 300.

According to the previous statistic, 90% of issues have less than three labels. Therefore, the hyperparameter k is set to 2. That means that we only select the two labels with the highest probability from the output of the model as the recommendation result.

We use the early stopping strategy during training. The early stopping method is a widely used method, and it is better than the regularization method in many cases. Optimizers are methods used to change the weights in a neural network in order to reduce losses. We used Stochastic Gradient Descent (SGD) for BiLSTM, CNN, and RCNN. For BERT, the optimizer is Adam with $\beta_1 = 0.9$ and $\beta_2 = 0.999$. In order to ensure that the model does not exceed the GPU memory limit when facing very long text, we only truncated the first 600 words in the text. For BERT, sentences are truncated if they are more than 128 tokens because BERT will take up more GPU memory. Table 2 shows some basic hyperparameter settings of the neural network models.

Different from traditional single-label classification models, an object could have multiple labels simultaneously in multi-label classification. There are many approaches to implement multi-label classification, such as one-vs-all, label embedding, label correlations [31] and so on. The most straightforward method to apply deep learning models from multi-class to multi-label classification is to extend the traditional cross-entropy loss function. Here we use binary cross-entropy loss (BCE) over sigmoid activation. The binary cross-entropy objective can be formulated as:

$$BCE = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{L}[y_{ij}log(\sigma(f_{ij})) + (1-y_{ij})log(1-\sigma(f_{ij}))] \quad (1)$$

where σ is sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$. n denotes the number of examples. L is the number of labels. y_{ij} is the jth real labels of the ith issue and f_{ij} is the jth label in prediction results of the ith issue.

4.5. Evaluation metrics

The metrics used to evaluate multi-label recommendation models can be divided into example-based and label-based metrics. Example-based metrics evaluate each example based on the list of labels in each sample, while label-based metrics evaluate each label separately.

Recall@k, Precision@k, Accuracy@k, and F1-score@k are popular example-based evaluation metrics in recommendation systems [1,32]. This means that models always recommend k items for any object. When recommending labels for issues, neural networks rank the labels by probabilities and then filter out the top-k labels. We use these metrics to evaluate deep learning label prediction models.

For an issue object o_i in a set of issues V, $Recall@k_i$ is computed by Eq. (2).

$$Recall@k_i = \begin{cases} \frac{|R_k(o_i) \cap o_i.T|}{k} & , |o_i.T| > k \\ \frac{|R_k(o_i) \cap o_i.T|}{|o_i.T|} & , |o_i.T| \leq k \end{cases} \tag{2}$$

where R_k is the recommendation model as a function that input an issue object and output k labels as recommendation results. $o_i.T$ is the truth labels of issue object o_i.

$Recall@k_i$ is the fraction of true labels in recommended list $R_k(o_i)$ among all true labels. $Recall@k$ is computed by Eq. (3). $Recall@k$ is the average of $Recall@k_i$ for all issues in the dataset.

$$Recall@k = \frac{\sum_{i=1}^{|V|} Recall@k_i}{|V|} \tag{3}$$

$Precision@k_i$ is the fraction of labels in recommended list $R_k(o_i)$ which are true labels. $Precision@k_i$ can be obtained by Eq. (4). $Precision@k$ is the average value of $Precision@k_i$ for all issues in dataset. $Precision@k$ is computed by Eq. (5).

$$Precision@k_i = \frac{|R_k(o_i) \cap o_i.T|}{k} \tag{4}$$

$$Precision@k = \frac{\sum_{i=1}^{|V|} Precision@k_i}{|V|} \tag{5}$$

If any label in the recommendation list is the true label of the issue, the issue is regarded as *correct*. $Accuracy@k$ is the percentage of issues marked as *correct* in V.

$$Accuracy@k = \frac{|\{o | o_i \in V, o_i.T \cap R_k(o_i) \neq \emptyset\}|}{|V|} \tag{6}$$

Precision and accuracy sometimes are not good metrics for classification models. A model may obtain a high level of accuracy but be useless. The top-k F1-score combines top-k precision and top-k recall. Given an issue object o_i in a set of issues V, $F1-score@k_i$ can be obtained by Eq. (7) and $F1-score@k$ is defined by Eq. (8).

$$F1-score@k_i = 2 \cdot \frac{Precision@k_i \cdot Recall@k_i}{Precision@k_i + Recall@k_i} \tag{7}$$

$$F1-score@k = \frac{\sum_{i=1}^{|V|} F1-score@k_i}{|V|} \tag{8}$$

5. Results

In this section, we present the experimental results and answer the research questions.

5.1. Parameter settings

This subsection explores the differences in performance with different pre-trained word vectors and k settings.

BiLSTM, CNN, and RCNN are deep learning models that can use different pre-trained word vectors to initialize the embedding layer to improve the model. We choose word2vec-google-news-300 [2], glove.6B.300d [30] and fasttext.wiki.en [33] as pre-training word vectors. As multi-label prediction models, we can set the hyper-parameter k to adjust the number of predictions. Because our approach aims to address the problem of multi-label prediction, we set k from 2 to 5 and compare the prediction results of these models.

