Soft Computing (2021) 25:5619–5632
https://doi.org/10.1007/s00500-020-05559-3

METHODOLOGIES AND APPLICATION

Recommending pull request reviewers based on code changes

Xin Ye[1] · Yongjie Zheng[1] · Wajdi Aljedaani[2] · Mohamed Wiem Mkaouer[3]

Published online: 9 January 2021

Abstract
Pull-based development supports collaborative distributed development. It enables developers to collaborate on projects hosted on GitHub. If a developer wants to collaborate on a project, he/she will fork the repository, make modifications on the forked repository and send a pull request to the development team to ask for a merge of the code changes to the official repository. When the development team receives a pull request, the team members will review the changes and make a decision on whether to accept the changes or not. However, efficiently finding suitable pull request reviewers is a challenge. In this paper, we propose a multi-instance-based deep neural network model to recommend reviewers for pull requests. Given a pull request, our model extracts three features, which pull request title, commit message, and code change. The proposed model extracts the three features automatically from the code changes of every commit in the pull request. The features of different commits are then merged to predict the likelihood that a reviewer candidate is the appropriate reviewer. We use CNN and LSTM-network to learn features since the pull requisition and commit message feature have different structures than code change, written in a programming language. To test the effectiveness of our model, we performed a set of experiments using 43,986 pull requests extracted from 12 open-source projects. We compare our model with two baselines approaches, CoreDevRec and Majority Classes. Experiments demonstrate that our model outperforms two state-of-the-art baselines. For instance, for the TensorFlow project, our model's accuracy in determining the appropriate reviewers is 50.80%, 74.70%, and 84.04%, respectively, in Top-1, Top-3, and Top-5 recommendation.

Keywords Pull request · Reviewer recommendation · Code changes · Artificial neural network

1 Introduction

GitHub is the largest code hosting site for collaborative and distributed development in the world. One of the important features of GitHub is the pull-based development (Gousios et al. 2014; Tsay et al. 2014), which is an emerging software development paradigm for collaborative development. Pull-based development enables any developers to collaborate on any projects hosted on GitHub. For example, for a project on GitHub, only its core team members have the write access to its official repository.

Other outside developers only have the read privileges. If an outside developer wants to contribute to the project, he/she can fork the project to create his/her own copy of the repository with write privileges. The outside developer then can make changes to the forked repository. In case the outside developer wants to contribute to the project he/she previously forked, he/she can send a pull request to the development team in order to ask the team to merge the changes from the forked repository to the official repository. The development team appoints one person to review the pull request and see if the features implemented by the requester, are compatible with the current version of the software. If the review outcome is positive, the pull request is then accepted by the development team, and its corresponding changes will be merged to the official repository. By following this pull-based development workflow, any developer can make contributions to any project on GitHub.

✉ Xin Ye
xye@csusm.edu

Yongjie Zheng
yzheng@csusm.edu

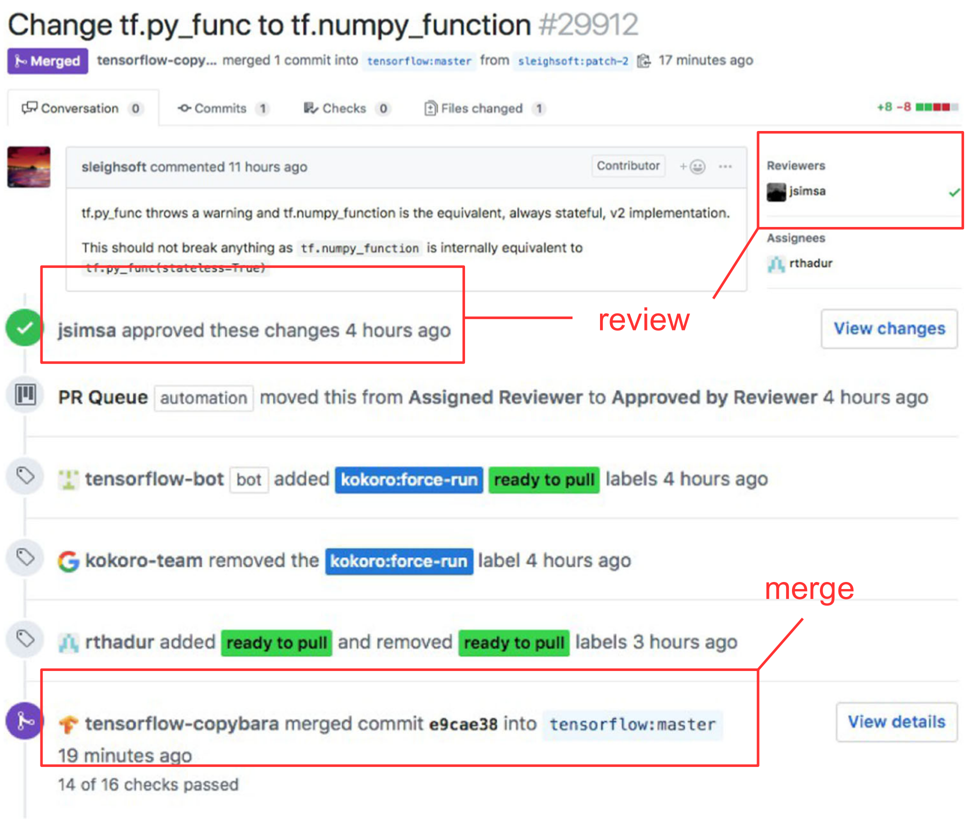
Wajdi Aljedaani
wajdialjedaani@my.unt.edu

Mohamed Wiem Mkaouer
mwmvse@rit.edu

1 California State University, San Marcos, USA

2 University of North Texas, Denton, USA

3 Rochester Institute of Technology, Rochester, USA



Change tf.py_func to tf.numpy_function #29912

Merged tensorflow-copy... merged 1 commit into `tensorflow:master` from `sleighsoft:patch-2` 17 minutes ago

Conversation 0 · Commits 1 · Checks 0 · Files changed 1 · +8 −8

sleighsoft commented 11 hours ago · Contributor

tf.py_func throws a warning and tf.numpy_function is the equivalent, always stateful, v2 implementation.

This should not break anything as `tf.numpy_function` is internally equivalent to `tf.py_func(stateless=True)`

Reviewers: jsimsa ✓

Assignees: rthadur

jsimsa approved these changes 4 hours ago

PR Queue automation moved this from Assigned Reviewer to Approved by Reviewer 4 hours ago

tensorflow-bot bot added kokoro:force-run ready to pull labels 4 hours ago

kokoro-team removed the kokoro:force-run label 4 hours ago

rthadur added ready to pull and removed ready to pull labels 3 hours ago

tensorflow-copybara merged commit e9cae38 into `tensorflow:master` 19 minutes ago

14 of 16 checks passed

Fig. 1 A pull request screenshot

A pull request contains a title and one or more commits. When the development team receive a pull request, the team will review the changes and decide whether to accept the proposed changes by merging it to their codebase or decline them by rejecting the pull request. Figure 1 shows a screenshot of a pull request from the TensorFlow project[1]. As shown in the figure, this pull request was assigned to a team member whose username is *jsimsa* for review. After reviewing the pull request, *jsimsa* approved the changes. The status of the pull request was moved from **Assigned Reviewer** to **Approved by Reviewer**. Then the changes were merged to the official repository by another team member *TensorFlow-copybara*. As shown in the figure, it is noteworthy that the team member who performs the merge operation may not be the same team member who actually reviews the pull request.

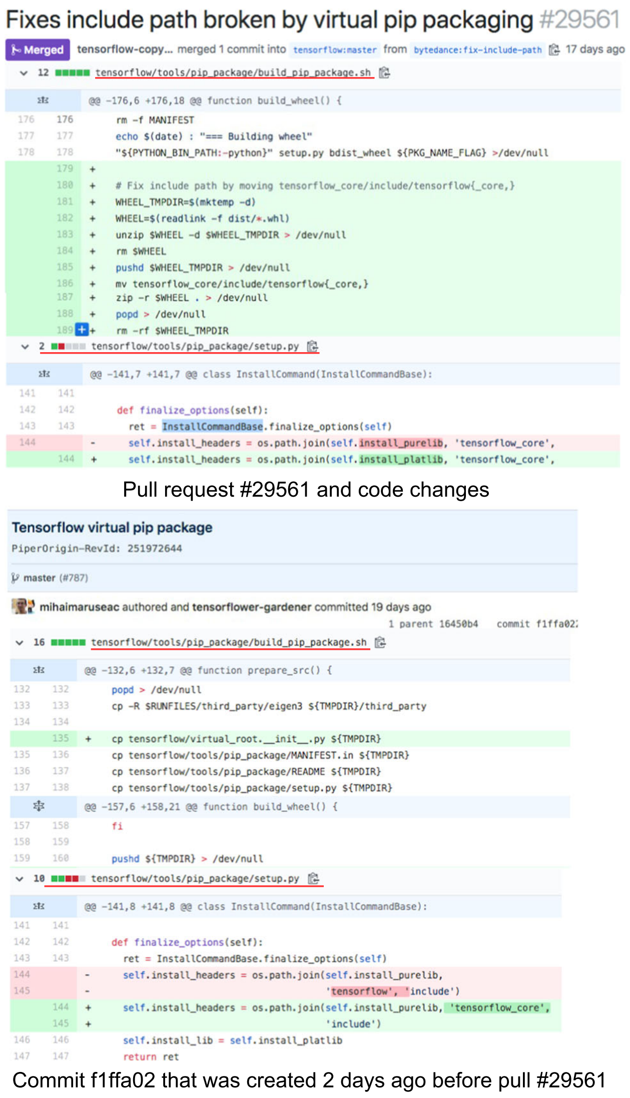
For many open-source projects on GitHub, the development teams frequently receive a large amount of pull requests on a regular basis. For example, the project TensorFlow received 11,717 pull requests from November 9, 2015, to June 24, 2019. The development teams need to spend a lot of time managing pull requests and reviewing code changes.

