4-2019

DeepReview: Automatic code review using deep multi-instance learning

Hengyi LI
Nanjing University

Shuting SHI
Nanjing University

Ferdian THUNG
Singapore Management University, ferdiant.2013@phdis.smu.edu.sg

Xuan HUO
Nanjing University

Bowen XU
Singapore Management University, bowenxu.2017@phdis.smu.edu.sg

See next page for additional authors

DeepReview: Automatic Code Review Using Deep Multi-instance Learning

Heng-Yi Li[1], Shu-Ting Shi[1], Ferdian Thung[2], Xuan Huo[1], Bowen Xu[2], Ming Li[1(✉)], and David Lo[2]

[1] National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China
{lihy,shist,huox,lim}@lamda.nju.edu.cn
[2] School of Information Systems, Singapore Management University, Singapore, Singapore
{ferdiant.2013,bowenxu.2017}@phdis.smu.edu.sg, davidlo@smu.edu.sg

Abstract. Code review, an inspection of code changes in order to identify and fix defects before integration, is essential in Software Quality Assurance (SQA). Code review is a time-consuming task since the reviewers need to understand, analysis and provide comments manually. To alleviate the burden of reviewers, automatic code review is needed. However, this task has not been well studied before. To bridge this research gap, in this paper, we formalize automatic code review as a multi-instance learning task that each change consisting of multiple hunks is regarded as a bag, and each hunk is described as an instance. We propose a novel deep learning model named DeepReview based on Convolutional Neural Network (CNN), which is an end-to-end model that learns feature representation to predict whether one change is approved or rejected. Experimental results on open source projects show that DeepReview is effective in automatic code review tasks. In terms of F1 score and AUC, DeepReview outperforms the performance of traditional single-instance based model TFIDF-SVM and the state-of-the-art deep feature based model Deeper.

Keywords: Software mining · Machine learning · Multi-instance learning · Automatic code review

1 Introduction

Software Quality Assurance (SQA) is essential in software development. Software code review [16] is an important inspection of code changes written by an independent third-party developer in order to identify and fix defects before integration. Effective code review can largely improve the software quality.

However, code review is a very time-consuming task that the reviewer needs to spend much time to understand, analyze and provide comments for the code review request. Additionally, with the rapid growth of software, the number of

Q. Yang et al. (Eds.): PAKDD 2019, LNAI 11440, pp. 318–330, 2019.
https://doi.org/10.1007/978-3-030-16145-3_25

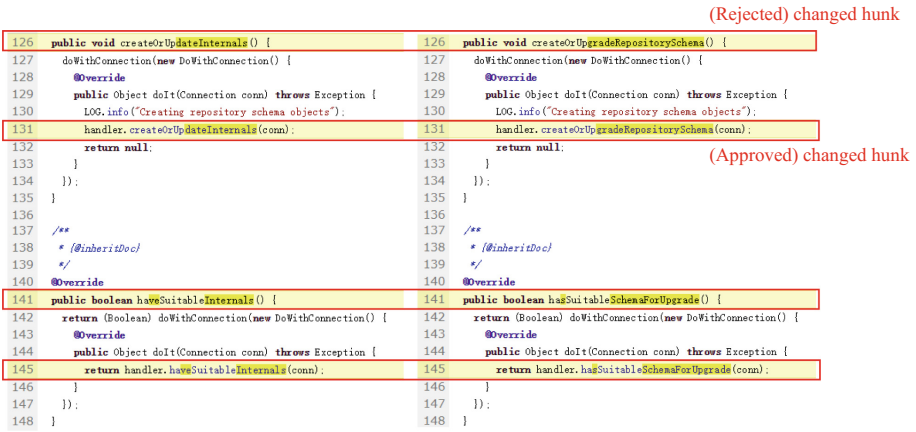


Fig. 1. An example of rejected change `JdbcRepository.java` of review request 26657 from Apache. This change contains four hunks and only one hunk is rejected.

review requests are growing, which leads to a heavier burden on code reviewers. Therefore, automatic code review is important to alleviate the burden of reviewers.