Table 3 shows the experimental results under different parameter settings. The rows of the table list the results of different models under different embedding. The columns of the table list the precision, recall, and F1 under different k. The best F1-score of each row is bolded in the table.

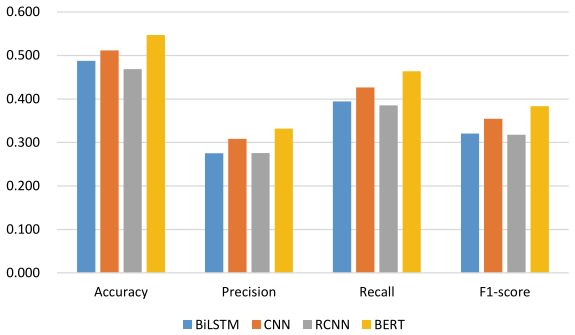

Fig. 2. The Accuracy@2, Precision@2, Recall@2, and F1-score@2 of four deep learning models.

According to the experimental results, even though recall keeps increasing as the k increases, the precision and F1-score decrease. Therefore, in multi-label prediction, F1-score achieved the highest value when k equals 2 or 3 for most projects. Because only 10% issues have three or more than three labels, we set k to 2 in the following paper. Compared to randomly initialized word embeddings, the pre-trained embeddings do improve the F1-score of the model. The improvement varies by different pre-trained embeddings. In most cases, GLoVe obtains the best performance in terms of F1-score, especially when $k = 2$. Therefore, we use GLoVe as the pre-trained word vector.

5.2. Label recommendation for issues

To answer RQ1, we trained CNN, RCNN, Bi-LSTM, and BERT on each project and compared the average performance of these four deep learning classifiers when k is set to 2. For the pre-trained BERT model, we choose the uncased version BERT-Base, which means that the BERT-Base-uncased model converts all uppercase letters to lowercase and the vocabulary has only lowercase words. Fig. 2 gives the average results in terms of Accuracy@2, Precision@2, Recall@2 and F1-score@2. The horizontal axis of the bar chart stands for the four metrics. The vertical axis indicates the value of different metrics. The blue, orange, gray, and yellow bars represent BiLSTM, CNN, RCNN, and BERT.

BERT is higher than other models in these deep learning models, and RCNN is the lowest in terms of four evaluation metrics. BERT achieved a 2.9% improvement over CNN in terms of F1-score@2. CNN improves 3.4% over Bi-LSTM and 3.6% over RCNN in terms of F1-score@2.

Wilcoxon signed-rank test [34] method is generally used to compare two algorithms, while the Friedman test [35] method is often used to compare multiple algorithms. In order to detect the significance of the difference between these four algorithms, we choose the Friedman test. We perform the Friedman test to detect the difference of these methods across multiple test attempts. The p-value of the Friedman test is 1.68e−19, which is less than 0.001. These four algorithms are different significantly.

Then we perform Nemenyi post-hoc test [36] to see which performance of these models differ significantly from each other. Fig. 3 shows the p-values of the Nemenyi post-hoc test on BiLSTM, CNN, RCNN, and BERT. Orange block indicates no significant difference between the two algorithms and blue block indicates that there are significant differences at p significance levels.

We noticed that BERT and CNN outperform BiLSTM and RCNN in terms of F1-score@2 significantly, but BERT is not significantly different from CNN, and RCNN is not significantly different from

Table 3
The experimental results under different parameter settings.

Model	Embed	k = 2			k = 3			k = 4			k = 5		
		P	R	F	P	R	F	P	R	F	P	R	F
BiLSTM	Word2Vec	0.265	0.384	**0.310**	0.232	0.478	0.306	0.204	0.540	0.289	0.185	0.606	0.277
	GLoVe	0.275	0.395	0.320	0.247	0.498	**0.324**	0.215	0.563	0.305	0.190	0.611	0.284
	fastText	0.267	0.392	**0.314**	0.234	0.482	0.310	0.208	0.551	0.296	0.188	0.613	0.282
	None	0.265	0.389	0.313	0.239	0.493	**0.317**	0.208	0.550	0.295	0.186	0.604	0.279
CNN	Word2Vec	0.294	0.404	**0.337**	0.243	0.464	0.313	0.205	0.503	0.285	0.179	0.543	0.262
	GLoVe	0.308	0.426	**0.354**	0.253	0.479	0.325	0.213	0.520	0.294	0.184	0.555	0.270
	fastText	0.306	0.428	**0.354**	0.249	0.484	0.322	0.209	0.523	0.291	0.181	0.557	0.266
	None	0.292	0.407	**0.336**	0.244	0.471	0.315	0.206	0.513	0.286	0.179	0.550	0.263
RCNN	Word2Vec	0.261	0.363	0.300	0.240	0.470	**0.310**	0.210	0.532	0.292	0.187	0.581	0.274
	GLoVe	0.276	0.385	**0.318**	0.236	0.461	0.305	0.208	0.527	0.290	0.187	0.584	0.275
	fastText	0.255	0.350	0.292	0.232	0.453	**0.300**	0.200	0.508	0.280	0.186	0.586	0.273
	None	0.263	0.364	0.302	0.238	0.463	**0.307**	0.212	0.535	0.294	0.190	0.591	0.278
BERT	None	0.332	0.464	**0.383**	0.270	0.522	0.349	0.224	0.558	0.311	0.192	0.587	0.281