According to a survey conducted by Gousios et al. (2015) with 749 developers, projects are struggling to review pull requests (Gousios et al. 2015). Since the development team takes the responsibility to assure quality, it is very important to find the appropriate reviewer with the solid knowledge to manage the review of pull requests (de Lima Júnior et al. 2018). However, it is human-intensive and time-consuming to find the appropriate reviewer that has the knowledge to review, approve or reject a pull request, especially in repositories with hundreds, even thousands of contributors. Therefore, finding the appropriate reviewer efficiently is a key challenge of pull-based development (de Lima Júnior et al. 2018).

Recently, researchers have proposed classification algorithms to automatically assign pull requests to appropriate reviewers (de Lima Júnior et al. 2015; Soares et al. 2018; Jiang et al. 2015). These approaches provide initial pieces of evidence that the classification approaches can help automatically find the appropriate reviewers for pull requests. Recent studies have been relying on code authorship and developer's experience as the main properties to narrow the search space for the appropriate pull-request handler. However, they did not take into account code changes, which is known to be the basic building block of developing and maintaining source code, and so, using it may provide useful insights to find the appropriate reviewers. In other terms, the more developers perform code changes in a given file, the more they are knowledgeable in evaluating any requested change on that file.

Figure 2 shows the screenshot of a pull request and the screenshot of a commit previously made in the official repos-

[1] https://github.com/tensorflow/tensorflow/pull/29912.



Fixes include path broken by virtual pip packaging #29561

Merged tensorflow-copy... merged 1 commit into tensorflow:master from bytedance:fix-include-path 17 days ago

```
12 tensorflow/tools/pip_package/build_pip_package.sh
@@ -176,6 +176,18 @@ function build_wheel() {
     rm -f MANIFEST
     echo $(date) : "=== Building wheel"
     "${PYTHON_BIN_PATH:-python}" setup.py bdist_wheel ${PKG_NAME_FLAG} >/dev/null
+
+    # Fix include path by moving tensorflow_core/include/tensorflow{_core,}
+    WHEEL_TMPDIR=$(mktemp -d)
+    WHEEL=$(readlink -f dist/*.whl)
+    unzip $WHEEL -d $WHEEL_TMPDIR > /dev/null
+    rm $WHEEL
+    pushd $WHEEL_TMPDIR > /dev/null
+    mv tensorflow_core/include/tensorflow{_core,}
+    zip -r $WHEEL . > /dev/null
+    popd > /dev/null
+    rm -rf $WHEEL_TMPDIR
2 tensorflow/tools/pip_package/setup.py
@@ -141,7 +141,7 @@ class InstallCommand(InstallCommandBase):

     def finalize_options(self):
       ret = InstallCommandBase.finalize_options(self)
-      self.install_headers = os.path.join(self.install_purelib, 'tensorflow_core',
+      self.install_headers = os.path.join(self.install_platlib, 'tensorflow_core',
```

Pull request #29561 and code changes

Tensorflow virtual pip package

PiperOrigin-RevId: 251972644

master (#787)

mihaimaruseac authored and tensorflower-gardener committed 19 days ago

1 parent 16450b4 commit f1ffa02

```
16 tensorflow/tools/pip_package/build_pip_package.sh
@@ -132,6 +132,7 @@ function prepare_src() {
     popd > /dev/null
     cp -R $RUNFILES/third_party/eigen3 ${TMPDIR}/third_party

+    cp tensorflow/virtual_root.__init__.py ${TMPDIR}
     cp tensorflow/tools/pip_package/MANIFEST.in ${TMPDIR}
     cp tensorflow/tools/pip_package/README ${TMPDIR}
     cp tensorflow/tools/pip_package/setup.py ${TMPDIR}
@@ -157,6 +158,21 @@ function build_wheel() {
     fi

     pushd ${TMPDIR} > /dev/null
10 tensorflow/tools/pip_package/setup.py
@@ -141,8 +141,8 @@ class InstallCommand(InstallCommandBase):

     def finalize_options(self):
       ret = InstallCommandBase.finalize_options(self)
-      self.install_headers = os.path.join(self.install_purelib,
-                                          'tensorflow', 'include')
+      self.install_headers = os.path.join(self.install_purelib, 'tensorflow_core',
+                                          'include')
       self.install_lib = self.install_platlib
       return ret
```

Commit f1ffa02 that was created 2 days ago before pull #29561

Fig. 2 Tensorflow pull #29561 and commit f1ffa02

itory. On top of the figure is TensorFlow pull #29561.[2] In this pull request, two files are changed. These two files are *build_pip_package.sh* and *setup.py*. At the bottom of the figure is TensorFlow commit f1ffa02,[3] which was created two days before pull #29561 was sent. From Fig. 2 we can observe that the two files changed in pull #29561 were previously modified in commit f1ffa02. Furthermore, both the code changes in pull #29561 and the code changes in commit f1ffa02 share many common key words such as "TMPDIR", "InstallCommandBase" and "install_purelib". Because pull #29561 and commit f1ffa02 have similar code changes, we consider the author of commit f1ffa02 to be appropriate since she/he has the knowledge to review pull #29561. Indeed, the author *mihaimaruseac* of commit f1ffa02 was assigned as a reviewer of pull #29561, and it is important to note that *mihaimaruseac* is not the original author or creator of the modified files. This example indicates that the history of code changes in the project, along with the code changes in the pull request can provide useful information to help find the appropriate reviewer.

In this paper, we propose a multi-instance-based deep neural network model to recommend reviewers for pull requests using code changes. A pull request contains one or more commits. Every commit is regarded as an instance. Given a pull request, our model extracts features from the title, the commit message and the code changes for every commit (instance) in the pull request. The features of different commits (instances) are then merged to predict the likelihood that a reviewer candidate is the appropriate reviewer.

We evaluated our model on 12 open-source projects and compared it to two baselines, including a state-of-the-art approach CoreDevRec (Jiang et al. 2015), and the majority classes approach. Experiment results show that our model outperforms the two baselines. Our model can help find the appropriate reviewers for 50.80%, 74.70% and 84.04% of the pull requests for the TensorFlow project when looking at the Top-1, Top-3 and Top-5 recommendations.

The main contributions of our work include:

1. We are the first to use code changes to recommend reviewers for pull requests automatically.
2. We propose a multi-instance-based deep neural network model to extract features from code changes for reviewer recommendation.
3. We enhance the model by incorporating file-path similarity and social relation of the requester into the loss function.
4. We evaluate our model on 12 open-source projects and show that it outperforms two state-of-the-art baselines.

The rest of this paper is structured as follows. Section 2 discusses the related work. Section 3 describes our approach in detail. Section 4 presents the evaluation of the model. Section 5 discusses threats to validity, before closing with a conclusion in Sect. 6.

2 Related work

In this section, we review literature related to supporting developers in handling pull requests. We start with existing studies related to recommending reviewers for pull requests, then we enumerate the works related to commenter recommendation, and finally, we reference studies on code review in general.

[2] https://github.com/tensorflow/tensorflow/pull/29561.

[3] https://github.com/tensorflow/tensorflow/commit/f1ffa02.

2.1 Studies on reviewer recommendation

de Lima Júior et al. (Soares et al. 2018) introduced 36 hand-crafted features (attributes in their paper) and proposed using different classification algorithms to assign a pull request to the appropriate reviewer. They divided these 36 features into three sets. Set A contains 14 features from their prior work (de Lima Júnior et al. 2015). Set B contains 11 features from the work of Jiang et al. (2015). Set C contains 11 novel features (Soares et al. 2018). They should read" They used these three sets of features to match the properties of the pull requests to the expertise of the review candidates. They performed an evaluation on 32 open-source projects and claimed that they approach using these 36 features outperformed the state-of-the-art. They also performed feature selection and reported that the user login name contributes most to the reviewer recommendation task.

