Automatic Code Review by Learning the Revision of Source Code

Shu-Ting Shi[1], Ming Li[1,2], David Lo[3], Ferdian Thung[3] and Xuan Huo[1]
[1]National Key Laboratory for Novel Software Technology, Nanjing University, China
[2]Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing University, China
[3]School of Information Systems, Singapore Management University, Singapore
{shist, lim, huox}@lamda.nju.edu.cn, davidlo@smu.edu.sg, ferdiant.2013@phdis.smu.edu.sg

Abstract

Code review is the process of manual inspection on the revision of the source code in order to find out whether the revised source code eventually meets the revision requirements. However, manual code review is time-consuming, and automating such the code review process will alleviate the burden of code reviewers and speed up the software maintenance process. To construct the model for automatic code review, the characteristics of the revisions of source code (i.e., the difference between the two pieces of source code) should be properly captured and modeled. Unfortunately, most of the existing techniques can easily model the overall correlation between two pieces of source code, but not for the "difference" between two pieces of source code. In this paper, we propose a novel deep model named DACE for automatic code review. Such a model is able to learn *revision features* by contrasting the revised hunks from the original and revised source code with respect to the code context containing the hunks. Experimental results on six open source software projects indicate by learning the revision features, DACE can outperform the competing approaches in automatic code review.

Introduction

Code review is important for software maintenance and evolution. A general process of code review is shown in Figure 1. Whenever a revision of the source code is triggered, a manual review or inspection the revision would be conducted, upon request, to check whether the revised source code eventually meets the revision requirements. The revisions that fail to fulfill the requirements would be rejected to be incorporated into the software system. In order to make a correct judgment, the code reviewer needs to carefully read the source code, analyzing the functionality and contrasting the revised source code and the original source code, which cost a huge amount of human efforts. To alleviate the burden of code reviewers and speed up the software maintenance process, techniques to automate the code review process are required.

Automatic code review can be formalized as a machine learning task where a model is constructed to take the revision on the source code (i.e., the original and revised source code) as input and output a recommendation on whether the revision should be rejected. To construct a well-performing model, the revision of source code, i.e., the "difference" between the original and revised source code, should be properly modeled. Unfortunately, most of the existing software mining approaches are equipped for modeling the "correlation" rather than the "difference" between software artifacts. For example, in software clone detection (Wei and Li 2017), models are constructed to identify source code fragments that may have same functionality; in bug localization (Huo, Li, and Zhou 2016; Huo and Li 2017), models are constructing to model the correlation between the bug report and the source code that is potentially responsible for the reported bug. All these approaches tend to *focus* on and *emphasize* the "similar" aspects of the two software artifacts. However, in the automatic code review, most of the revised source code is almost the same as the original source code, and only a tiny little portion is different. If adapting these approaches to automatic code review, the "difference", which is the key for determining the review result, would be ignored by these approaches. Thus, how to model the revision of source code is the key challenge for automatic code review.

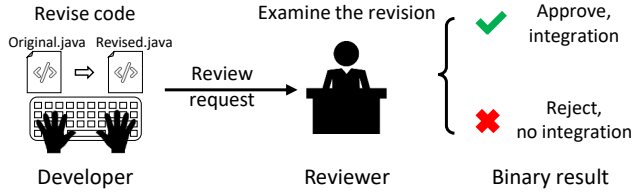

Figure 1: The process of code review.

To model the revision of source code and avoid being misled by the overwhelming among of unchanged code in the revision, a straight-forward solution is only to consider the *hunks*, i.e., the block of changed lines of code, and discard all the unchanged code. However, such a solution may perform poorly due to the following two reasons. First, the hunk is usually small and only contains a few consecutive lines. It is usually insufficient to modeling the correct functionality of the revised code. Second, the unchanged lines provide a *context* in which the revision is taken place. Such contextual information is sometimes crucial to determining the review result. Figure 2 provides a concrete example the same re-

vised hunk may lead to different review results if placed in different context. Therefore, in order to model the revision, the context should be considered appropriately.

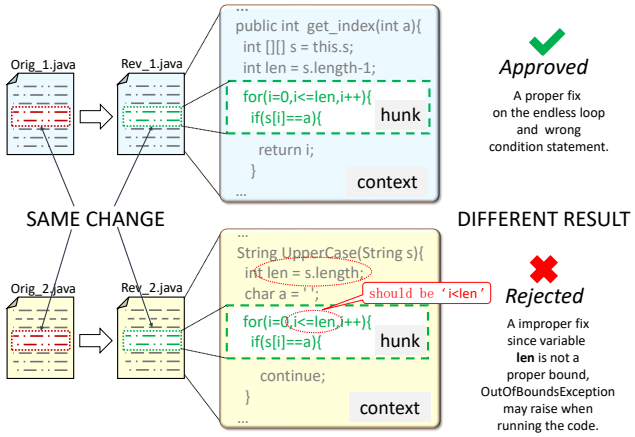

Figure 2: Context is indispensable. A same change may have different review results in different contexts. The second revision is improper because it may raise the *OutOfBoundsException* error.