Recently, some studies have been proposed to improve the effectiveness of code review [1,16]. Thongtanunam et al. [16] revealed that 4%–30% of reviews have code-reviewer assignment problems. They proposed a code reviewer recommendation approach named REVFINDER to solve it by leveraging the file location information. Ebert et al. [1] proposed to identify the factors that confuse reviewers and understand how confusion impacts the efficiency of code reviewers. However, the task of automatic code review has not been well studied previously.

Considering the above issues, an automated approach is needed, which is able to help reviewers to review the code submitted by developers. Usually, a review request submitted by developers contains some changes of source code in the form of `diff` files and textual descriptions indicating the intention of the change. Notice that each change may contain multiple change hunks and each hunk corresponds to a set of continuous lines of code. For example, Fig. 1 shows the change in the file `JdbcRepository.java` of review request 26657 from Apache project. This change contains four hunks. One of the most common ways to analyze this change is to combine all hunks together and generate a unified feature to represent the change. However, this method may lead to two problems. First, the hunks appearing in each change may be discontinuous and unrelated to one another. Directly combining the hunks together may generate misleading feature representations, leading to a poor prediction performance. Second, when the change is rejected, not every hunk in the change is rejected. Some hunks have no issues and can be approved by reviewers. So the approved hunks and the rejected hunks should not be processed together for feature extraction. Therefore, separately generating features from each individual hunk in automatic code

review is needed. If the label (referring to *approved* or *rejected*) of each hunk is available, we can directly build classification models on hunk data. However, in code review tasks, the label of each hunk is hard to be obtained while the label of each change can be extracted. A question arises here, can we build a model to generate hunk-level feature representations for automatic code review based on change-level labels?

To solve this problem, we formulate the automatic code review as a binary classification task in the *multi-instance learning* setting. Instead of regarding each change as an individual instance following traditional machine learning method, multi-instance learning method regards each change as a bag of instances while each hunk of the change is described as an instance. The basic assumption in multi-instance learning is that if one instance is positive then the bag is also positive, which is consistent with code review task whereas if one hunk is rejected then the change is also rejected. In our paper, we propose a deep learning model named DeepReview based on Convolutional Neural Network (CNN) via multi-instance learning, which is able to automatically learn semantic features from each hunk and predict if one change is approved or rejected. Additionally, in order to obtain the features that capture the difference of code changes, DeepReview firstly recovers the code before change (old source code) and after change (new source code) according to the `diff` markers. These snippers are then fed in to the deep model to generate feature presentation, based on which the label of each change is predicted. We conduct experiments on large datasets collected from open source Apache projects for evaluation. The results in terms of widely-used metrics AUC and F1 score indicate that DeepReview is effective in automatic code review and outperforms previous state-of-the-art feature representation methods previously used for related software engineering tasks.

The contributions of our work are several folds:

- We are the first to study automatic code review task as multi-instance learning task. One change always contains multiple hunks, where each hunk is described as an instance and the change can be represented by a set of instances. Experiment results on five large datasets show that the proposed multi-instance model is effective in automatic code review tasks.
- We propose a novel deep learning model named DeepReview based on Convolutional Neural Network (CNN), which learns semantic feature representation from source code change and change descriptions, to predict if one change is approved or rejected.

2 The DeepReview Approach

In this section, we introduce the details of applying DeepReview for automatic code review. The goal of this task is to predict if one code change of review request submitted by developers is approved or rejected. The general process of automatic code review based on machine learning model is illustrated in Fig. 2.

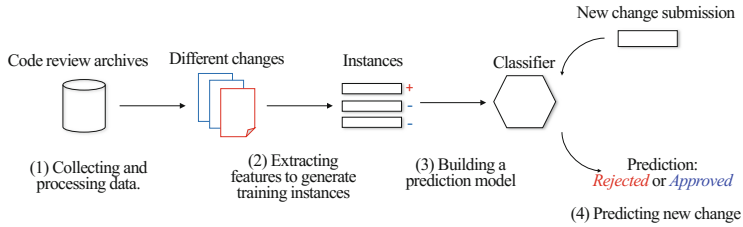


Fig. 2. The general automatic code review process based on machine learning model.