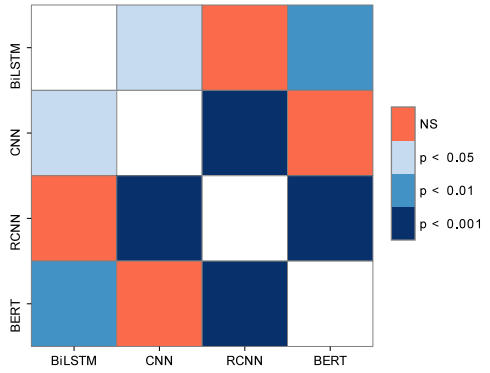

Fig. 3. The Nemenyi tests between BiLSTM, CNN, RCNN and BERT.

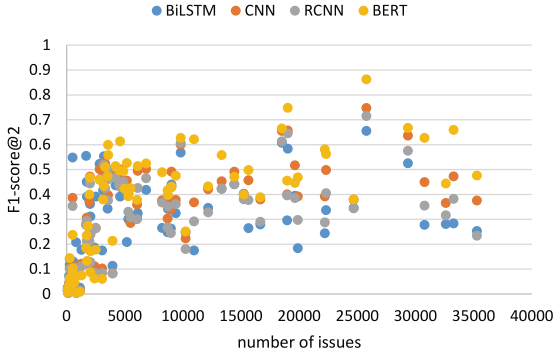

Fig. 4. Scatter plot of issues and F1-score@2 of four deep learning models.

BiLSTM. Therefore, BERT and CNN perform better than BiLSTM and RCNN on these training datasets. Though the metrics of BERT are slightly higher than CNN, the improvement is not significant.

In order to further explore the performance of BERT and CNN, we try to make a correlation analysis for the number of issues and F1-score@2. Here we try to plot the number of issues and the F1-score@2 as a scatter plot in Fig. 4. The *x*-axis of the scatter represents the number of issues in projects, and the *y*-axis is F1-score obtained by training models on projects. The blue, orange, gray and yellow markers represent BiLSTM, CNN, RCNN, and BERT.

The figure illustrates that BERT performs much better than other models when a project has more than 5000 issues. However, BERT tends to underperform CNN when the number of issues is less than 5000.

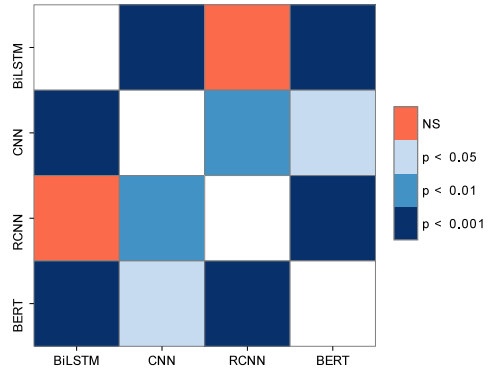

Fig. 5. The Nemenyi tests between BiLSTM, CNN, RCNN and BERT when the number of issues is more than 5000.

To confirm that the improvement of BERT over CNN is statistically significant when the number of issues is more than 5000, we selected 35 projects of which the number of issues is more than 5000 from projects mentioned above and performed the Friedman test and Nemenyi post-hoc test again. The *p*-value of the Friedman test is 1.42e−08, which is smaller than 0.001. These four models are significantly different. The Nemenyi test result is shown in Fig. 5. In these 35 projects, RCNN and BiLSTM are still not significantly different, but BERT is significantly better than CNN.

It is indicated that the performance of BERT is much better than other models when training data is enough.

> **Answer to RQ1.** BERT and CNN significantly outperforms Bi-LSTM and RCNN with F1-score in most projects. When training data is enough, BERT is better than CNN.

5.3. Software development-specific corpus

Our goal in this experiment is to answer the question of whether further pre-training on software engineering related corpus can further improve the performance of BERT. To further pre-train BERT, we totally generated two types of software engineering domain-specific corpus. One of the pre-training corpora is from GitHub. We removed projects which are chosen as the training datasets and then saved the title and description of issues as pre-training text. The other pre-training corpus is generated from StackOverflow. We saved all questions body in StackOverflow as a pre-training corpus. We pre-trained BERT in these

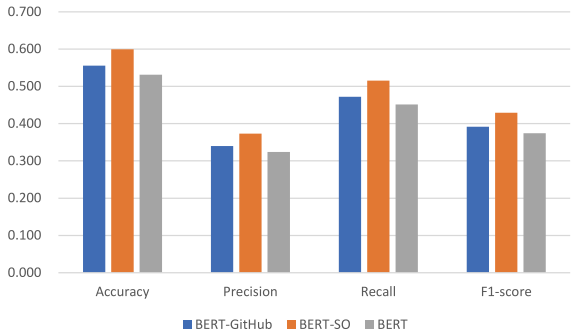

Fig. 6. The F1-score of BERT and other two BERT models pre-trained in software engineering corpus.