One question arises here: can we learn the revision by contrasting the revised hunks from the original and revised source code with the code context properly embedded?

In this paper, we propose a novel deep learning method called DACE (Deep Automatic Code reviEw). This method first leverages CNN and LSTM to enrich the feature representation of each changed lines by exploiting their execution correlation with the context, and a particularly designed pairwise autoencoder is employed to learn the revision features from both the original hunks and revised hunks, based on which the review result is determined. Experimental results on six open source software projects indicate by learning the revision features that DACE can outperform the competing approaches in automatic code review.

The contributions of our work are:

- We put forward a new software mining challenge, which aims to model the difference between the two source files by learning the revision features.
- We propose a novel deep model which learns the revision features based on pairwise autoencoding and context-enriched representation of source codes.

The rest of this paper is organized as follows. In Section 2, we present the proposed DACE model. In Section 3, we report the experimental results. In Section 4, we discuss several related works and finally, in Section 5, we conclude the paper and issue some future works.

The DACE Model

We formalize the code review as a learning task, which is a binary classification problem. Given a sample of data $(c_i^O, c_i^R, y_i), i \in \{1, 2, ..., m\}$, where $c_i^O \in \mathcal{C}^O$ denotes the collection of *Original code* (referring to the source code before change) and $c_i^R \in \mathcal{C}^R$ denotes the collection of *Revised code* (referring to the source code after change), respectively. $y_i \in \mathcal{Y} = \{0, 1\}$ indicates whether the change is approved when $y_i = 0$ or rejected when $y_i = 1$, and m is the number of instances. The goal of DACE is to learn a mapping from $\mathcal{C}^O \times \mathcal{C}^R$ to $\mathcal{Y}$, $f : \mathcal{C}^O \times \mathcal{C}^R \mapsto \mathcal{Y}$, to predict whether an unseen code pair is rejected or not. The prediction function f can be learned by minimizing the following objective function:

$$\min_f \sum_i \mathcal{L}(f(c_i^O, c_i^R), y_j) + \lambda\Omega(f), \tag{1}$$

where $\mathcal{L}(\cdot, \cdot)$ is the empirical loss and $\Omega(f)$ is a regularization term imposing on the prediction function, and λ is the trade-off parameter to be tuned.

We instantiate the aforementioned learning task by proposing a novel deep neural network DACE which takes the source code pairs as input and then learns a revision feature for a given (c_i^O, c_i^R) pair, based on which the prediction can be made with a subsequent linear output layer.

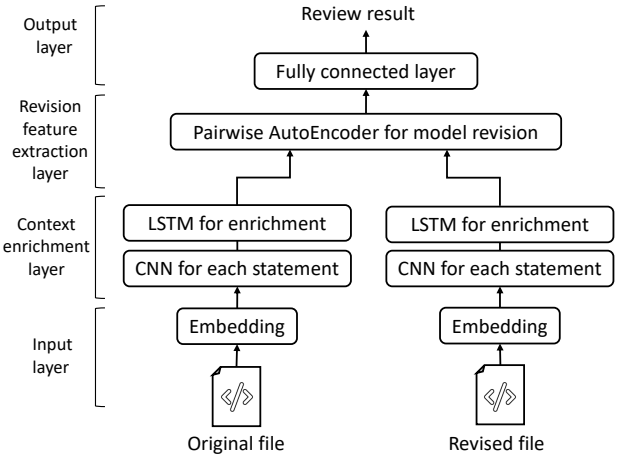

Figure 3: The general framework of DACE.

The general framework of DACE is illustrated in Figure 3. The DACE model contains four parts: input layer, context enrichment layer, revision feature extraction layer, and the output layer. In order to feed the raw textual data to the neural network, (c_i^O, c_i^R) are firstly embedded by the input layer. Then the embedded data of source code pair is fed into context enrichment layer which leverages CNN and LSTM to enrich the feature representation of each changed lines by exploiting their execution correlation with the context. After that, in revision feature extraction layer, to capture the relationship of revision between (c_i^O, c_i^R), we trained a pairwise recursive autoencoder that could learn a fixed length compact feature from two sequential data. Finally, after generating the revision feature representation, in the output layer, DACE predicts a review score for each (c_i^O, c_i^R) pair by a fully connected layer. The detail of these parts will be discussed in the subsequent subsections.