The automatic code review prediction process mainly contains several parts:

- Collecting data from code review systems and processing data.
- Generating feature representations of the input data.
- Training a classifier based on the generated features and labels.
- Predicting if a new change is approved or rejected.

In the following subsections, we first introduce the general framework of DeepReview in Subsect. 2.1, and the data processing is reported in Subsect. 2.2. The core parts of DeepReview is elaborated in Subsects. 2.3 and 2.4.

2.1 The Framework of DeepReview

We introduce some notations of our framework. Let $\mathcal{C}^o = \{c_1^o, c_2^o, \ldots, c_N^o\}$ and $\mathcal{C}^n = \{c_1^n, c_2^n, \ldots, c_N^n\}$ denotes the collection of old code and new code. Let $\mathcal{D} = \{d_1, d_2, \ldots, d_N\}$ denotes the collection of change descriptions, where N is the number of changes. In this paper, we formalize the code review as a learning task, which attempts to learn a prediction function $f : \mathcal{X} \mapsto \mathcal{Y}$. $x_i \in \mathcal{X} = (c_i^o, c_i^n, d_i)$ denotes each change, where c_i^o and c_i^n denotes the i-th old code (before changed) and new code (after changed) respectively. Here $c_i^o = \{h_{i1}^o, h_{i2}^o, \ldots, h_{im}^o\}$ and $c_i^o = \{h_{i1}^n, h_{i2}^n, \ldots, h_{im}^n\}$ contains multiple hunks and m is the number of hunks. d_i denotes the text description of i-th change. $y_i \in \mathcal{Y} = \{1, 0\}$ indicates whether the change is approved or rejected.

We instantiate the code review prediction model by constructing a multi-instance learning based deep neural network named DeepReview. The general framework of DeepReview is illustrated in Fig. 3. The DeepReview model contains three parts: input layers, instance feature generation layers and multi-instance based prediction layers.

In the DeepReview model, each hunk of source code change is regarded as an instance. In the input layers, the source code and text description of each instance is encoded as feature vectors and then are fed into the neural network for processing. The details of data processing in the input layers will be discussed in Subsect. 2.2. Then the encoded data of each instance is fed into instance feature generation layers. In these layers, DeepReview utilizes different convolutional neural networks (CNN) to extract features from the source code input and the textual description input. The convolutional neural networks for programming

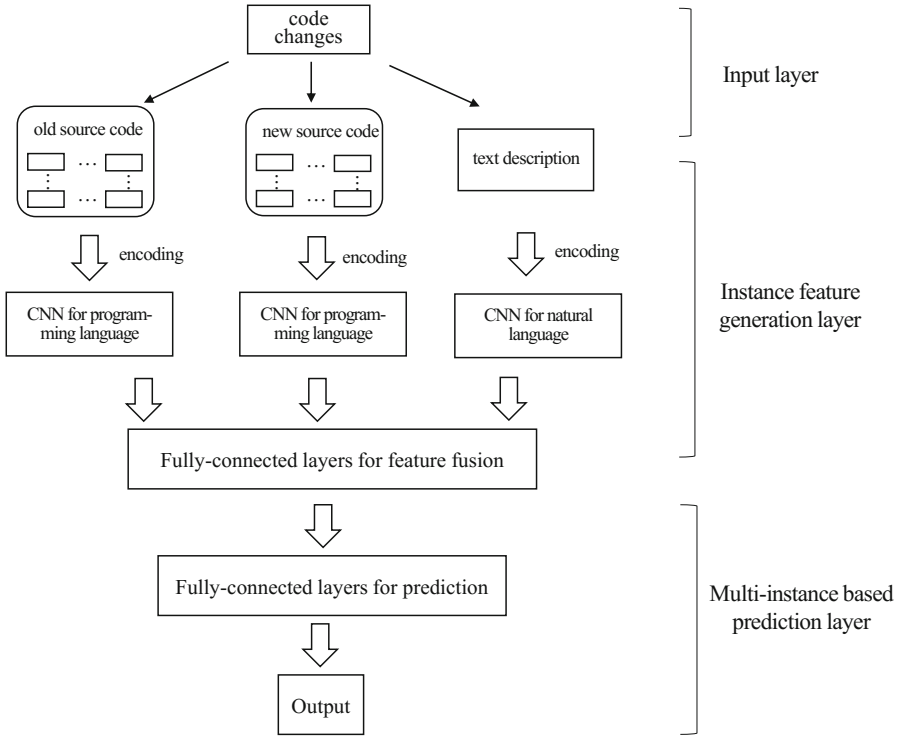


Fig. 3. The general framework of DeepReview for automatic code review prediction. The DeepReview model contains three parts: input layer, instance feature generation layer, multi-instance based prediction layer.