Jiang et al. (2015) proposed an approach named CoreDevRec, which uses 11 hand-crafted features, for reviewer recommendation. Their approach gives higher weight to the activity level of the reviewer candidates. If a candidate has previously reviewed many pull requests, he/she is very likely to review a new pull request. Based on these 11 features, they used Support Vector Machine (SVM) to assign pull requests to different reviewer candidates. They performed evaluation on five software projects and reported that CoreDevRec helped find the appropriate reviewers for 72.3% of the pull requests when looking at the Top-1 recommendation.

Compared with these three studies, our study also uses the information from code changes to evaluate whether a candidate has the expertise to review a pull request.

2.2 Studies on commenter recommendation

Several studies were conducted to recommend developers to leave comments on pull requests. Yu et al. (2014b, a) proposed using several different approaches for commenter recommendation. In the classification-based approach, they convert a pull request into a weighted vector by using the Vector Space Model (VSM). They then use SVM to classify the pull request to the appropriate developer as the commenter. In the comment-network (CN)-based approach, a pull request is assigned to the developer that interacts frequently with the requester. In the Information Retrieval (IR)-based approach, they measure the cosine similarity between a new pull request and an old pull request. They then assign the new pull request to the developer that has commented on many similar pull requests before. Besides the above approaches, they also used a file location technique proposed by Thongtanunam et al. (2015) to compute the similarities between file paths in different pull requests. By using this technique, a new pull request will be assigned to the developer that has similar experience to review the files changed in the pull request. Finally, they performed evaluation on 84 open-source projects and claimed that the IR+CN combination helped obtain the best result.

Rahman et al. (2016) converted the changed code in a pull request into a bag of tokens. They then measure the similarity between two pull requests by computing the cosine similarity between their bags of tokens. If a developer has commented on many pull requests that have large similarities with a new pull request, this developer is likely to have the knowledge to leave useful comments on the new pull request and hence is recommended as an appropriate commenter.

Jiang et al. (2017) proposed four features. The first type of features measures the activity level of developers within a time window. The second type of features measures the similarity between the titles of two pull requests. The third type of features measures the similarity between the changed files in two pull requests. The fourth type of features measures the social relations among developers. They used these four features to compute an expertise score for every developer and ranked all the developers based on their expertise scores. A higher position in the ranked list indicates that the developer is more likely to be a helpful commenter. They evaluated their approach using 19,543 pull requests and reported that the first type of features contributed most to the commenter recommendation task.

Yang et al. (2018) proposed a two-layer approach to find developers to comment on pull requests. In the first layer, they use a hybrid method to recommend developers as commenters for pull requests. In the second layer, they further specify whether the developers will technically or managerially participate in the discussion process. They performed evaluation on two open-source projects and reported that their approach outperformed previous work.

In contrast with these studies, our study recommends reviewers instead of commenters for pull requests. Thus, our study complements these existing studies to complete the cycle of discussing and deciding about a given pull request.

2.3 Studies on code review

Lee et al. (2013) proposed a graph-based technique to recommend reviewers for patches. Balachandran (2013) introduced Review Bot to recommend reviewers that had frequently worked on related code sections. Thongtanunam et al. (2015) designed a file location-based technique to recommend reviewers that had worked on similar file paths. Xia et al. (2015) proposed a hybrid approach that utilizes text mining and file location-based techniques to recommend code reviewer. These studies focus on traditional open source software environment, while our study focuses on pull-based development environments.

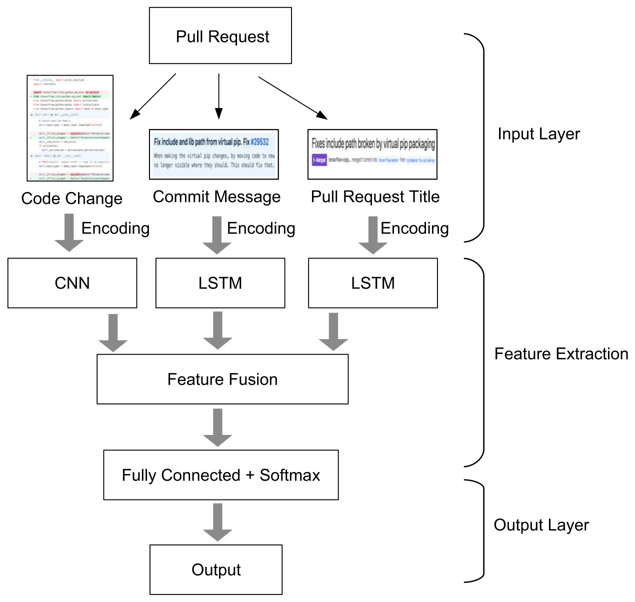


Fig. 3 The framework of the proposed approach

3 Approach

In this section, we first formulate the reviewer recommendation problem and introduce the overall framework of our approach. We then describe each part of the framework in more detail. Finally, we introduce a novel technique to enhance the loss function for the reviewer recommendation task.

3.1 The framework

Given a pull request, the goal of our approach is to predict which reviewer candidates are appropriate to review the pull request. Let $\mathcal{R} = \{r_1, r_2, ..., r_{N_1}\}$ denote a set of reviewer candidates to a software project and $\mathcal{X} = \{x_1, x_2, ..., x_{N_2}\}$ denote the collection of pull requests received by the software development team, where N_1 and N_2 are the total number of reviewer candidates and the number of pull requests. We formulate the reviewer recommendation problem as a learning task and construct a prediction function $f : \mathcal{X} \mapsto \mathcal{Y}$, where $y_{ij} \in \mathcal{Y} = \{+1, -1\}$ indicates whether a reviewer candidate r_j is an appropriate reviewer for pull request x_i. The prediction function can be learned by minimizing the following objective function:

$$\min_f \sum_{i,j} \mathcal{L}(f(x_i), y_{ij}) + \lambda \Omega(f) \qquad (1)$$

where $\mathcal{L}(.)$ is the empirical loss, $\Omega(f)$ is the regularization term to prevent over fitting, and λ is the trade-off between $\mathcal{L}(.)$ and $\Omega(f)$.

The framework of our approach is shown in Fig. 3. The framework contains an input layer, a feature extraction layer and an output layer. Each pull request contains a title and one or more commits. Each commit contains a commit message and one or more changed files. In our approach, each commit is regarded as an instance. In the input layer, the pull request title, the commit message and the code change for each instance are encoded into vectors of real numbers. In the feature extraction layer, the encoded vectors from the previous layer are fed into different neural networks (CNN and LSTM-network). The neural networks generate the middle-level features for the pull request title, the commit message and the code change. These middle-level features are further fused to learn a unified feature representation for each instance. In the output layer, the unified feature representation for each instance is fed into a Softmax layer to produce the categorical (probability) distribution over all the possible categories (each reviewer candidate is regarded as a category). Finally, a max pooling layer is used to combine the output of multiple instances to produce a prediction for each pull request.

3.2 Parsing a pull request

A pull request contains a title and one or more commits. Each commit contains a message and one or more changed files. The code change of a commit can be obtained from https://github.com/user/project/commit/commitID. In the input layer, we encode the pull request title, the commit message and the code change of each commit into vectors of real numbers.

For a pull request title or a commit message, we remove punctuation and numbers. According to a study (Bissyandé et al. 2013), removing stop words could lead to decreasing classification accuracy. So we retain stop words such as "should" and "not". After that, we split the text into a bag-of-words by whitespace. Then we use PorterStemmer (Willett 2006) to convert each word into its root form. For example, "programs" and "programming" are converted to "program".

For the code change of a given commit, we follow (Hoang et al. 2019) to parse it into a set of added and deleted lines. Comments and blank lines are ignored. Every numerical number is replaced with a $< num >$ token. Rare words appearing less than three times are replaced with $< unk >$ tokens. Unknown words that appear in the test data but are not seen in training data are also replaced with $< unk >$ tokens. An $< added >$ token is added at the beginning of an added line. A $< deleted >$ token is added at the beginning of a deleted line. We then parse every added or deleted line into a bag-of-words as well.

Next, for the pull request title, the commit message and the code change, we represent every word as a d-dimensional vector of real numbers called word embeddings that captures some contextual semantic meanings (Levy and Goldberg 2014). The word embeddings used in this study were learned with size of 100 by using the Mikolov's Skip-gram model (Le and Mikolov 2014) on the Wikipedia data dumps.[4] After that, a bag-of-words with size n can be represented as vectors $X \in \mathbb{R}^{nx100}$. Let X_i^t denote the encoded vectors of the title of the pull request x_i, X_{ij}^m denote the encoded vectors of the commit message of the jth commit in x_i, and X_{ij}^c denote the encoded vectors of the code change in the jth commit. The encoded data X_i^t, X_{ij}^m and X_{ij}^c are passed to the feature extraction layer.