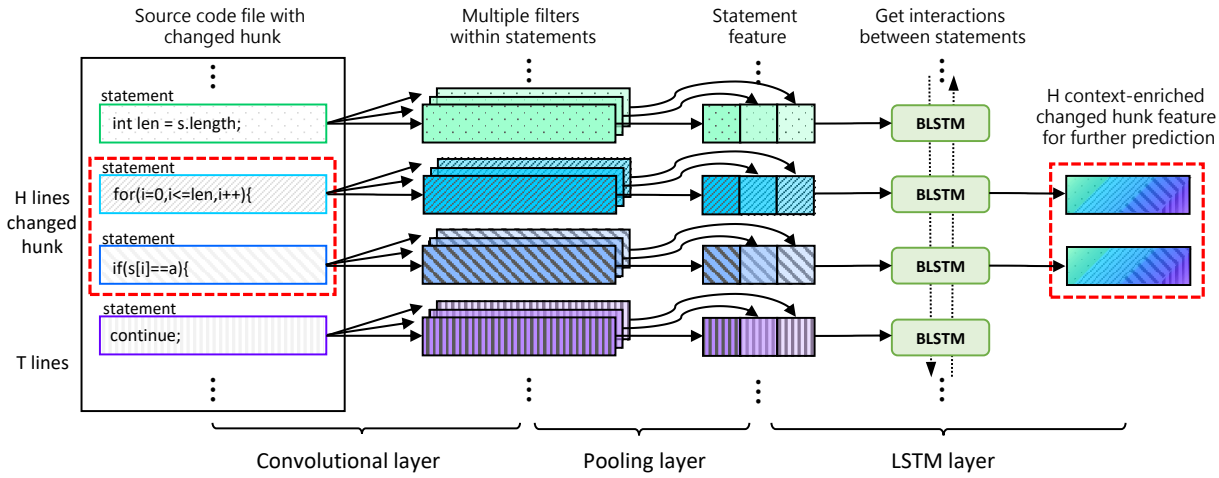

Figure 4: The structure of context enrichment layer. The convolutional and pooling layer aim to represent the semantics of a statement based on the terms within a statement, and the subsequent LSTM layer is used to enrich the sequential information for the changed hunk from relevant surrounding statements.

Data Processing

In the data preprocessing, we extract both the original code and revised code from the review system as input. In this paper, we assume all the change are i.i.d., so there is no interaction between all the change, and if a file has multiple changed hunks, we extract each hunk separately. There are situations that the modification is only adding or deleting, which causes half of the input pair to be empty set and is unable to feed into the network. So in this paper, we only consider the situation that the developer changed codes. In this settings, the input of each instance in the dataset can be formed as (c_i^O, c_i^R), where $c_i^O \in \mathcal{C}^O$ denotes the collection of original code and $c_i^R \in \mathcal{C}^R$ denotes the collection of revised code, respectively. It is worth noting that the c_i^O usually only contains a few consecutive lines that different with c_i^R while most of the lines in c_i^O are unchanged and is exactly the same with that in c_i^R. The different part in c_i^O is referring to as the original hunk. And the corresponding different part in c_i^R is referring to as the changed hunk. Both the original hunk and the changed hunk is a consecutive line of code.

To obtain word embeddings, a pre-trained word2vec (Mikolov et al. 2013) technique is used to embed every text term in (c_i^O, c_i^R) 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 (Mikolov et al. 2013; Kim 2014). Since the meaning of text term in software is not the same as that in natural language, the weights of pre-trained word2vec will be tuned later.

The key of the DACE model lies in the context enrichment layer as well as the revision feature extraction layer, which will be discussed in detail in the following subsections.

Context Enrichment Layer

Automatic code review faces the challenge of *context dilemma*: not capturing the context would be too deficient to predict but capturing the context all would overwhelm the information of changed hunk. That is because the context provides abundant correlated knowledge of the changed hunk along with plenty of irrelevant information.

One question arises here: can we find a way to enrich the changed hunk with useful information in context, rather than use raw intact context or despite it? To process source code, Huo et al. (Huo, Li, and Zhou 2016) designed a particular CNN network for source code processing, which extracts features based on multiple layers of convolutional neurons, where the convolution operation for source code is particularly designed to reect the program structure and preserves statement integrity. Moreover, Huo et al. (Huo and Li 2017) claims that one crucial point to extract semantic features from the source code is that, the statements in programming language contain sequential nature, which means the previous statements may affect subsequent statements according to the execution path and statements may have long-term dependency via data stream transmission. To extract semantic features of source code file, LSTM is designed to extract semantic features reflecting sequential nature from source code and handle long-term dependency between statements and CNN is designed to capture the local and structure information within statements. Inspired by the aforementioned methods, to solve the context dilemma, a richer feature representation which captures the sequential semantics of the surrounding code is necessary to be exploited, especially for the fragmentary changed hunk which is usually inefficient to represent the program functionality. To enrich the feature of statements, we extend CNN for source code processing (Huo, Li, and Zhou 2016) by combining Long Short-Term Memory (LSTM) (Sutskever, Vinyals, and Le 2014),

a type of Recurrent Neural Network (RNN) that captures the sequential relevance between statements, which has been demonstrated effective for maintaining sequential property in text processing, to exploit sequential features for fragmentary changed hunk from the source code file. To avoid the gradient vanishing and exploding problems for LSTM processing data with too much time steps, and considering the property of the JAVA programming language, we only take the method block where the revision take place.