language processing (called PCNN) is carefully designed respecting to the characteristics of source code, which is similar to the network structure in [4]. The convolutional neural networks for textual description processing (called NCNN) is a standard way in [6]. Then the generated middle-level features of old code, new code and textual descriptions of each instance are fused to learn a unified feature representation via fully-connected networks mapping. Finally, after generating unified feature representations, the DeepReview model make a prediction for each change via the multi-instance learning way in the multi-instance based prediction layers.

2.2 Data Processing

The datasets used for automatic code review is the changed source code submitted by developers, which always appears in form of `diffs` and contains both source code and `diff` markers (e.g., `+` stands for adding a line, `-` stands for deleting a line). The main features in code changes are the difference between the code before changed and after changed. So in data preprocessing shown in

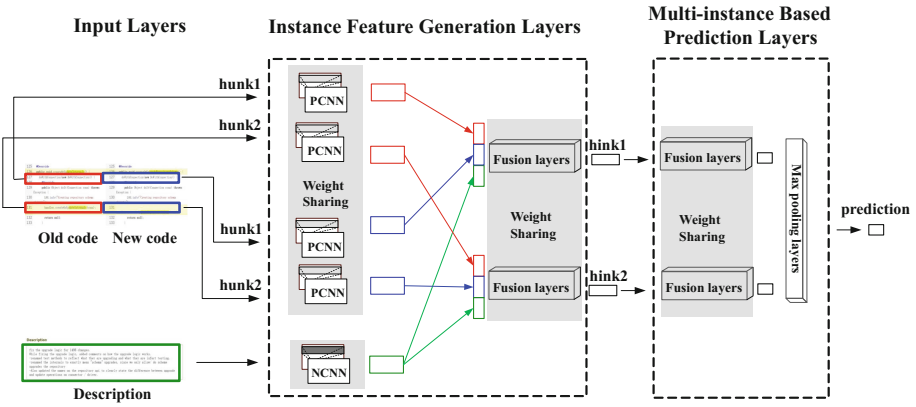


Fig. 4. Automatic code review by DeepReview. When a change is processed for prediction, three parts of the change (old code, new code and text descriptions) are firstly encoded as feature vectors to feed into deep model. Then three parts of convolutional neural networks are followed to extracte semantic features for source code and text description separately. After that a fully-connected network is used to get fusion feature for hunks. Finally, another fully-connected network and a max-pooling layer is connected to generate a prediction indicating approved or rejected of the change.

the left part of Fig. 4, we extract both old code (before changed) and new code (after changed) from `diffs` as input. We also use the change descriptions since they contain the goal of this change and are helpful to improve the prediction performance.

After splitting `diff` files into old code, new code and text description, a pre-trained word2vec [10] technique is used to encode every token as vector representations (e.g., a 300 dimension vector), which has been shown effective in processing textual data and widely used in text processing tasks [6,10]. In a similar way, we split descriptions as words and encode them as vector representations too. All these vector representations are sent into the deep neural network to learn the semantic features.

2.3 Instance Feature Generation Layer

DeepReview takes old source code (before change) and new source code (after change) along with the text descriptions as inputs. Noticing that the source code and text descriptions are with different structures. Therefore we use PCNN network for code and NCNN network for text to extract feature, respectively.

As aforementioned, each change will contain multiple hunks and different hunks are individual instance, therefore the instance features should be extracted separately by the same neural network. In other words, the weight of PCNN is shared for all code hunks. In this way, we can get unbiased feature representations for each hunk with both old code and new code.