3.3 Feature extraction layer

As pull request titles and commit messages are written in natural language, code change is written in programming language, they have different structures. So we use CNN for code and LSTM-network for text in the feature extraction layer to learn features. CNN has been shown to perform well in extracting structural features from code (Huo et al. 2016). LSTM-network has shown to be effective in extracting semantic features from text (Ye et al. 2018).

3.3.1 LSTM-network for text

The LSTM-network used in this study is similar to Ye et al. (2018). Given the encoded vectors of a text $X \in \mathbb{R}^{nx100}$, $X = (\mathbf{w}_1, \mathbf{w}_2, ..., \mathbf{w}_n)$, where $\mathbf{w}_i \in \mathbb{R}^{100}$ is the embedding of word token w_i and n is the number of word tokens, the LSTM-network takes X as a time series input. At each time step, an embedding $\mathbf{w}_i$ is fed into the LSTM network. The output of the LSTM unit from the last time step $\mathbf{h}_n \in \mathbb{R}^m$ is used as the feature vector of X, where m is the number of hidden units (states) in the memory cell.

3.3.2 CNN for code

The CNN used in this study is similar to Huo et al. (2016); Hoang et al. (2019). As shown on top of Fig. 4, a line of code is represented as a matrix of real numbers $M \in \mathbb{R}^{nx100}$, where n is the number of word tokens and 100 is the size of the embeddings. In the first convolutional layer, multiple filters with different size are used to perform convolution operations on M. A filter with size $m \times 100$ will generate a feature map of size $(n - m + 1) \times 1$. After the convolutional layer, a max pooling layer is used to convert multiple feature maps to one feature map of size $k \times 1$, where k is the number of filters. So, every line of code is represented as a feature map of size $k \times 1$ after the first convolutional and max pooling layer.

As shown at the bottom of Fig. 4, after the first convolutional and max pooling layer, a code change with l lines of code is represented as a feature map of size $l \times k_1$, where k_1 is the number of filters in the first convolutional layer. The second convolutional and max pooling layer, which is similar to the first convolutional and max pooling layer, is used to further convert the feature map into a feature vector of size k_2, where k_2 is the number of filters in the second convolutional layer.

3.3.3 Feature fusion

As discussed in Sect. 3.1, a pull request contains one or more commits. In this study, every commit is regarded as an instance. Given a pull request x, which contains n commits $(c_1, c_2, ..., c_n)$, we first use a LSTM-network to learn a feature vector $\mathbf{z}^t$ for its title. Then for every commit c_j, we use a LSTM-network to learn a feature vector $\mathbf{z}_j^m$ for its commit message and use CNN to learn a feature vector $\mathbf{z}_j^c$ for its code change. As shown in the left hand side of Fig. 5, we then concatenate these three parts as follows:

$$\mathbf{z}_j^h = \mathbf{z}_j^m \oplus \mathbf{z}_j^c \oplus \mathbf{z}^t \tag{2}$$

where $\oplus$ is the concatenation operator.

The new vector $\mathbf{z}_j^h$ is the unified feature representation for the j^{th} commit c_j in the pull request x. The unified feature vector is further fed into a fully connected layer to generate a new vector $\mathbf{h}_j$ as follows:

$$\mathbf{h}_j = \sigma(\mathbf{w}_h \cdot \mathbf{z}_j^h + \mathbf{b}_h) \tag{3}$$

where $\mathbf{w}_h$ is the weighting matrix, $\mathbf{b}_h$ is the bias and σ is the rectifier (ReLU) (Nair and Hinton 2010) activation function.

To obtain unbiased feature representation for every commit, the weight parameters of CNN are shared for all commits. Similarly, the weights for LSTM networks and the weights for the fully connected layers are also shared for all commits.

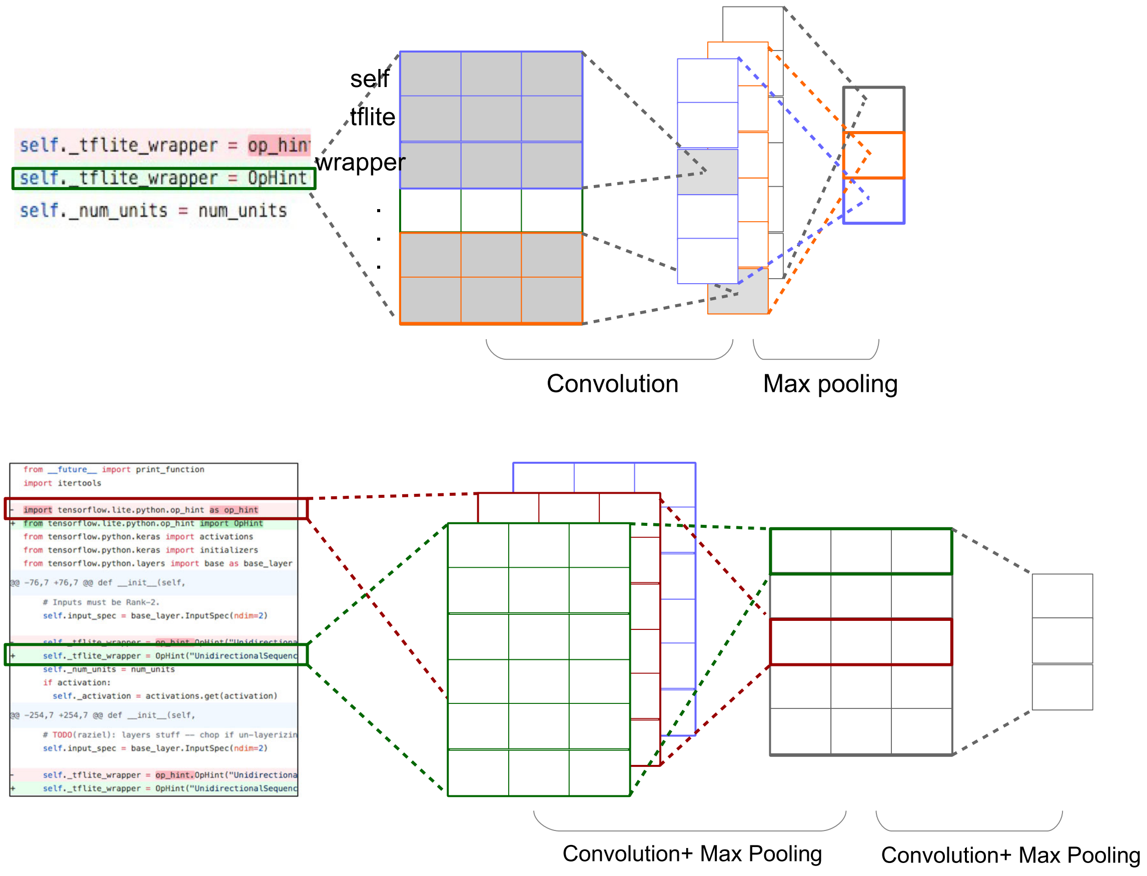
Finally, the feature extraction layer outputs a feature vector $\mathbf{h}_j$ for every commit c_j in the pull request x.

3.4 Multi-instance-based output layer

Figure 5 illustrates the multi-instance-based prediction process. Following a previous work (Li et al. 2019), we first make a prediction for every commit (instance). We then combine the results of multiple instances to produce the final prediction.