The structure of the programming language feature extraction network is illustrated in Figure 4. The first convolutional layer aims to extract semantic features within statements to protect the integrity of programming language, which employs multiple filters sliding within the statement and converts statements into new feature vectors. In order to extract high-level features with different granularity, the filters in the convolutional layer have different sizes. A max pooling operation is applied within the statement to extract the most informative information of each feature map. After convolutional and pooling process, the network generates T feature maps (H feature maps of changed hunk included) of $\mathbf{x} \in R^k$, where T is the length of the JAVA method, H is the length of the changed hunk inside the method, and k is the dimension for each statement's feature generated by the pooling layer.

After processed by convolutional and pooling operations, the T feature maps are then fed into the LSTM layer. LSTM is a recurrent neural network (RNN) that takes terms in a sequence one by one, i.e., at time t, it takes the t-th term as input. In our model, $\mathbf{x}_t \in R^k$ in the input vectors in time step t, represents the feature maps of t-th statements generated by CNN. Therefore, the input vectors maintain the inherent sequential nature, which can be fed into LSTM that is specified for sequential inputs. LSTM overcomes the difficulty in learning long-term dynamics by incorporating memory cells c that allow the network to learn when to forget previously hidden states and when to update hidden states given new information, which is able to exploit sequential nature from source code to enrich the high-level semantic features.

By take the changed hunk's representation of length H, which contains the sequential and helpful information out flown from memory cells, we can obtain a richer sequential feature representation which captures the sequential semantics of the surrounding code.

Revision Feature Extraction Layer

In this subsection, we introduce the structure of the revision feature extraction layer, where a novel network structure, pairwise autoencoder (PAE) is designed. In short, PAE construct an encoder to pact the revision feature of (c_i^O, c_i^R) into a compact feature representations by exploiting their fused context vector which is able to facilitate the determination on whether a revision of source code should be rejected or approved.

Back during the process of human code review, reviewers compare the original code to the revised code, and sum up with an abstract term denoting the revision in their mind, and further determine this revision is proper or improper. This process inspires us to extract a feature that denotes the meaning of revision given a pair of source code. To do this, one of the simplest ways is to concatenate all of the statements in original code and revised code together to represent revision. But this kind of splice method will suffer from the loss of wasting the structure of sequential statements and ignoring the order of code version. However, the structure of statements is important because that is how different statements interact with each other to accomplish specific functionality. Moreover, the order of code version is essential for reviewer, e.g., a revision `code1` $\rightarrow$ `code2` usually has a result that different from the revision `code2` $\rightarrow$ `code1`. That is to say, the feature of revision should summarize the meaning of change with both structure of statements within hunk and the change direction between hunks.

One question arises here, can we learn a revision feature that indirectly learns the change process from a pair of sequential data? To address this problem, we design a novel network structure pairwise autoencoder (PAE), an extension of Seq2Seq autoencoder, but with pairwise sequences as input. An illustration of two structure is shown in Figure 5. Classical autoencoder learns efficient data codings from sequential data in an unsupervised manner and has been demonstrated effective. Different from classical autoencoder, PAE receives pairwise input, and a particular fusion operation is designed to fuse two hidden states into a fixed length vector.

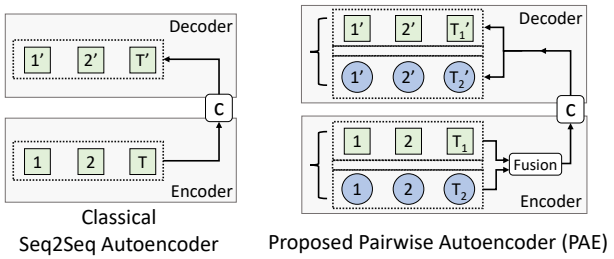

Figure 5: An illustration of the Seq2Seq Autoencoder and the proposed PAE.

The encoder of PAE encodes the pairwise features generated from the last layer into a hidden feature that can be decoded back to the feature pair by minimizing the reconstructed loss. When the reconstructed loss closing to zero or small enough within tolerance, the hidden feature becomes an efficient feature of revision. That is because using this hidden state the decoder can reconstruct the input pair. The hidden state knows the version of two hunks and is able to capture the structure in both hunks since every statement can be reconstructed in the right order. Figure 6 below illustrates the model. The traditional autoencoder reads the input sequence $S_{seq} = \{s_1, s_2, ..., s_T\}$ sequentially, where $s_i \in R^k$. The hidden state of the RNN updates at every time step based on the current input and hidden state inherited from the previous step. The last hidden state which refers to context vector c which are used for decoding. In our model, the encoder reads two sequential vectors respectively and gets a pair of context vectors $(context^O, context^N)$, which refers to the