Suppose one change contains m modified hunks. Let $(\mathbf{z}^o_{i1}, \mathbf{z}^o_{i2}, \ldots, \mathbf{z}^o_{im})$ denotes the middle-level vectors of old source code c^o_i, $(\mathbf{z}^n_{i1}, \mathbf{z}^n_{i2}, \ldots, \mathbf{z}^n_{im})$ denotes the middle-level vectors of new source code c^n_i and $\mathbf{z}^t_i$ denotes the middle-level vectors of text description d_i. In the instance feature generation layers, DeepReview first concatenates this three part for each instance as following:

$$\mathbf{z}^h_{ij} = \mathbf{z}^o_{ij} \odot \mathbf{z}^n_{ij} \odot \mathbf{z}^t_i \tag{1}$$

where $\odot$ is the concatenating operation and the generated $\mathbf{z}^h_{ij}$ represents the features of the j-th hunk of the i-th change (referring to one instance).

To capture the difference between new code and old code as well as the relation between code change and change description, this concatenated features are then fed into fully-connected networks for feature fusion.

2.4 Multi-instance Based Prediction Layer

In the prediction layers, we first make a prediction for each hunk (also called instance) using fully-connected networks following a sigmoid layer based on the generated hunk representations. Similarly, all the fully-connected networks are shared weights to each hunk so that the generated prediction does not have bias. The output prediction of each hunk $\mathbf{p_i} = (p_{i1}, p_{i2}, \ldots, p_{im})$ is generated.

In the multi-instance setting, if any instance is positive (rejected), the bag is also positive (rejected). So the maximum value of predictions for hunks is used for predicting the label of each change. Then, a max-pooling layer is employed to get the final prediction for the change, that is $\hat{p_i} = \max\{\mathbf{p}\}$.

Specifically, the parameters of the convolutional neural networks layers can be denoted as $\Theta = \{\theta_1, \theta_2, \ldots, \theta_l\}$ and the parameters of the fully-connected networks layers can be denoted as $W = \{\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_3\}$. Therefore, the loss function implied in DeepReview is:

$$\mathcal{L}(\Theta, W) = -\sum_{i=1}^{N}(c_a y_i \log \hat{p_i} + c_r(1 - y_i)\log(1 - \hat{p_i})) + \lambda\Omega(f) \tag{2}$$

where $\mathcal{L}$ is a cross-entropy loss, $\Omega(f)$ is the regularization term which imposes regularization (e.g., L_2 regularization) on the weights of model, and λ is the trade-off parameter balancing these two terms. c_a denotes the cost of incorrectly predicting a rejected change as approved and c_r denote the cost of incorrectly predicting a approved change as rejected. This objective function can be effectively optimized by SGD (Stochastic gradient descent) algorithm.

3 Experiments

To evaluate the effectiveness of DeepReview, we conduct experiments on thousands of code reviews from open source software projects and compare with several state-of-the-art code review methods.

3.1 Experiment Settings

The datasets used in our experiment are from Apache[1] Code Review Board, which are also analyzed by prior studies on code reviews [13,14]. We downloaded all reviews on October 2017 and selected only code reviews in which the reviewers highlighted the line numbers that they have issues with, totally 1,011 code reviews. We further extracted five repositories with the largest number of involved files in the collected code reviews – the different datasets and their statistics are shown in Table 1. For each repository, we have more than 1,000 involved files and at least 3,500 hunks.

Table 1. Statistics of our data sets.

Datasets	#changes	#hunks	#rejected
cloudstack-git	1,682	6,171	128
aurora	1,161	6,762	168
drill-git	1,015	3,575	43
accumulo	1,011	5,798	152
hbase-git	1,009	6,702	140

As indicated by Table 1, the number of rejected hunks is only a small part of all hunks and the datasets are very imbalanced. Therefore, we use F1 to evaluate the performance; F1 has been widely used in imbalanced learning settings. Additionally, we record the AUC, which is a non-parametric method to evaluate model performance and is unaffected by class imbalance. The evaluation metrics used in our experiments were adopted to evaluate many approaches that automate various software engineering tasks [4,5,9,12,17].