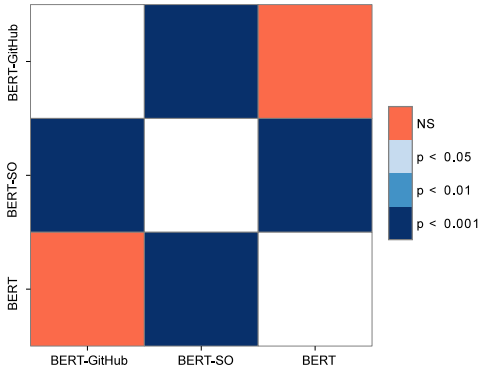

Fig. 7. The Nemenyi tests between BERT, BERT further pre-trained in GitHub and BERT further pre-trained in StackOverflow.

software engineering corpora for 100,000 steps and compared the performance.

The results are shown in Fig. 6. In the bar chart, the horizontal axis represents the four metrics, including Accuracy@2, Precision@2, Recall@2 and F1-score@2. The vertical axis indicates the value of different metrics. The gray bar is the result of BERT without further pre-training. The blue bar which is named "BERT-GitHub" is the result of the BERT further pre-trained on the GitHub corpus, and the orange bar which is named "BERT-SO" is the result of the BERT further pre-trained on StackOverflow corpus.

The absolute improvement on F1-score@2 is 1.7% for BERT pre-trained on GitHub and BERT without further pre-training. Compared to the BERT without further fine-tuning, Pre-training BERT on StackOverflow improves F1-score@2 from 37.4% to 42.9%. It can be observed that the performance of BERT further pre-trained on GitHub corpus is not as good as BERT further pre-trained On StackOverflow corpus in terms of F1-score@2.

To confirm the improvement is statistically significant, we perform the Friedman test and Nemenyi post-hoc test again. The p-value of the Friedman test is 2.09e−13. Fig. 7 shows the result of Nemenyi post-hoc test.

It can be seen that the model further pre-trained on StackOverflow is better, but the model further pre-trained on GitHub has not significantly improved.

As shown in the figure, further pre-training in software engineering corpus can improve the performance of BERT. Furthermore, BERT further pre-training in the StackOverflow dataset obtains better results.

In the further pre-training process, the step is set to 100,000. What should the step hyper-parameter be set to? In order to explore the effect of step on the result of further pre-training. We

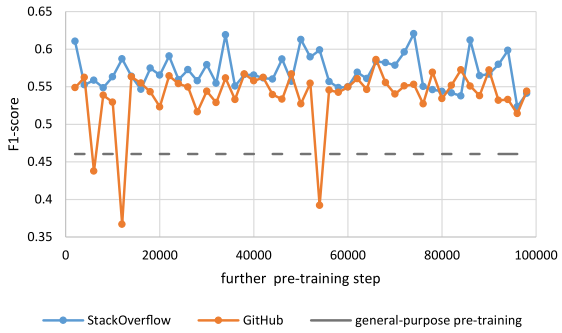

Fig. 8. The F1-score@2 of each further pre-training step on project tensorflow.

save checkpoints every 2000 steps during further pre-training. Then we fine-tuning each checkpoint and evaluate on the most popular machine learning project - "tensorflow".

Fig. 8 shows the results. The x-axis in the line chart represents the number of steps from 2000 to 98 000, and the y-axis represents the F1-score of the checkpoint trained on the tensorflow project. The blue line is the performance of the model pre-trained on StackOverflow corpus, and the orange line is the performance of the model pre-trained on GitHub corpus. The gray line stands for the ten replicates average performance of BERT-Base-uncased without further pre-training.

The F1-score of models pre-trained on StackOverflow are varying from 52.34% to 62.07%. The F1-score of models pre-trained on GitHub are varying from 36.71% to 58.62%. In most cases, further pre-trained models perform better than the average of general-purpose pre-trained models in terms of F1-score. Though saved checkpoints are not as good as the model pre-trained on general-purpose corpus at some further pre-training steps, it is possible to test these saved models on the validation dataset and select the best model in practice.

In most cases, models pre-trained on StackOverflow are better than pre-trained on GitHub. In addition, a small number of steps is enough to improve performance in further pre-training.

> **Answer to RQ2.** Further pre-training using software engineering corpus can improve the performance of BERT. Furthermore, a small number of further pre-training steps is sufficient to improve performance. More further pre-training steps may not bring additional performance improvement.

6. Discussion

This section evaluates some other pre-trained contextual language models and performs a manual analysis of the results. We also discuss the implications for researchers. Then, we present the limitations of this study.

6.1. Dataset split ratio

In the above experiments, we randomly divided 80% of issues into a training set and 20% into a test set. In this section, we study the variation of model performance with the dataset split ratio.

We split the dataset in the ratio from 9:1, 8:2, to 1:9 into a training dataset and test dataset. We train the BERT-Base model on the training set and record the performance in the test dataset. The other training parameters are the same as mentioned in Section 5.1.