As discussed in the previous section, for a pull request with n commits, we obtained a set of feature vectors for these commits $(\mathbf{h}_1, \mathbf{h}_2, ..., \mathbf{h}_n)$. A feature vector $\mathbf{h}_j$ of the

[4] https://dumps.wikimedia.org/enwiki/.


```
self._tflite_wrapper = op_hin
self._tflite_wrapper = OpHint
self._num_units = num_units
```
```
  from __future__ import print_function
  import itertools

- import tensorflow.lite.python.op_hint as op_hint
+ from tensorflow.lite.python.op_hint import OpHint
  from tensorflow.python.keras import activations
  from tensorflow.python.keras import initializers
  from tensorflow.python.layers import base as base_layer
@@ -76,7 +76,7 @@ def __init__(self,
      # Inputs must be Rank-2.
      self.input_spec = base_layer.InputSpec(ndim=2)

-     self._tflite_wrapper = op_hint.OpHint("Unidirectiona
+     self._tflite_wrapper = OpHint("UnidirectionalSequenc
      self._num_units = num_units
      if activation:
        self._activation = activations.get(activation)
@@ -254,7 +254,7 @@ def __init__(self,
      # TODO(raziel): layers stuff -- chop if un-layerizin
      self.input_spec = base_layer.InputSpec(ndim=2)

-     self._tflite_wrapper = op_hint.OpHint("Unidirectiona
+     self._tflite_wrapper = OpHint("UnidirectionalSequenc
```

Fig. 4 CNN for code

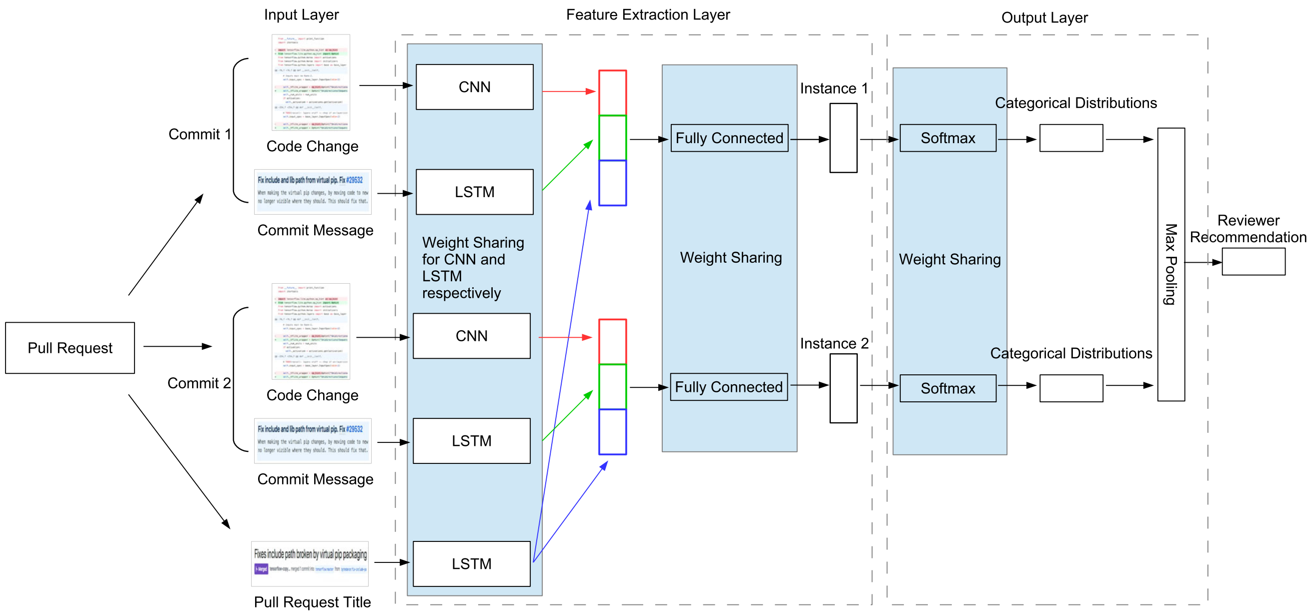


Fig. 5 Multi-instance-based feature learning and prediction

jth commit (instance) is fed into a Softmax layer to produce the categorical (probability) distribution.

$$p_{ij} = P(x \in i|\mathbf{h}_j) = \frac{exp(\mathbf{v}_i{}^T\mathbf{h}_j)}{\sum_{k=1}^{N} exp(\mathbf{v}_k{}^T\mathbf{h}_j)}, \quad i \in [1, N] \quad (4)$$

Given the feature vector $\mathbf{h}_j$ of the jth commit, the probability of the ith category for pull request x is denoted in Eq. 4, where $\mathbf{v}$ is the weighting vector and N is the number of categories..

Given a pull request with n commits (instances), the output probability of the ith category for every commit (instance) $\mathbf{p}_i = (p_{i1}, p_{i2}, ..., p_{in})$ is generated. Then, a max pooling layer is applied to obtain the final probability of the ith category $\hat{p}_i = max\{\mathbf{p}_i\}$ for pull request x.

Then, the final categorical distribution $(\hat{p}_1, \hat{p}_2, ..., \hat{p}_N)$ for pull request x is obtained, where N is the total number of categories (the numbe of reviewer candidates).

Finally, the top N reviewer candidates with the largest probability values are recommended as the potential reviewers for the pull request.

3.5 Loss function

The CNN and LSTM-network learn semantic and structural features from code and text in a pull request. But they do not contain information about the file paths and the social relation of the requester (the sender of the pull request), which are important indicators about who should be the appropriate reviewers (Thongtanunam et al. 2015; de Lima Júnior et al. 2018). Therefore, in this study, we improve the model by adding file-path similarity and social relation of the requester as two penalty terms to the cost function.

3.5.1 File-path similarity

A pull request contains changed files. According to Thongtanunam et al. (2015), if a reviewer candidate has previously modified some files, and these files are changed in a pull request, then this candidate is likely to have the experience to reviews the changes. Similarly, if a reviewer candidate has modified some files, and these files are located at similar file system paths with the changed files in a pull request, then this reviewer candidate is also likely to have the knowledge to review the changes (Thongtanunam et al. 2015).

As discussed above, different files may have similar functionality if they locate at the same directory (Thongtanunam et al. 2015). For two files f_k and f_l, we split their file paths into components by using the slash character as a delimiter. Every component is a word. Then the path similarity $filePathSim(f_k, f_l)$ between these two files can be computed in Eq. 5, where $LCP(f_k, f_l)$ is the longest common prefix of f_k and f_l, and $Len(f_k)$ is the number of components in f_k. LCP calculates the number of common components that appear in both file paths from the beginning to the last.

$$filePathSim(f_k, f_l) = \frac{LCP(f_k, f_l)}{\max(Len(f_k), Len(f_l))} \quad (5)$$

$$P_{ij} = \begin{cases} \sum_{f_k \in F_i} \sum_{f_l \in F_j} \frac{filePathSim(f_k, f_l)}{|F_i| \times |F_j|} & \text{if} |F_i| > 0 \\ 0 & \text{if} |F_j| = 0 \end{cases} \quad (6)$$

We let F_i denotes the set of files changed in sample (pull request) i, F_j denotes the set of files that are modified by a reviewer candidate j recently (i.e., within one week), we then compute the file path similarity P_{ij} using Eq. 6. If reviewer candidate j did not modify any files recently (i.e., $|F_j| = 0$), P_{ij} is set to 0. Otherwise, P_{ij} is the sum of path similarity values between files changed in the pull request and files modified by the developer recently normalized by $|F_i| \times |F_j|$.

3.5.2 Social relation of the requester

According to de Lima Júnior et al. (2018), a new pull request may be reviewed by a reviewer, if this reviewer has frequently reviewed pull requests sent by the same requester before.

When we receive a new pull request, we let i denotes its requester, we let j denotes a reviewer candidate and let $pulls\_reviewed(i, j)$ denotes the set of pull requests that were sent i and reviewed by j. As shown in Eq. 7, we measure the social relation S_{ij} of the requester i by computing the total number of such pull requests.

$$S_{ij} = |pulls\_reviewed(i, j)| \quad (7)$$

3.5.3 Loss function

Let Θ_{CNN} denote the parameters of CNN, Θ_{LSTM} denote the parameters of the LSTM-network, W_c denote the parameters of the fully-connected layer, and W_s denote paramters of the Softmax Layer. Let $\Theta = \{\Theta_{CNN}, \Theta_{LSTM}\}$ and $W = \{W_c, W_s\}$, then the model is trained to minimize the following mean cross-entropy error loss (Gu et al. 2016):

$$\begin{aligned} \mathcal{L}(\Theta, W) &= \frac{1}{N} \sum_{i}^{N} \sum_{j}^{T} cost_{ij} \\ cost_{ij} &= -y_{ij} log(\hat{p}_{ij}) \end{aligned} \quad (8)$$

where N is the number of training samples in a batch, T is the number of categories, $cost_{ij}$ is the cost function, y_{ij} is the true value of category j of sample i, and $\hat{p}_{ij}$ is the output probability of category j of sample i.