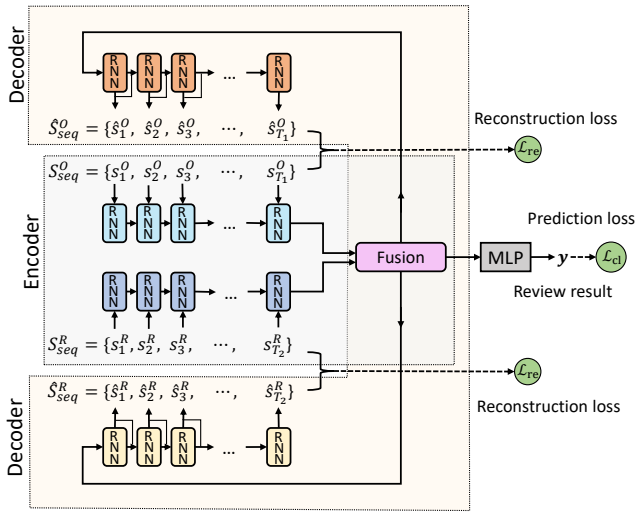

Figure 6: The structure of PAE. A pairwise sequential data been encoded into a pair of context vector by two RNN based encoder, then through a maxpool fusion operator they are fused into one overall context vector which could been decoded back to a pair through two RNN decoder. The encoders pact the revised code into a compact feature representations based on which the transformation can be learned.

context vector of input (c^O, c^N), respectively. Then a fusion operation is applied on $(context^O, context^N)$ to get a joint compact overall context vector $context$. Note that the fuse operation can be implemented as a different kind of mechanism, e.g., max pooling, linear transformation, multiple layers of fully connected work, etc. In this paper, we use max pooling to extract the most informative feature and get the overall context vector. The decoder is a recurrent network which reconstructs $context$ back into the original pairwise sequence. To exemplify this, the decoder starts by reading the context vector $context$ at $t = 1$. It then decodes the information through two RNN structure and outputs two sequences of reconstruction features $\widehat{S}^O_{seq}$ and $\widehat{S}^N_{seq}$.

At a high level, the RNN encoder reads an input pair sequence separately and summarizes all information into a fixed length vector. This vector contains the knowledge of both the revision and the semantic meaning of source code since the decoder can read the vector and reconstructs the original sequence pair. By minimizing the reconstruction loss, the internal structure bounded the hidden state with the condition that to extract the revision information of the input pair, so that it is able to narrow the search space of parameters. The overall context vector is feed into a fully connected layer for prediction.

Specifically, the parameters of PAE can be denoted as Θ_{PAE}, and the parameters of the convolutional neural networks layer and the LSTM can be denoted as Θ_{CNN} and Θ_{LSTM}, respectively, and the parameters of fully-connected networks layer is W. Let $\Theta = \{\Theta_{CNN}, \Theta_{LSTM}, \Theta_{PAE}\}$, therefore, the loss function implied in DACE is:

$$\mathcal{L}(\Theta, W) = \mathcal{L}_{cl}(\Theta, W) + \lambda \mathcal{L}_{re}(\Theta), \tag{2}$$

in which

$$\mathcal{L}_{cl}(\Theta, W) = -\sum_{i=1}^{m}(c_a y_i \log \hat{p}_i + c_r(1-y_i)\log(1-\hat{p}_i)), \tag{3}$$

$$\mathcal{L}_{re}(\Theta) = \sum_{i=1}^{m}(\frac{1}{T_i^O}\sum_{j=1}^{T_i^O}||s_{ij}^O - \hat{s}_{ij}^O||_2^2 + \frac{1}{T_i^R}\sum_{k=1}^{T_i^R}||s_{ik}^R - \hat{s}_{ik}^R||_2^2). \tag{4}$$

where $\mathcal{L}_{cl}$ is a cross-entropy classification loss, and $\mathcal{L}_{re}$ is the restrict term denotes reconstruction loss in PAE and λ is the trade-off parameter balancing these two terms. And c_a denotes the cost of incorrectly predicting a rejected change as approved and c_r denote the cost of incorrectly predicting an approved change as rejected, and these two terms provide an opportunity to handle imbalanced costs or imbalanced label distribution.

Another interpretation of loss function in DACE is that, when the reconstructed loss is close enough to zero (smaller than a constant), the PAE can capture the revision feature accurately relatively. Under this condition, the hidden state $context$ are further used to learn a classifier in the output layer. The overall optimization can be formed as:

$$\begin{aligned} \min_{\Theta, W} \ & \mathcal{L}_{cl}(\Theta, W), \\ \text{s.t.} \ & \mathcal{L}_{re}(\Theta) \leq C. \end{aligned} \tag{5}$$

where $\mathcal{L}_{cl}$ and $\mathcal{L}_{re}$ are defined in Equation 3 and Equation 4, and C is a constant. Interestingly, the form of Equation 5 and the form of Equation 2 are equivalent with a simple deduce of Lagrange multipliers.