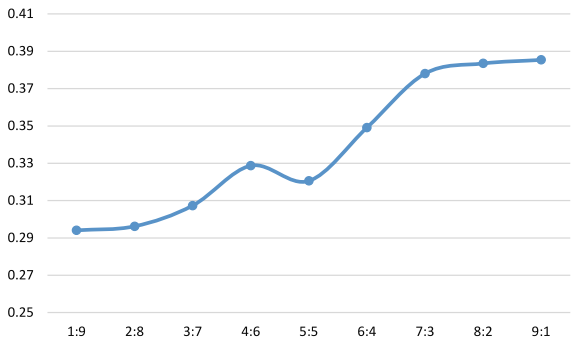

Fig. 9. The F1-score@2 of BERT-Base with different dataset split ratios.

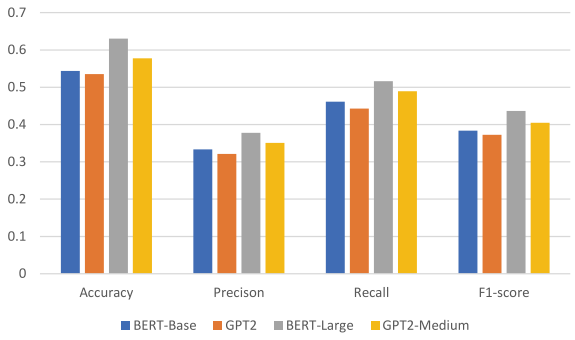

Fig. 10. The F1-score@2 of BERT and GPT2.

Fig. 9 shows the result of the experiment. The horizontal axis represents the proportion of the training dataset and the test dataset. The vertical axis is the test results of BERT-Base in terms of F1-score@2.

It can be seen that the dataset split ratio has effect on the test results of the model. With the increase of training dataset size, the F1-score@2 of the model increase from 29.4% to 38.4%. It can be observed that the more the training data, the higher the F1-score of the result.

When the split ratio is greater than 7:3, the split ratio has little effect on performance ($<1\%$). Therefore, we choose 8:2 as the split ratio according to the widely used experimental setting.

6.2. Other pre-trained models

In addition to BERT, there are some other pre-trained contextual embedding models, for example, GPT2. Here we select BERT and GPT2 to make a simple comparison. The criterion we follow in this subsection is the number of transformer block layers. BERT-Base and GPT2 have 12 layers transformer block, 768 hidden size, and 12 heads of each multi-headed self-attention. BERT-Large and GPT2-Medium have 24 layers transformer block, 1024 hidden size, and 16 heads of each multi-headed self-attention.

The results are shown in Fig. 10. The horizontal axis represents Accuracy@2, Precision@2, Recall@2, and F1-score@2. The vertical axis indicates the value of these metrics. The blue and gray bars are the results of BERT-Base and BERT-Large. The orange and yellow bars are the results of GPT2 and GPT2-Medium.

Among these models, BERT-Large has better performance than BERT-Base. GPT2-Medium is better than GPT2. Obviously, the more layers of transformer blocks, the better the model performance when models have similar structures. In addition, BERT is better than GPT2 when they have the same number of transformer blocks. We can observe that BERT is more suitable than GPT2 for the classification task.

6.3. Analysis of the results

In this section, we manually analyzed the predictions of these language models in the validation dataset. We discuss our findings, the challenges we faced, and where we see the potential for improvement.

To obtain examples for analysis, we got all predictions of models in the validation dataset of project "chart.js". "chart.js" is one of the most popular charting libraries. The issues data include the title and description of issues. The predictions of models are series of labels with corresponding probabilities. Here we use top-2 labels as the prediction of models, but in the tables below we show five prediction results for better understanding the behavior of the model.

The Table 4 shows two examples where the BERT model predicts correctly. The most frequent labels in the "chart.js" project are "type: bug", "type: support" and "type: enhancement" and their F1-score of BERT are 70.87%, 61.00% and 47.12%. We found that the model tends to predict more accurately when labels that appear more frequently in training dataset, such as "bug", "support", etc. For labels such as "bug", "feature", "support", "document", etc., text information is sufficient for these models to distinguish between the different types of issues. By contrast, some special labels such as "stale", "duplicate" require more issue information. In this case, more information about the state of issues are needed to improve accuracy. For example, we can calculate the average time of all issues from issue creation to the time issues marked as "stale". Then we consider issues that exceed the average time as stale issues.

In the first example, the model successfully predicts the "support" label. According to our experimental results of BERT in the validation dataset, if the issue description contains "help", "try", "how" and "want", the percentage of issues predicted "support" are 62.69%, 58.72%, 56.86% and 66.34%. We found that the model seems to be more inclined to output the "support" label when there are words such as "help", "how", "try" and "want" in the input. At the same time, we found some high frequency words in the issues labeled as "bug". If there is a "bug" in the text, then 85.57% of the issues are labeled with "bug". If "error", "issue" and "problem" appeared in the text, the percentage of issues predicted "bug" is 72.72%, 67.89% and 69.12% respectively.

In the second example, the real labels for the issue are "help wanted" and "type:enhancement". The model successfully predicts these two labels, although the order is not the same as the ground truth.