After adding file-path similarity and social relation of the requester as two penalty terms, the new cost function

Table 1 Benchmark datasets

Project	Language	Time range	Pull requests selected	Majority classes			Commits	Reviewers
				Top-1(%)	Top-3(%)	Top-5(%)		
tensorflow	C++	2015-11–2019-03	6,335	13.12	22.92	33.11	51,974	370
opencv	C++	2012-07–2019-03	1,603	69.49	79.93	87.25	26,395	70
bitcoin	C++	2010-12–2019-03	2,652	22.74	29.46	41.22	20,117	233
electron	C++	2013-06–2019-03	3,279	33.88	48.62	64.12	21,631	128
swift	C++	2015-11–2019-03	6,843	18.35	28.57	35.86	85,615	228
node	JavaScript	2015-11–2019-03	11,039	14.63	32.11	43.08	26,762	447
react	JavaScript	2013-06–2019-03	2,292	35.56	47.27	63.38	10,856	261
keras	Python	2015-03–2019-03	1,416	69.50	82.17	87.03	5,104	139
pandas	Python	2015-03–2019-03	3,774	41.94	67.93	82.17	19,218	138
scikit-learn	Python	2010-09–2019-03	1,973	29.78	50.39	61.87	23,898	121
jekyll	Ruby	2010-11–2019-03	918	57.84	70.25	86.22	10,448	50
rails	Ruby	2010-09 – 2019-03	1,862	30.56	36.69	50.15	73,110	261

is defined as follows:

$$cost_{ij} = -y_{ij}\log(\hat{p}_{ij}) - w_1 p_{ij} - w_2 s_{ij} \tag{9}$$

where w_1 and w_2 are tuned in the training phase, $p_{ij} = P_{ij}$, and s_{ij} is the scaled value of S_{ij}.

$$s_{ij} = \begin{cases} 0 & \text{if } S_{ij} < S.\min \\ \dfrac{S_{ij} - S.\min}{S.\max - S.\min} & \text{if } S.\min \le S_{ij} \le S.\max \\ 1 & \text{if } S_{ij} > S.\max \end{cases} \tag{10}$$

4 Experiments

In this section, we introduce the data used in this study, the evaluation metric, the baselines used for comparison purposes, the research questions and the results.

4.1 Data collection

In this study, we collected a dataset from twelve open-source projects, which are the most forked projects on GitHub, using the GitHub API[5]. These projects have 3235 watches, 56,629 stars and 19,989 forks on average. Each of these projects has more than 1,000 pull requests and more than 50 reviewer candidates. A reviewer candidate is a person who has reviewed one or more pull requests before. As previously shown in Fig. 2, a reviewer is the person who reviews the pull request and the person who may approve or reject the changes, a reviewer may not be the person who performs the merge operation. We used GitHub API to obtain the review information and the reviewer information. If a pull request does not contain review information, we ignore it. A requester who submits the pull request tends to have other developers review it. If a pull request is self-reviewed (reviewed by the requester), we also ignore it since it does not represent the normal review behavior. Finally, we collect 43,986 pull requests totally for experiments.

Table 1 shows the statistics of these twelve projects. Column "Project" shows the project name. Column "Top-1" under "Majority Classes" shows the ratio of pull requests in each project, and which were subject of review by the most active developer. Take TensorFlow for example, about 13.12% of the pull requests are reviewed by the most active reviewer "gunan". Column "Top-3" under "Majority Classes" shows the percentage of pull requests that are reviewed by the three most active reviewers. Take TensorFlow again for example, about 22.92% of the pull requests are reviewed by the three most active reviewers "gunan", "caisq" and "martinwicke". Similarly, column "Top-5" shows the percentage of pull requests that are reviewed by the five most active reviewers. Column "Reviewers" shows the number of reviewer candidates of each project.

For each project, all the pull requests are sorted chronologically based on their create timestamp. The sorted pull requests are split into K equally sized bins, where $fold_1$ contains the oldest pull requests and $fold_K$ contains the most recent pull requests. We use $fold_k$ for training, use $fold_{k+1}$ for validation (to validate whether the model is converged or not), and use $fold_{k+2}$ for testing for $k \in [1, K-2]$. We then pool together the results from all test bins to compute the overall system performance. In this study, by tuning, we found that the model achieved best performance when the training set size is 500. So the number of bins K for each project can be computed by $K = (\text{\# pull requests}/500)$.

[5] https://developer.github.com/v3/.

4.2 Evaluation metric

As discussed in Sect. 3.4, we compute a probability value for every reviewer candidate and rank all the candidates based on their probability values. The higher position in the ranked list the larger chance that the candidate is the appropriate reviewer to review the pull request. The ranking result is compared to the best ranking where the actual reviewers are listed at the top. We then evaluate the system performance using the following evaluation metrics:

- *Accuracy@k* is defined as the percentage of pull requests for which we make at least one correct recommendation when looking at the top k recommendations.
- *Mean Average Precision (MAP)* (Manning et al. 2008) is the mean of the average precision over all queries.
- *Mean Reciprocal Rank (MRR)* (Voorhees 1999) is defined as the average of the reciprocal ranks of the first positive instance. It is a metric measuring the ranking performance on the first recommendation.

4.3 Training and testing procedure

We used the Adam (adaptive moment estimation) optimizer (Kingma and Ba 2014) to train the model. We first split the training set into small *batches*. We then compute the objective function gradient using mini-batch set to update the model. The advantage of using batches is that we can train the model efficiently, in order to help with avoiding any local minima, and obtaining higher convergence (Goodfellow et al. 2016).

A cycle is one forward pass, followed by a backward pass, in which, all training data is seen, and labeled an *epoch*. We train our model over a maximum of 500 epochs, along with a predefined stopping criterion. Whenever we observe a specific pattern of continuous performance degradation on the testing set, we consider the model has converged. We report the testing results when the model achieves its best performance in terms of MAP on the validation set.

4.4 Research questions and results

In this subsection, we present our research questions and the experimental results.

RQ1: How do different model settings affect the system performance?

To address this question, we perform experiments on four projects chosen from four different programming languages. These four projects are TensorFlow (written in C++), node.js (written in JavaScript), pandas (written in Python) and rails (written in Ruby).

First, for CNN, we analyze the impact of the number of filters. We set the filter window size m to be 3. We set the number of hidden units in the LSTM-network to be 64. We then perform experiments with various numbers of filters in both convolutional layers. Table 2 shows the result. When the number of filters is set to 64 for both convolutional layers, we achieve the best MAP across four projects.

Next, for CNN, we analyze the impact of the filter window size m. We set the number of filters for both convolutional layers to be 64. We set the number of hidden units in the LSTM-network to be 64. We try different filter window size m on four projects. Table 3 shows the result. When the filter window size m is set to 3, we obtain the best MAP for all four projects.

Therefore, according to the CNN structure described in Sect. 3.3, the first convolutional layer uses 64 filters with size 3×100, where 3 is the filter window size and 100 is the word embedding size. The second convolutional layer uses sixty-four filters with size 3×64, where 3 is the filter window size and 64 is the number of filters in the first layer.

For the LSTM network, we analyze the impact of the number of hidden units in the memory cell. Table 4 shows the result. When the number of hidden units is set to 64, we obtain the best MAP for four projects. Therefore, we choose the number of hidden units to be 64 in the following experiments.

Table 2 MAP comparison: different numbers of filters in CNN

Project	Number of Filters			
	16	32	64	128
tensorflow	0.637	0.640	**0.643**	0.641
node	0.311	0.312	**0.313**	0.313
pandas	0.850	0.853	**0.854**	0.851
rails	0.699	0.701	**0.705**	0.703

Table 3 MAP comparison: different filter window size m in CNN

Project	Filter window size m			
	2	3	4	5
tensorflow	0.643	**0.643**	0.641	0.638
node	0.307	**0.313**	0.311	0.312
pandas	0.847	**0.854**	0.853	0.852
rails	0.702	**0.705**	0.704	0.701

Table 4 MAP Comparison: different numbers of hidden units in LSTM

Project	Number of Hidden Units			
	16	32	64	128
tensorflow	0.639	0.643	**0.643**	0.641
node	0.311	0.312	**0.313**	0.313
pandas	0.850	0.852	**0.854**	0.854
rails	0.702	0.702	**0.705**	0.701

Table 5 Accuracies per project

Project	Baseline			CoreDevRec					Ours				
				Accuracy					Accuracy				
	Top-1 (%)	Top-3 (%)	Top-5 (%)	Top-1 (%)	Top-3 (%)	Top-5 (%)	MAP	MRR	Top-1 (%)	Top-3 (%)	Top-5 (%)	MAP	MRR
tensorflow	13.12	22.92	33.11	43.71	61.90	70.43	0.507	0.585	**50.80**	**74.70**	**84.04**	**0.643**	**0.652**
opencv	69.49	79.93	87.25	81.51	90.99	92.73	0.870	0.878	**81.85**	**94.82**	**98.25**	**0.871**	**0.883**
bitcoin	22.74	29.46	41.22	31.24	63.57	73.79	0.401	0.520	**32.88**	**65.69**	**77.60**	**0.442**	**0.524**
electron	33.88	48.62	64.12	41.13	70.44	77.97	0.521	0.587	**43.47**	**70.49**	**79.93**	**0.535**	**0.591**
swift	18.35	28.57	35.86	34.20	51.50	60.70	0.357	0.513	**41.92**	**68.63**	**80.21**	**0.553**	**0.583**
node	14.63	32.11	43.08	41.10	47.60	61.40	0.309	0.528	**43.38**	**58.40**	**62.08**	**0.313**	**0.535**
react	35.56	47.27	63.38	34.30	62.70	73.10	0.448	0.482	**35.76**	**63.66**	**74.17**	**0.451**	**0.496**
keras	69.50	82.17	87.03	62.71	95.33	97.19	0.813	0.862	**89.90**	**98.23**	**99.79**	**0.933**	**0.941**
pandas	41.94	67.93	82.17	67.30	88.10	94.90	0.753	0.794	**82.40**	**95.39**	**98.22**	**0.854**	**0.892**
scikit-learn	29.78	50.39	61.87	65.90	83.60	88.10	0.688	0.771	**75.90**	**91.58**	**95.84**	**0.782**	**0.844**
jekyll	57.84	70.25	86.22	70.10	87.90	95.60	0.753	0.816	**83.10**	**97.50**	**99.20**	**0.849**	**0.907**
rails	30.56	36.69	50.15	46.30	70.40	81.80	0.593	0.603	**62.03**	**82.01**	**89.42**	**0.705**	**0.738**

RQ2: How effective is our approach compared to the baselines?