This objective function can be effectively optimized using SGD (Stochastic gradient descent) (Bottou 1998).

Experiments

To evaluate the effectiveness of DACE, we conduct experiments on open source software projects and compare the results with several baselines.

Experiment Settings

The code review data used in the experiments is crawled from Apache projects[1], which is a widely used code review source (Rigby and Bird 2013; Rigby and German 2006; Rigby, German, and Storey 2008). In many practical cases, a review request may contain multiple files and hunks, and we assume these hunks are independent of each other. For each changed hunk, if it has the highlighted lines that marked by reviewers denoting they have issues, we regard the hunk as rejected. It is worth noting that our model is language-independent. Although we conducted our experiments on JAVA projects, it can be directly applied to the automatic code review tasks with other programming languages (e.g., C++, Python). We divided our dataset into six

[1]Apache Code Review Board, https://reviews.apache.org/r/

repositories, as shown in Table 1, i.e., *cloudstack*, *ambari*,*aurora*, *drill-git*, *accumulo* and *hbase-git*. For each repository, we have at least 3,500 hunks.

Table 1: Statistics of our data sets.

Repository	#hunks	#rejected	reject rate
accumulo	5,620	152	3%
ambari	6,810	138	2%
aurora	6,762	168	2%
cloudstack	6,171	128	2%
drill-git	3,575	43	1%
hbase-git	6,702	140	2%

Because it has the class imbalance problem, the total number of rejects is far less than the total number of approves, so the most common metrics like accuracy and error can be deceiving in certain situations and are highly sensitive to changes in data. For each data set, 10-fold cross validation is repeated ten times, and we use F1 and AUC (Area Under ROC Curve) to measure the effectiveness of the DACE model, which have been widely applied for evaluating imbalanced data. We report the average value of all compared methods in order to reduce the evaluation bias. We also apply statistic test to evaluate the significance of DACE, the pairwise *t*-test at 95% confidence level is conducted.

Since there are no previous studies that applied machine learning methods for automatic code review tasks, we firstly compare our proposed model DACE with several traditional models on software engineering. One of the most common methods is to employ the Vector Space Model (VSM) to represent the source code and then train a classifier to predict if a change is approved or not. Applying VSM in software engineering tasks have been widely studied (Gay et al. 2009; Liu et al. 2007; Zhou, Zhang, and Lo 2012). In addition, we compare DACE with the latest deep learning based models Deeper (Yang et al. 2015) on software engineering, which applies Deep Believe Network for semantic feature extraction. Moreover, to explore the effect of each layer of our model, several variants of DACE have also been compared. Specifically, the compared methods are as follows:

- TFIDF-LR (Schütze, Manning, and Raghavan 2008; Gay et al. 2009), which uses TFIDF technique to represent the original and the revised source code. After a concatenating operation, a basic classifier Logistic Regression (LR) is used for prediction.
- TFIDF-SVM, which uses TFIDF technique to representing which is the same as above. After representing, a basic classifier Support Vector Machine (SVM) is used for prediction.
- Deeper (Yang et al. 2015), which is one of state-of-the-art deep learning model on software engineering, which uses some basic features (e.g., how many lines of code added, lines of code deleted, etc.) to extract expressive features using Deep Belief Network (DBN) for changes and then apply Logistic Regression for classification.
- Deeper-SVM, which is a slight variant of state-of-the-art model Deeper. Deeper-SVM uses the same DBN model for feature extraction but apply Support Vector Machine for classification.
- LSCNN, a variant of DACE that without PAE. It is similar to the state-of-the-art method LSCNN (Huo and Li 2017) using Multilayer Perceptron (MLP) for prediction.
- PAE, a variant of DACE that without considering sequential and long-term dependency information between statements in context extraction layer and only use PAE to extract features from CNN.

For TFIDF-LR and TFIDF-SVM, we build a vocabulary that considers the top 300 frequency term and using class-balanced weight for classifiers, and in TFIDF-SVM the penalty parameter C are chosen by cross-validation. For Deeper and Deeper-SVM, we follow the same experiment settings in (Yang et al. 2015). We employ the most commonly used ReLU $\sigma(x) = max(x, 0)$ as active function and the filter windows size is set as 2, 3, 4, with 100 feature maps each in CNN. The number of neuron dimension in LSTM is set as 300. The encoders and decoders in PAE are GRUs with the cell size of 256. And the MLP for final prediction is two layers of a fully connected network of size 256 and 100. The cost weights c_a and c_r are set inversely proportional to class instance numbers.