Next, we show examples with incorrect predictions in Table 5 and explain why neural network predictions are wrong. In the first case, the model failed to predict the label "type: infrastructure". It is possible that the low frequency of occurrence of the label in the training dataset makes it difficult for the model to recommend. Even if we resample the unbalanced dataset, the accuracy of some low-frequency labels still cannot be improved. For the second sample, The issue only has a title but no description, so the model figured out from "don't work" that this issue is related to a bug, but failed to infer which version the issue is related to. Interestingly, even though "version" appears in the title, the model does not recommend version-related labels for issues.

The last Table 6 shows some samples where BERT predicted correctly but CNN predicted incorrectly. We discuss how BERT is actually better than CNN. In the top-2 recommendation of the first example, both two models are able to identify the "type: bug" label accurately, but CNN seems to have trouble identifying the version-related labels. In the second sample, "Category: Support" is a label used by the chart.js project. Later, "Category: Support" was changed to "type: support", but the label in the issue was still retained. Here we can see that when BERT encounters labels with similar meanings, it can infer that the two

Table 4
Example issues with correct predictions from BERT.

	Title	Description	Ground truth	Predictions
1	How add space between labels in group?	I try create horizontal bar, but I don't how add space between labels 'Yes', 'No' and 'Maybe' and add roundness for bars. Please, help me. My code: <CODE> Now: <URL> Necessary: <URL>	Type: support	Type: support, v1.x, type: enhancement, status: duplicate, Version: 2.x
2	[FEATURE] Custom angles for polarArea graph	Provide custom value for each angle of the polar graph. Every angle is 360/#values. Provide the custom angles in the options like this <CODE> and use it in <CODE>	Help wanted, type: enhancement	Type: enhancement, help wanted, type: support, status: implement externally, status: duplicate

Table 5
Example issues with incorrect predictions from BERT.

	Title	Description	Ground truth	Predictions
1	Add more documentation around running docs locally	Should be able to get local environment for gitbook docs to work through CONTRIBUTING.md Local dev is hard since there is only gulp docs to build the static site and if you want a local dev server, you have to install gitbook cli and a bunch of commands Add more docs to contributing about how to run docs locally. Add gitbook serve to gulp or npm tasks	Type: infrastructure, type: documentation	Type: documentation, help wanted, type: enhancement, type: bug, Version: 2.x
2	Minified version doesn't work, but regular does.	No description provided.	v1.x, type: bug	Type: bug, type: support, type: enhancement, help wanted, status: duplicate

Table 6
Example issues where BERT predicted correctly but CNN predicted wrong.

	Title	Description	Ground truth	BERT predictions	CNN predictions
1	v2 graph overflows on y Axis	<CODE> As you can see, my graph is strangely overflowing. The min y value is 0 and the max y value is about 50. The y-axis-0 (y axis by default, didn't touch anything about it) has only 9 ticks valued from 0 to 9. How come?	v2.x, type: bug,	Type: bug, v2.x, help wanted, Priority: p1, status: duplicate	Type: bug, type: support, status: duplicate, help wanted, v2.x
2	Remove padding	Hello, I tried to remove the padding in the canva but nothing works. I already search I old issues. As you can see on the screen there is a left and bottom padding : Here is my conf : <CODE>	Category: Support	Type: support, Category: Support, status: needs test case, v1.x, type: bug	Type: support, type: bug, help wanted, Needs Investigation, status: needs test case

labels have similar meanings from the content of the issue. When recommending one of them, the other one will be recommended by BERT as an alternative answer. However, CNN fails to make such inferences.

6.4. Implications

Our experimental results and manual analysis show that it is feasible to recommend labels for issues using a neural network. It still needs developers to confirm the labels added to issues. However, the automatic recommendation of labels can greatly reduce the workload of developers and decrease the number of errors in the process of labeling.

6.4.1. Dataset size

The results of our experiments show that the transformer-based pre-trained contextual language models may not perform as well as other models in small datasets. However, the transformer-based models are far better than the traditional word embedding language models most of the time when there is enough training data.

6.4.2. Further pre-training

According to our experiments, further pre-training models in software engineering related corpus can significantly improve performance. However, the improvement brought by further pre-training on different datasets is varying. Multiple attempts are needed in different datasets to find the dataset which brings the most improvement.

6.4.3. Model size

In our experiments, the model with more layers does show better performance. GPT2-Medium has better performance than BERT-Base. However, the improvement is not significant, and the deeper models take up more GPU memory.

6.4.4. Out-of-dictionary words

In addition, word-based language models have a problem with out-of-dictionary words (OOV). If some important words are not in the dictionary, these out-of-dictionary words will be replaced with a UNK token and the information will be lost. One idea is to use characters as the basic unit to build the character-level model, but the input sequence will be longer since the basic unit is replaced with characters. The longer the input sequence length, the harder the model is able to learn long-distance dependencies and the longer the training time. Another approach is to use subwords as the basic unit. The pre-trained contextual language model mostly uses subword representations for tokenizer, which breaks words into small and frequent parts and then returns a sequence of subword tokens instead of whole word tokens. For example, BERT uses WordPiece algorithm [29] and GPT2 uses BPE algorithm [37] in the byte sequences of text.

6.5. Threats to validity

There are some threats that can affect the validation of our experimental results.