In order to address this question, we run experiments on twelve projects and compare the results of our approach with the results of two baselines.

1. The Majority Classes showing in Table 1 are used as the first baseline. It shows the percentage of pull requests reviewed by the top-k ($k = 1, 3, 5$) most active candidates. In other words, the first baseline simply assigns the most active candidates as the reviewers.
2. A state-of-the-art approach called CoreDevRec (Jiang et al. 2015) is used as the second baseline. CoreDevRec uses 11 attributes to measure the social relations, file location and activity level of candidates. Based on these 11 attributes, it classifies a pull request to the appropriate candidates for review.

Table 5 shows the results. As shown in the table, our approach outperforms the two baselines for all 12 projects in terms of Accuracy, MAP and MRR.

For TensorFlow, Bitcoin, Swift and Node.js, the Top-1 majority reviewer reviewed less than 20% of the pull requests, the Top-3 majority reviewers reviewed about 30% of the pull requests, the Top-5 majority reviewers reviewed less than 50% of the pull requests. In these projects, there are multiple reviewers that take charge of the review activity. For these projects, our approach achieves Accuracy@1 higher than 30%, Accuracy@3 higher than 50%, and Accuracy@5 higher than 60%. Take TensorFlow for example, if we recommend the most active developer as the reviewer, we can only make the correct recommendation for 13.12% of the pull request. However, if we use our approach to recommend a reviewer, we can find the appropriate reviewer for 50.80% of the pull requests. If we recommend the five most active developers, we can find the actual reviewer for 33.11% of the pull request. But if we use our approach to recommend five reviewers, we can help find the appropriate reviewer for 84.04% of the pull requests.

For Electron, React, Pandas, Scikit-learn and Rails, the Top-1 majority class ranges from 30% to 42%. The Top-5 majority class ranges from 50% to 82%. In each of these projects, there is a small group of reviewers that reviewed most of the pull requests. Take electron for example, if we recommend the five most active developers, we can find the appropriate reviewer for 64.12% of the pull requests. The results of the first baseline in this group of projects are better than the results in the previous group. However, if we use our approach to recommend five candidates, we can find the appropriate reviewer for 79.93% of the pull requests. Our approach still outperforms both baselines in this group of projects.

Table 6 MAP comparison: contribution of different components

Model	tensorflow	node	pandas	rails
title+message	0.443	0.241	0.641	0.532
code	0.611	0.302	0.837	0.688
title+message+code	**0.643**	**0.313**	**0.854**	**0.705**

For OpenCV, Keras and Jekyll, the Top-1 majority reviewer reviewed more than 57% of the pull requests. In each of these projects, there is one reviewer that dominates the review activity. Therefore, the first baseline performs quite well on this group of projects. Take OpenCV for example, if we recommend the most active developer as the reviewer, we can help 69.49% of the pull requests. Even the baseline performs well, our approach still can achieve better performance. If we use our approach to recommend one candidate as the reviewer, we can make the correct recommendation for 81.85% of the OpenCV pull requests.

Overall, our approach outperforms the baseline and CoreDevRec on these twelve projects.

RQ3: Does our approach benefit from both the text and the code change?

Our approach uses both the text (pull request title and commit message) and the code change. We want to analyze the contribution of different components. To address this research question, we created two variants of our model that one uses only the pull request title plus the commit message and the other uses only the code change. We then ran experiments with different models on four projects chosen from different programming languages. Table 6 shows the result in terms of MAP. As shown in the table, when we use only the pull request title plus the commit messages, we obtain the worst performance. When we use only the code change, we obtain better performance. When we use pull request title, commit messages and code change, we obtain the best result for four projects. It suggests that both the text and the code change contribute to the system performance. It also indicates that the code change is more helpful than the text.

RQ4: Does our approach benefit from the multi-instance setting?

A pull request contains one or more commits. As discussed in Sect. 3.4, in our approach, every commit in a pull request is regarded as an instance. we first make a prediction for every instance and then combine the results of all instances to produce a final prediction for a pull request. In this research question, we want to evaluate the effectiveness of applying this multi-instance setting on the system performance. To answer this question, we created a traditional single-instance model. This single-instance model concatenates the feature vectors of multiple commits into one vector, which serves as the input to the softmax layer for producing the output probabilities. We ran experiments with this single-instance model and the multi-instance model on four projects chosen from different programming languages.

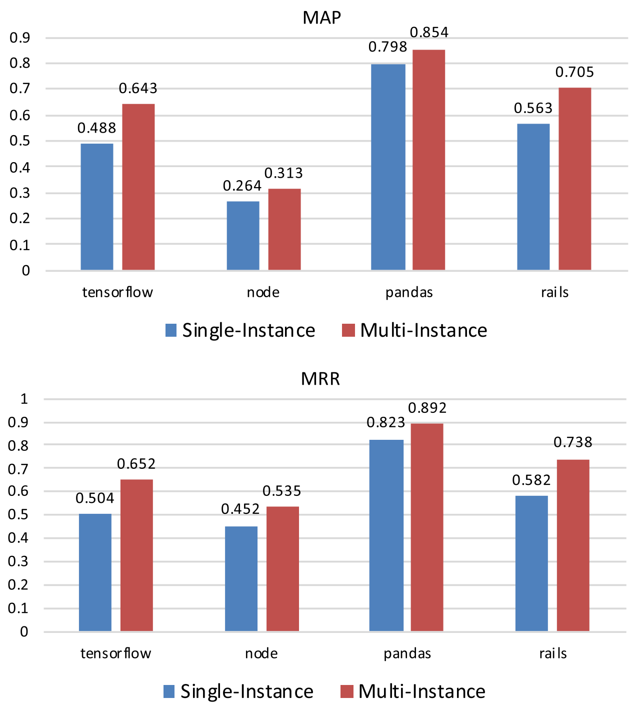


Fig. 6 Single instance vs. multiple instances

Figure 6 shows the results of comparing the single-instance model and the multi-instance model. The results show that the multi-instance model achieves better performance in terms of MAP and MRR than the single-instance model for four projects. This observation indicates that the multi-instance setting is effective in the reviewer recommendation task.

RQ5: Does our approach benefit from the enhanced loss function?

As discussed in Sect. 3.5, we incorporate the file-path similarity and the social relation of requester into the loss function. In this research question, we want to know whether the enhanced loss function helps improve the ranking performance compared with the traditional cross-entropy loss. To answer this question, we create a model trained by minimizing the traditional cross-entropy loss. We conducted experiments with two models on four projects chosen from different programming languages.

Figure 7 shows the results of comparing the model trained by minimizing the cross-entropy loss and the model trained by minimizing the enhanced loss. The results show that the second model achieves better performance in terms of MAP and MRR than the first model for four projects. This observation indicates that the enhanced loss function can help

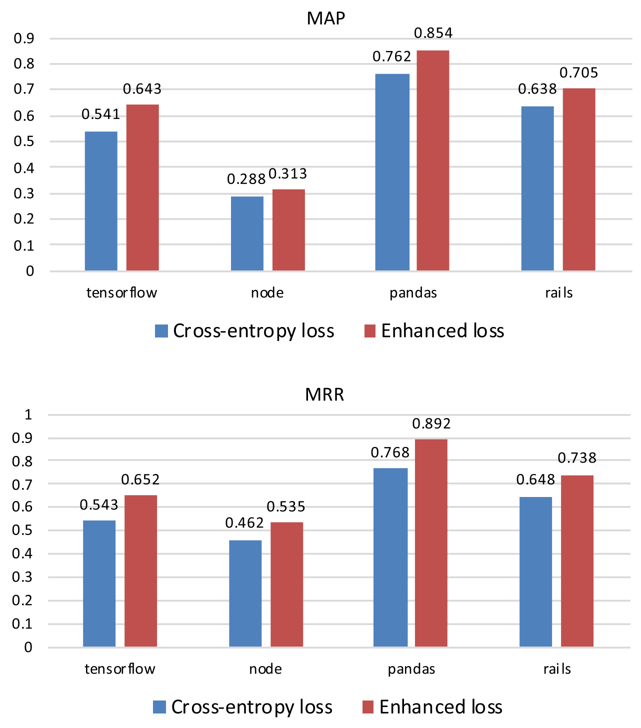


Fig. 7 Cross-entropy loss vs. enhanced loss

improve the ranking performance for the reviewer recommendation task.