Experiment Results

For each data set, 10-fold cross validation is repeated ten times and the average performance of all compared methods with respect to F1 and AUC are tabulated in Table 2 and Table 3, where the best performance on each data set is boldfaced. Mann-Whitney U-test is conducted at 95% confidence level to evaluate the significance. If DACE significantly outperforms a compared method, the inferior performance of the compared method would be marked with ∘ .

Table 2: The performance comparison in terms of F1.

Repository	TFIDF -LR	TFIDF -SVM	Deeper	Deeper -SVM	LSCNN	PAE	DACE
accumulo	0.227∘	0.239∘	0.202∘	0.199∘	0.417∘	0.373∘	**0.493**
ambari	0.240∘	0.278∘	0.306∘	0.238∘	0.444∘	0.473∘	**0.509**
aurora	0.204∘	0.220∘	0.349∘	0.299∘	0.336∘	**0.571**	0.403
cloudstack	0.250∘	0.275∘	0.352∘	0.265∘	0.360∘	0.415∘	**0.516**
drill-git	0.212∘	0.236∘	0.229∘	0.212∘	0.318∘	0.382∘	**0.573**
hbase-git	0.232∘	0.256∘	0.193∘	0.154∘	0.348∘	**0.411**	0.396
Average	0.228∘	0.251∘	0.272∘	0.228∘	0.370∘	0.438∘	**0.482**

Table 3: The performance comparison of in terms of AUC.

Repository	TFIDF -LR	TFIDF -SVM	Deeper	Deeper -SVM	LSCNN	PAE	DACE
accumulo	0.666∘	0.703∘	0.688∘	0.705∘	0.787	**0.814**	0.786
ambari	0.708∘	0.848∘	0.680∘	0.572∘	0.824∘	0.861∘	**0.905**
aurora	0.582∘	0.645∘	0.682∘	0.564∘	0.750∘	**0.819**	0.793
cloudstack	0.745∘	0.827∘	0.795∘	0.646∘	0.761∘	0.820∘	**0.852**
drill-git	0.658∘	0.725∘	0.593∘	0.540∘	0.788∘	0.806∘	**0.820**
hbase-git	0.679∘	0.759∘	0.590∘	0.524∘	0.751∘	0.764∘	**0.813**
Average	0.673∘	0.751∘	0.671∘	0.592∘	0.777∘	0.814∘	**0.828**

From the results, we can find that DACE achieves the best average performance in both terms of F1 and AUC. The F1 of DACE improves traditional baseline models (TFIDF-LR,

TFIDF-SVM) because DACE spends more time and space to extract features during the training process. Compared with the best deep model Deeper and its variant Deeper-SVM, DACE still has a better performance.

Area under the receiver operating characteristic curve (AUC) is a standard approach to imbalanced classification. AUC measures the comprehensive performance of different predictors and DACE still performs the best on average. Comparing with the VSM based model as well as the best deep learning model Deeper and its variant Deeper-SVM, DACE still improves the performance.

Additionally, although the state-of-the-art method for programming language LSCNN (Huo and Li 2017) also utilizes CNN and LSTM in source code processing, in code review task, we find LSCNN does not perform well in comparison to DACE. One of the differences between LSCNN and DACE is that DACE utilizes PAE in the revision feature extraction layer, and the internal structure of PAE helps DACE to captures the information of revision by narrowing searching space, thus leading to a better feature representation.

Moreover, although PAE is particularly designed for code review, it still not perform better than DACE. This is probably because it falls into the context dilemma. For some datasets, with the absence of context the information of changed hunk is too deficient to review. The example in Figure 2 may confuse the model by the same input leading to opposite output. That is because the context provides abundant correlated information of the changed hunk. Experiments show that the convenient of ignoring the context, e.g., less computation and simpler hypothesis space, is not enough to make up for the absence of context for many datasets.

In summary, the revision feature of original-revised source code pair would be learned by DACE which applies PAE to model revision. And experiments also show the efficiency of exploiting the sequential nature as well as enriching the context of the changed hunk.

Related Work

Previous empirical studies have shown that code review practice involves a significant amount of human effort since reviewers need to understand, identify, analysis and discuss until they make the decision. Thus, many tasks and approaches that aim to improve the effectiveness of code review have been presented. Thongtanunam et al. (Thongtanunam et al. 2015) 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.

Zanjani et al. (Zanjani, Kagdi, and Bird 2016) 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). Their results showed that cHRev outperforms prior approaches. Different from the above works, Ebert et al. (Ebert et al. 2017) proposed an approach to identify the factors that confuse reviewers and understand how confusion impacts the efficiency and effectiveness of code reviewers. They first manually classify 800 comments and then they trained classifiers based on the labeled data and found that confusion identification in inline comments is more a difficult task than in general ones.