6.5.1. Internal validity

Label imbalance. Due to the different frequency of labels, labels with few occurrences often lack sufficient learning samples. To reduce this threat, we resampled the training dataset by the frequency of labels and reported the performance according to appropriate metrics. Some other sampling strategies may help to solve this problem, such as SMOTE [38] and MLSMOTE [39].

Evaluation Metrics. In this paper, we use example-based metrics Accuracy@k, Precision@k, Recall@k, and F1-score@k to evaluate the performance of the algorithms which are widely used in recommendation systems. Evaluation in terms of label-based metrics can be our future work.

6.5.2. External validity

Top-k. In this paper, k is set as 2 because 90% issues only have less than three labels. If k is more than 2, it will lead to lower precision and higher recall. It is possible that the higher or lower k value can bring a higher F-measure in some special projects.

Generalizibility. In this paper, we use the GitHub dataset and select the 100 most popular projects with different issues and labels for classifiers comparison. The performance of these deep learning models needs to be validated in more projects.

7. Conclusion

In this paper, we compared different deep learning language models. We used the issue history in the 73 most popular open source projects collected from GitHub as datasets for training and testing the language models. We first explored the feasibility of pre-training the contextual language models to issue label recommendations in GitHub. The suggestions we make based on our experimental results can help future researchers further improve label recommendation in GitHub.

From the empirical study, we came to the following conclusions.

- We showed that CNN and BERT are significantly better than BiLSTM and RCNN. When the training dataset is large enough, the performance of BERT is better than CNN. If the size of the training dataset is small, CNN will be a better choice.
- We further pre-trained the BERT on software engineering related corpus and found that further pre-training on domain-specific corpus can improve the performance.
- Although each project has its optimal parameters, we recommend using GLoVe as word embeddings and setting k as 2 for most projects according to our experiments. Generally, the larger the parameter number of the pre-training contextual language model, the better the performance is obtained.
- For pre-trained language models, the more layers of the model, the better the performance. According to the experiments in this paper, BERT has better results than GPT2 in the classification task.

There are still some challenges to be solved for deep learning models. First of all, the inference speed of BERT can be faster than BiLSTM and RCNN because the transformer can be highly parallelized. Even so, the training of BERT is still a very time-consuming and computing-intensive step. Second, we train an independent model for each project to recommend labels because different projects have different label sets and labeling rules. A cross-project recommendation model can save more training time. Furthermore, there is some other information in the issue besides text, such as code and image. If we can take advantage of the content, the performance of the recommendation may be further improved. Finally, most of the previous research used content-based recommendation models. These approaches can be combined with collaborative filtering, matrix decomposition, and other methods used in recommendation systems.

CRediT authorship contribution statement

Jun Wang: Conceptualization, Methodology, Software, Data curation, Writing – original draft. **Xiaofang Zhang:** Investigation, Writing – review & editing. **Lin Chen:** Validation, Writing – review & editing.

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

This work is partially supported by the National Natural Science Foundation of China (61772263, 61872177), Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Priority Academic Program Development of Jiangsu Higher Education Institutions.

References

[1] P. Zhou, J. Liu, X. Liu, Z. Yang, J. Grundy, Is deep learning better than traditional approaches in tag recommendation for software information sites? Inf. Softw. Technol. 109 (2019) 1–13.
[2] T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, in: ICLR (Workshop Poster), 2013.
[3] K. Lee, J. Devlin, M.-W. Chang, K. Toutanova, BERT: Pre-training of deep bidirectional transformers for language understanding, in: Proceedings of NAACL-HLT, 2019, pp. 4171–4186.
[4] M.E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, L. Zettlemoyer, Deep contextualized word representations, in: Proceedings of NAACL-HLT, 2018, pp. 2227–2237.
[5] R. Alec, N. Karthik, S. Tim, S. Ilya, Improving Language Understanding with Unsupervised Learning, Tech. Rep., Technical report, OpenAI, 2018.
[6] C. Treude, M.-A. Storey, How tagging helps bridge the gap between social and technical aspects in software development, in: 2009 IEEE 31st International Conference on Software Engineering, IEEE, 2009, pp. 12–22.
[7] G. Tsoumakas, I. Katakis, Multi-label classification: An overview, Int. J. Data Warehous. Min. (IJDWM) 3 (3) (2007) 1–13.
[8] X. Xia, D. Lo, X. Wang, B. Zhou, Tag recommendation in software information sites, in: 2013 10th Working Conference on Mining Software Repositories (MSR), IEEE, 2013, pp. 287–296.
[9] D. Yang, Y. Xiao, Y. Song, J. Zhang, K. Zhang, W. Wang, Tag propagation based recommendation across diverse social media, in: Proceedings of the 23rd International Conference on World Wide Web, 2014, pp. 407–408.
[10] P. Zhou, J. Liu, Z. Yang, G. Zhou, Scalable tag recommendation for software information sites, in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2017, pp. 272–282.
[11] S. Wang, D. Lo, B. Vasilescu, A. Serebrenik, Entagrec++: An enhanced tag recommendation system for software information sites, Empir. Softw. Eng. 23 (2) (2018) 800–832.
[12] J. Liu, P. Zhou, Z. Yang, X. Liu, J. Grundy, Fasttagrec: fast tag recommendation for software information sites, Autom. Softw. Eng. 25 (4) (2018) 675–701.
[13] C. Li, L. Xu, M. Yan, J. He, Z. Zhang, Tagdeeprec: Tag recommendation for software information sites using attention-based bi-LSTM, in: International Conference on Knowledge Science, Engineering and Management, Springer, 2019, pp. 11–24.
[14] J. Cabot, J.L.C. Izquierdo, V. Cosentino, B. Rolandi, 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, 2015, pp. 550–554.