5 Threats to validity

We collected reviewer information from GitHub by using the GitHub API. For a pull request that was assigned to multiple reviewers, only one reviewer actually approved or rejected the pull request. However, we used all the assigned reviewers as the ground truth. Even though our experiment results may be over optimistic, we considered all these assigned reviewers to have the knowledge to review the pull request.

We conducted experiments on twelve open-source projects. Our results may not be generalized to all software projects such as industrial projects. To mitigate this problem, we chose different types of projects written in different programming languages and with different sizes. The chosen projects are the most forked and most active projects on GitHub. Furthermore, many projects may contain various developers who have the ability to review almost any type of change, and others may require certain developers to handle specialized types of code changes, including those who require a high level of permissions. In our experiments, we did not handle such cases, because such information is not available in issue trackers.

We compared our approach to a state-of-the-art approach CoreDevRec. Because their source code is not publicly available, we re-implemented CoreDevRec by following the steps in their paper. But our implementation might not have all the details as their approach. Besides, we ran experiments on our dataset instead of theirs. So the results were not exactly the same as their results.

Our model has several hyperparameters such as the training set size, the number of filters in CNN, the filter window size in CNN and the number of hidden units in the LSTM-network. Ideally, we should try all the combinations of hyperparameters. But this is very time consuming. So when we tuned each hyperparameter, we set other hyperparameters to fixed values and tried different combinations of them.

6 Conclusion and future work

In this work, we formulated reviewer recommendation as a multi-instance learning task and proposed a deep neural network model to recommend reviewers for pull requests automatically. We enhanced the model by incorporating file-path similarity and social relation of the requester into the loss function. When a pull request is received, the model extracts features from the pull request title, the commit message and the code change for every commit. Features of different commits are then combined to predict how likely a candidate is the appropriate reviewer to review the pull request.

We evaluated the model on twelve open-source software projects. The evaluation results show that our approach outperforms two baselines including a state-of-the-art approach CoreDevRec. Using project TensorFlow as an example, our approach can help find the appropriate reviewer for 50.80%, 74.70% and 84.04% of the pull requests when considering the Top-1, the Top-3 and the Top-5 recommendations. Experiment results also show that the information from code changes contributes most to the reviewer recommendation task.

In future work, we will challenge our approach with more software projects including industrial projects. We will try more combinations of hyperparameters to find out the optimal model setting. We plan to explore alternative methods like *sent2vec* (Pagliardini et al. 2017) to convert pull request title and commit message into vector representations. Besides, we plan to implement our model into a tool and publish it so that everyone can use it. We also plan to perform a developer study to evaluate the efficacy and usability of the tool.

Compliance with ethical standards

Conflict of interest Xin Ye declares that he has no conflict of interest. Yongjie Zheng declares that he has no conflict of interest. Wajdi

Mohammed Aljedaani declares that he has no conflict of interest. Mohamed Wiem Mkaouer declares that he has no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

References

Balachandran V (2013) Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In: 2013 35th international conference on software engineering (ICSE), IEEE, pp 931–940

Bissyandé TF, Lo D, Jiang L, Réveillere L, Klein J, Le Traon Y (2013) Got issues? Who cares about it? A large scale investigation of issue trackers from github. In: 2013 IEEE 24th international symposium on software reliability engineering (ISSRE), IEEE, pp 188–197

Goodfellow I, Bengio Y, Courville A, Bengio Y (2016) Deep learning, vol 1. MIT press Cambridge

Gousios G, Pinzger M, Deursen Av (2014) An exploratory study of the pull-based software development model. In: Proceedings of the 36th international conference on software engineering, pp 345–355

Gousios G, Zaidman A, Storey MA, Van Deursen A (2015) Work practices and challenges in pull-based development: the integrator's perspective. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering, IEEE, vol 1, pp 358–368

Gu X, Zhang H, Zhang D, Kim S (2016) Deep api learning. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, pp 631–642

Hoang T, Dam HK, Kamei Y, Lo D, Ubayashi N (2019) Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In: 2019 IEEE/ACM 16th international conference on mining software repositories (MSR), IEEE, pp 34–45

Huo X, Li M, Zhou ZH, et al (2016) Learning unified features from natural and programming languages for locating buggy source code. In: IJCAI, pp 1606–1612

Jiang J, He JH, Chen XY (2015) Coredevrec: automatic core member recommendation for contribution evaluation. J Comput Sci Technol 30(5):998–1016

Jiang J, Yang Y, He J, Blanc X, Zhang L (2017) Who should comment on this pull request? analyzing attributes for more accurate commenter recommendation in pull-based development. Inf Softw Technol 84:48–62

Kingma DP, Ba J (2014) Adam: A method for stochastic optimization. arXiv preprint arXiv:14126980

Le Q, Mikolov T (2014) Distributed representations of sentences and documents. In: International conference on machine learning, pp 1188–1196

Lee JB, Ihara A, Monden A, Matsumoto Ki (2013) Patch reviewer recommendation in oss projects. In: APSEC (2), pp 1–6

Levy O, Goldberg Y (2014) Neural word embedding as implicit matrix factorization. In: Advances in neural information processing systems, pp 2177–2185

Li HY, Shi ST, Thung F, Huo X, Xu B, Li M, Lo D (2019) Deepreview: automatic code review using deep multi-instance learning. In: Pacific-Asia conference on knowledge discovery and data mining, Springer, pp 318–330

de Lima Júnior ML, Soares DM, Plastino A, Murta L (2015) Developers assignment for analyzing pull requests. In: Proceedings of the 30th annual ACM symposium on applied computing, pp 1567–1572

de Lima Júnior ML, Soares DM, Plastino A, Murta L (2018) Automatic assignment of integrators to pull requests: the importance of selecting appropriate attributes. J Syst Softw 144:181–196

Manning CD, Schütze H, Raghavan P (2008) Introduction to information retrieval. Cambridge University Press, Cambridge

Nair V, Hinton GE (2010) Rectified linear units improve restricted boltzmann machines. In: ICML

Pagliardini M, Gupta P, Jaggi M (2017) Unsupervised learning of sentence embeddings using compositional n-gram features. arXiv preprint arXiv:170302507

Rahman MM, Roy CK, Collins JA (2016) Correct: code reviewer recommendation in github based on cross-project and technology experience. In: Proceedings of the 38th international conference on software engineering companion, pp 222–231

Soares DM, de Lima Júnior ML, Plastino A, Murta L (2018) What factors influence the reviewer assignment to pull requests? Inf Softw Technol 98:32–43

Thongtanunam P, Tantithamthavorn C, Kula RG, Yoshida N, Iida H, Matsumoto Ki (2015) Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In: 2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER), IEEE, pp 141–150

Tsay J, Dabbish L, Herbsleb J (2014) Influence of social and technical factors for evaluating contribution in github. In: Proceedings of the 36th international conference on Software engineering, pp 356–366

Voorhees EM et al (1999) The trec-8 question answering track report. Trec 99:77–82

Willett P (2006) The porter stemming algorithm: then and now. Program

Xia X, Lo D, Wang X, Yang X (2015) Who should review this change?: Putting text and file location analyses together for more accurate recommendations. In: 2015 IEEE international conference on software maintenance and evolution (ICSME), IEEE, pp 261–270

Yang C, Zhang X, Lb Z, Fan Q, Wang T, Yu Y, Yin G, Hm W (2018) Revrec: a two-layer reviewer recommendation algorithm in pull-based development model. J Central South Univ 25(5):1129–1143

Ye X, Fang F, Wu J, Bunescu R, Liu C (2018) Bug report classification using lstm architecture for more accurate software defect locating. In: 2018 17th IEEE international conference on machine learning and applications (ICMLA), IEEE, pp 1438–1445

Yu Y, Wang H, Yin G, Ling CX (2014a) Reviewer recommender of pull-requests in github. In: 2014 IEEE international conference on software maintenance and evolution, IEEE, pp 609–612

Yu Y, Wang H, Yin G, Ling CX (2014b) Who should review this pull-request: reviewer recommendation to expedite crowd collaboration. In: 2014 21st Asia-Pacific software engineering conference, IEEE, vol 1, pp 335–342