We compare the proposed model DeepReview with following baseline methods and some of its variants:

- TFIDF-LR [2], which uses Term Frequency-Inverse Document Frequency (TFIDF) feature to represent source code changes and Logistic Regression (LR) for classification.
- TFIDF-SVM, which uses TFIDF features to represent source code changes and Support Vector Machine (SVM) for classification.
- Deeper [20], one of the state-of-the-art deep learning models on software engineering, which extracts deep features from changes with DBN models and then apply Logistic Regression (LR) for classification.
- Deeper-SVM, a slight variant of Deeper, which uses DBN model for feature extraction and then apply Support Vector Machine for classification.
- DeepReview-SingleInstance, one variant of DeepReview, which does not consider the multi-instance setting and concatenate the all hunks together as one instance for input.

[1] https://reviews.apache.org/r/.

– DeepReview-diff, one variant of DeepReview, which does not separate the code change and taking diff marks and diff code as input.

The settings of DeepReview and its variants are introduced here: in the convolution layers, we use activation function $\sigma(x) = max(x, 0)$. Also, we set the size of convolution windows as 2 and 3 with 100 feature maps each.

3.2 Experiment Results

For each dataset, 10-fold cross validation is repeated 5 times and we report the average value of all compared methods in order to reduce the evaluation bias. We also apply the statistic test to evaluate the significance of DeepReview. Pairwise t-test at 95% confidence level is conducted.

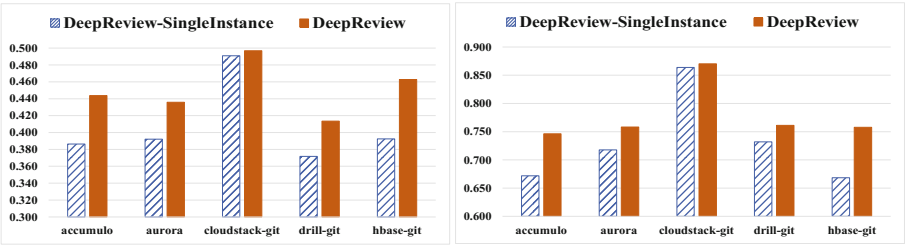
We firstly compare our proposed model DeepReview with several traditional non-multi instance models. One of the most common methods is to employ Vector Space Model (VSM) to represent the changes. In addition, we compare DeepReview with latest deep learning based models Deeper [20] on software engineering, which applies Deep Believe Network for semantic feature extraction. The results are shown in Tables 2 and 3. The highest results of each repository is highlighted in bold. The compared methods that are significantly inferior than our approach will be marked with "∘" and significantly better than our approach be marked with "•".

Table 2. The performance comparison in terms of F1 on all methods.

Datasets	TFIDF-LR	TFIDF-SVM	Deeper	Deeper-SVM	DeepReview
accumulo	0.219∘	0.231∘	0.208∘	0.199∘	**0.444**
aurora	0.202∘	0.214∘	0.352∘	0.298∘	**0.436**
cloudstack-git	0.252∘	0.276∘	0.392∘	0.257∘	**0.497**
drill-git	0.213∘	0.235∘	0.277∘	0.226∘	**0.414**
hbase-git	0.235∘	0.257∘	0.182∘	0.142∘	**0.463**
Avg.	0.224∘	0.243∘	0.282∘	0.224∘	**0.451**

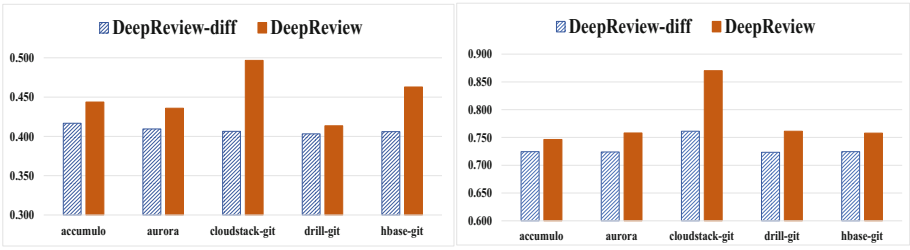
Table 3. The performance comparison in terms of AUC on all methods.