[15] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, Y.-G. Guéhéneuc, Is it a bug or an enhancement? A text-based approach to classify change requests, in: Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds, 2008, pp. 304–318.
[16] K. Herzig, S. Just, A. Zeller, It's not a bug, it's a feature: how misclassification impacts bug prediction, in: 2013 35th International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 392–401.
[17] R. Kallis, A. Di Sorbo, G. Canfora, S. Panichella, Ticket tagger: Machine learning driven issue classification, in: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2019, pp. 406–409.
[18] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, S. Fidler, Aligning books and movies: Towards story-like visual explanations by watching movies and reading books, in: Proceedings of the IEEE International Conference on Computer Vision, 2015, pp. 19–27.
[19] E. Guzman, D. Azócar, Y. Li, Sentiment analysis of commit comments in GitHub: an empirical study, in: Proceedings of the 11th Working Conference on Mining Software Repositories, 2014, pp. 352–355.
[20] R. Padhye, S. Mani, V.S. Sinha, A study of external community contribution to open-source projects on GitHub, in: Proceedings of the 11th Working Conference on Mining Software Repositories, 2014, pp. 332–335.
[21] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780.
[22] Y. Kim, Convolutional neural networks for sentence classification, in: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, 2014, pp. 1746–1751.
[23] S. Lai, L. Xu, K. Liu, J. Zhao, Recurrent convolutional neural networks for text classification, in: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 29, 2015, pp. 2267–2273.
[24] Y. Chen, L. Liu, J. Tao, R. Xia, Q. Zhang, K. Yang, J. Xiong, X. Chen, The improved image inpainting algorithm via encoder and similarity constraint, Vis. Comput. (2020) 1–15.
[25] Y. Chen, H. Zhang, L. Liu, J. Tao, Q. Zhang, K. Yang, R. Xia, J. Xie, Research on image inpainting algorithm of improved total variation minimization method, J. Ambient Intell. Humaniz. Comput. (2021) 1–10.
[26] Y. Chen, L. Liu, J. Tao, X. Chen, R. Xia, Q. Zhang, J. Xiong, K. Yang, J. Xie, The image annotation algorithm using convolutional features from intermediate layer of deep learning, Multimedia Tools Appl. 80 (3) (2021) 4237–4261.
[27] Y. Chen, L. Liu, V. Phonevilay, K. Gu, R. Xia, J. Xie, Q. Zhang, K. Yang, Image super-resolution reconstruction based on feature map attention mechanism, Appl. Intell. (2021) 1–14.
[28] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, P. Kuksa, Natural language processing (almost) from scratch, J. Mach. Learn. Res. 12 (Aug) (2011) 2493–2537.
[29] Y. Wu, M. Schuster, Z. Chen, Q.V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al., Google's neural machine translation system: Bridging the gap between human and machine translation, 2016, arXiv preprint arXiv:1609.08144.
[30] J. Pennington, R. Socher, C.D. Manning, Glove: Global vectors for word representation, in: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2014, pp. 1532–1543.
[31] J. Zhang, C. Li, D. Cao, Y. Lin, S. Su, L. Dai, S. Li, Multi-label learning with label-specific features by resolving label correlations, Knowl.-Based Syst. 159 (2018) 148–157.
[32] G. Shani, A. Gunawardana, Evaluating recommendation systems, in: Recommender Systems Handbook, Springer, 2011, pp. 257–297.
[33] P. Bojanowski, E. Grave, A. Joulin, T. Mikolov, Enriching word vectors with subword information, Trans. Assoc. Comput. Linguist. 5 (2017) 135–146.
[34] F. Wilcoxon, Individual comparisons by ranking methods, in: Breakthroughs in Statistics, Springer, 1992, pp. 196–202.
[35] M. Friedman, The use of ranks to avoid the assumption of normality implicit in the analysis of variance, J. Amer. Statist. Assoc. 32 (200) (1937) 675–701.
[36] P. Nemenyi, Distribution-free multiple comparisons (doctoral dissertation, princeton university, 1963), Diss. Abstr. Int. 25 (2) (1963) 1233.
[37] R. Sennrich, B. Haddow, A. Birch, Neural machine translation of rare words with subword units, in: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2016, pp. 1715–1725.
[38] N.V. Chawla, K.W. Bowyer, L.O. Hall, W.P. Kegelmeyer, Smote: synthetic minority over-sampling technique, J. Artificial Intelligence Res. 16 (2002) 321–357.
[39] F. Charte, A.J. Rivera, M.J. del Jesus, F. Herrera, Mlsmote: Approaching imbalanced multilabel learning through synthetic instance generation, Knowl.-Based Syst. 89 (2015) 385–397.