Recently, deep learning (Goodfellow, Bengio, and Courville 2016), which is a recent breakthrough in machine learning domain, has been applied in many areas. Software engineering is not an exception. 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 (Yang et al. 2015). Guo et al. use word embedding and one/two layers Recurrent Neural Network (RNN) to link software subsystem requirements (SSRS) to their corresponding software subsystem design descriptions (SSDD) (Guo, Cheng, and Cleland-Huang 2017). 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 (Xu et al. 2016). Lee et al. applied word embedding and CNN to identify developers that should be assigned to fix a bug report (Lee et al. 2017). Mou et al. (Mou et al. 2016), applied tree based CNN on abstract syntax tree to detect code snippets of certain patterns. Lam et al. (Lam et al. 2015) combined deep model autoencoder with a information retrieval based model, which shows good results for identifying buggy source code. Huo et al. (Huo, Li, and Zhou 2016; Huo and Li 2017) applied learned unified semantic feature based on bug reports in natural language and source code in a programming language for bug localization tasks. Wei et al (Wei and Li 2017) proposed an end-to-end deep feature learning framework for functional clone detection, which exploiting the lexical and syntactical information via AST-based LSTM network.

Conclusion

In this paper, we study the important problem of automatic code review and formalize this task as a learning problem. Then we propose a novel neural network called DACE, which first leverages CNN and LSTM to enrich the feature representation of each changed lines by exploiting their execution correlation with the context, and a particularly designed pairwise autoencoder (PAE) is employed to learn the revision features from both the original hunks and revised hunks, based on which the review result is determined. Experimental results on six open source repositories show the efficiency of DACE.

In the future, considering the relationship between hunks will be investigated. And since the users are often very sensitive to code review errors, a cost-sensitive analysis will be studied. Moreover, incorporating additional data to enrich the structure of DACE is also another interesting future work.

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

References

Bottou, L. 1998. *Online learning and stochastic approximations*. Cambridge University Press.

Ebert, F.; Castor, F.; Novielli, N.; and Serebrenik, A. 2017. Confusion detection in code reviews. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, 549–553.

Gay, G.; Haiduc, S.; Marcus, A.; and Menzies, T. 2009. On the use of relevance feedback in ir-based concept location. In *Proceedings of International Conference on Software Maintenance*, 351–360.

Goodfellow, I.; Bengio, Y.; and Courville, A. 2016. *Deep Learning*. MIT Press.

Guo, J.; Cheng, J.; and Cleland-Huang, J. 2017. Semantically enhanced software traceability using deep learning techniques. In *Proceedings of the International Conference on Software Engineering*, 3–14.

Huo, X., and Li, M. 2017. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1909–1915.

Huo, X.; Li, M.; and Zhou, Z.-H. 2016. Learning unified features from natural and programming languages for locating buggy source code. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1606–1612.

Kim, Y. 2014. Convolutional neural networks for sentence classification. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 1746–1751.

Lam, A. N.; Nguyen, A. T.; Nguyen, H. A.; and Nguyen, T. N. 2015. Combining deep learning with information retrieval to localize buggy files for bug reports. In *Proceedings of the International Conference on Automated Software Engineering*, 476–481.

Lee, S.; Heo, M.; Lee, C.; Kim, M.; and Jeong, G. 2017. Applying deep learning based automatic bug triager to industrial projects. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, 926–931.

Liu, D.; Marcus, A.; Poshyvanyk, D.; and Rajlich, V. 2007. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 234–243.

Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G. S.; and Dean, J. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, 3111–3119.

Mou, L.; Li, G.; Zhang, L.; Wang, T.; and Jin, Z. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 1287–1293.

Rigby, P. C., and Bird, C. 2013. Convergent contemporary software peer review practices. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, 202–212.

Rigby, P. C., and German, D. M. 2006. A preliminary examination of code review processes in open source projects. Technical report, Technical Report DCS-305-IR, University of Victoria.

Rigby, P. C.; German, D. M.; and Storey, M.-A. 2008. Open source software peer review practices: a case study of the apache server. In *Proceedings of the International Conference on Software Engineering*, 541–550.

Schütze, H.; Manning, C. D.; and Raghavan, P. 2008. *Introduction to information retrieval*. Cambridge University Press.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, 3104–3112.

Thongtanunam, P.; Tantithamthavorn, C.; Kula, R. G.; Yoshida, N.; Iida, H.; and Matsumoto, K.-i. 2015. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*, 141–150.

Wei, H.-H., and Li, M. 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 3034–3040.

Xu, B.; Ye, D.; Xing, Z.; Xia, X.; Chen, G.; and Li, S. 2016. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 51–62.

Yang, X.; Lo, D.; Xia, X.; Zhang, Y.; and Sun, J. 2015. Deep learning for just-in-time defect prediction. In *Proceedings of IEEE International Conference on Software Quality, Reliability and Security*, 17–26.

Zanjani, M. B.; Kagdi, H.; and Bird, C. 2016. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering* 42(6):530–543.

Zhou, J.; Zhang, H.; and Lo, D. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the International Conference on Software Engineering*, 14–24.