Datasets	TFIDF-LR	TFIDF-SVM	Deeper	Deeper-SVM	DeepReview
accumulo	0.635∘	0.678∘	0.697∘	0.705∘	**0.746**
aurora	0.577∘	0.629∘	0.687∘	0.566∘	**0.758**
cloudstack-git	0.755∘	0.827∘	0.825∘	0.637∘	**0.870**
drill-git	0.676∘	0.725∘	0.639∘	0.571∘	**0.761**
hbase-git	0.685∘	0.751	0.597∘	0.547∘	**0.758**
Avg.	0.666∘	0.722∘	0.689∘	0.605∘	**0.779**



(a) F1. (b) AUC

Fig. 5. F1 and AUC of the compared methods on five datasets.



(a) F1. (b) AUC.

Fig. 6. F1 and AUC of the compared methods on five datasets.

As indicated in Tables 2 and 3, DeepReview achieves the best performance on all datasets in terms of F1 score. On average, DeepReview can lead to AUC value 0.779, which is significant better than the value achieves by TFIDF-LR (0.666), TFIDF-SVM (0.722). When compared with Deeper and its variant Deeper-SVM, it can be easily find that DeepReview achieves the best F1 score and AUC value. On average, the superiority of DeepReview to other deep feature based methods is statistically significant. In conclusion, the proposed DeepReview is effective in automatic code review prediction, which indicates that DeepReview can learn better features than traditional hand-crafted features or previous deep learning based features.

To evaluate the effectiveness of applying multi-instance learning strategy for code review, we compare our model to traditional single-instance learning model, named DeepReview-SingleInstance. Figure 5a and b show the performance comparison of DeepReview and a variant DeepReview-SingleInstance. It can be observed that DeepReview achieves higher AUC value and F1 score than DeepReview-SingleInstance on all datasets, indicating that multi-instance learning approach is effective in code review task.

To evaluate the effectiveness of applying both source code before and after changes to model the difference features of change, we compare another variant of DeepReview, named DeepReview-diff. We use the same network structure to extract the features of code in `diffs` and fuse it with the features of corre-

sponding change description as the final representations. Figure 6a and b show the performance comparison of DeepReview and its variant DeepReview-diff. Compared to DeepReview-diff, it is clear that DeepReview outperforms it by improving 4.2% in terms of F1 score and 4.7% in terms of AUC on average.

4 Related Work

Many empirical studies aim to help researchers and practitioners to understand code review practice from different perspectives [7,13,15]. To characterize and understand the differences between a diverse set of software projects, Rigby et al. [13] found that many characteristics of code review have independently converged to similar values which indicates general principles of code review, e.g., reviewers prefer discussion and fixing code over reporting defects, the number of involved developers can vary. Kononenko et al. [7] investigated a set of factors that might affect the quality of code review based on a large open-source project Mozilla, and focused on the relationship between human factors (e.g., personal characteristics of developers, team participation and involvement) and code review quality. Tao et al. [15] investigated the reasons behind 300 rejected Eclipse and Mozilla patches by surveying 246 developers. They concluded that the poor quality of the solution, the large size of the involvement of unnecessary changes, the ambiguous documentation of a patch and inefficient communication. Moreover, Thongtanunam et al. [16] revealed that 4%–30% of reviews have code-reviewer assignment problem. Thus, they proposed a code-reviewer recommendation approach REVFINDER to solve the problem by leveraging the file location information. The intuition is that files that are located in similar file paths would be managed and reviewed by experienced code-reviewers. Zanjani et al. [21] also studied on code reviewer recommendation problem and they proposed an approach cHRev by leveraging the specific information in previously completed reviews (i.e., quantification of review comments and their recency).

Recently, deep learning has been applied in software engineering. For example, Yang et al. applied Deep Belief Network (DBN) to learn higher-level features from a set of basic features extracted from commits (e.g., lines of code added, lines of code deleted, etc.) to predict buggy commits [20]. Xu et al. applied word embedding and convolutional neural network (CNN) to predict semantic links between knowledge units in Stack Overflow (i.e., questions and answers) to help developers better navigate and search the popular knowledge base [19]. Lee et al. applied word embedding and CNN to identify developers that should be assigned to fix a bug report [8]. Mou et al. [11] applied tree based CNN on abstract syntax tree to detect code snippets of certain patterns. Huo et al. [3,4] applied learned unified semantic feature based on bug reports in natural language and source code in programming language for bug localization tasks. Wei et al. [18] proposed deep feature learning framework AST-based LSTM network for functional clone detection, which exploits the lexical and syntactical information.

5 Conclusion

In this paper, we are the first to formulate code review as a multi-instance learning task. We propose a novel deep learning model named DeepReview for automatic code review, which takes raw data of a changed code containing multiple hunks along with the textual descriptions as inputs and predicts if one change is approved or rejected. Experimental results on five open source datasets show that DeepReview is effective and outperforms the state-of-the-art models previously proposed for other automated software engineering tasks.

Acknowledgment. This research was supported by National Key Research and Development Program (2017YFB1001903) and NSFC (61751306).

References

1. Ebert, F., Castor, F., Novielli, N., Serebrenik, A.: Confusion detection in code reviews. In: ICSME, pp. 549–553 (2017)
2. Gay, G., Haiduc, S., Marcus, A., Menzies, T.: On the use of relevance feedback in IR-based concept location. In: ICSM, pp. 351–360 (2009)
3. Huo, X., Li, M.: Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In: IJCAI, pp. 1909–1915 (2017)
4. Huo, X., Li, M., Zhou, Z.H.: Learning unified features from natural and programming languages for locating buggy source code. In: IJCAI, pp. 1606–1612 (2016)
5. Jiang, T., Tan, L., Kim, S.: Personalized defect prediction. In: ASE, pp. 279–289 (2013)
6. Kim, Y.: Convolutional neural networks for sentence classification. In: EMNLP, pp. 1746–1751 (2014)
7. Kononenko, O., Baysal, O., Guerrouj, L., Cao, Y., Godfrey, M.W.: Investigating code review quality: do people and participation matter? In: ICSME, pp. 111–120 (2015)
8. Lee, S., Heo, M., Lee, C., Kim, M., Jeong, G.: Applying deep learning based automatic bug triager to industrial projects. In: ESEC/FSE, pp. 926–931 (2017)
9. Menzies, T., Greenwald, J., Frank, A.: Data mining static code attributes to learn defect predictors. IEEE TSE **33**(1), 2–13 (2007)
10. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: NIPS, pp. 3111–3119 (2013)
11. Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z.: Convolutional neural networks over tree structures for programming language processing. In: AAAI, pp. 1287–1293 (2016)
12. Nam, J., Pan, S.J., Kim, S.: Transfer defect learning. In: ICSE, pp. 382–391 (2013)
13. Rigby, P.C., Bird, C.: Convergent contemporary software peer review practices. In: FSE, pp. 202–212 (2013)
14. Rigby, P.C., German, D.M., Storey, M.A.: Open source software peer review practices: a case study of the apache server. In: ICSE, pp. 541–550 (2008)
15. Tao, Y., Han, D., Kim, S.: Writing acceptable patches: an empirical study of open source project patches. In: ICSME, pp. 271–280 (2014)

16. Thongtanunam, P., Tantithamthavorn, C., Kula, R.G., Yoshida, N., Iida, H., Matsumoto, K.I.: Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In: SANER, pp. 141–150 (2015)
17. Wang, S., Liu, T., Tan, L.: Automatically learning semantic features for defect prediction. In: ICSE, pp. 297–308 (2016)
18. Wei, H.H., Li, M.: Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: IJCAI, pp. 3034–3040 (2017)
19. Xu, B., Ye, D., Xing, Z., Xia, X., Chen, G., Li, S.: Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In: ASE, pp. 51–62 (2016)
20. Yang, X., Lo, D., Xia, X., Zhang, Y., Sun, J.: Deep learning for just-in-time defect prediction. In: QRS, pp. 17–26 (2015)
21. Zanjani, M.B., Kagdi, H., Bird, C.: Automatically recommending peer reviewers in modern code review. IEEE TSE **42**(6), 530–543 (2016)