Proceedings of Machine Learning Research 129:481–496, 2020 ACML 2020

Learning Code Changes by Exploiting Bidirectional Converting Deviation

Jia-Wei Mi mijw@lamda.nju.edu.cn
Shu-Ting Shi shist@lamda.nju.edu.cn
Ming Li lim@nju.edu.cn
National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China

Editors: Sinno Jialin Pan and Masashi Sugiyama

Abstract

Software systems evolve with constant code changes when requirements change or bugs are found. Assessing the quality of code change is a vital part of software development. However, most existing software mining methods inspect software data from a static view and learn global code semantics from a snapshot of code, which cannot capture the semantic information of small changes and are under-representation for rich historical code changes. How to build a model to emphasize the code change remains a great challenge. In this paper, we propose a novel deep neural network called CCL, which models a forward converting process from the code before change to the code after change and a backward converting process inversely, and the change representations of bidirectional converting processes can be learned. By exploiting the deviation of the converting processes, the code change can be evaluated by the network. Experimental results on open source projects indicate that CCL significantly outperforms the compared methods in code change learning.

Keywords: Software mining, Machine learning, Deep learning, Learning code changes, Bidirectional converting deviation

1. Introduction

Software mining aims to look for patterns in large batches of software data, which can facilitate a productive development of software and high-quality software products. Mining novel and useful patterns from data involved in software systems improves the efficiency of software tasks, e.g., clone detection (Jiang et al., 2007; White et al., 2016), bug localization (Gay et al., 2009; Huo et al., 2016), defect prediction (Tan et al., 2015; Wang et al., 2016), etc. These studies all share the same principle that inspects software data from a static view, so that all the semantic representations of code are learned from one current snapshot of software systems.

However, software systems are not static and software change is an integral part of software engineering (Sommerville, 2007). For example, code in software changes when bugs are fixed, when software requirements are modified, when new features are added, and code in software changes to improve the robustness, to match new API versions, etc. Figure 1 shows an example. As software systems become increasingly large and complex, the need increases to consider the effects of software changes (Arnold, 1996). It brings many software mining challenges, such as code review, which is the process of inspection on the revision of the source code, and just-in-time defect prediction, which focuses on predicting whether

© 2020 J.-W. Mi, S.-T. Shi & M. Li.

```
DateFormat formatter = new SimpleDateFormat("MM/dd/yy");
Date date = formatter.parse("01/29/02");
```

```
DateFormat formatter = new SimpleDateFormat("MM/dd/yy");
try {
  Date date = formatter.parse("01/29/02");
}catch (ParseException e) {
  e.printStackTrace();
}
```

Figure 1: An example of JAVA code. Code changes to improve the robustness, where adding exception handling mechanism enables the program to deal with the exception which may be caused by `formatter.parse`.

a particular file involved in a change is buggy to help developers fix defects as soon as they are introduced, etc. Although we can study these challenges case by case, they all share one important commonality: these tasks all rely on modeling code changes. Code change learning is a major task in software mining. Assessing the quality of code change is a vital part of software development and is essential for software quality assurance. By learning from code changes, we can provide valuable feedback in the process of software evolution.

Nevertheless, learning to model code changes is not easy. Code change learning is different from previous tasks, e.g., clone detection and bug localization, which learn global semantics of a snapshot and correlate the relevance by measuring functional similarity. In most cases, the change of code is small that the global semantics of the code before change and the code after change are almost the same. The methods without a particular design for code changes may lose efficacy when two versions of source files have a slight difference. So how to model the change process between pairs of source code remains a great challenge.

Recently, Shi et al. (2019) propose a method to learn the correlation between the code before change and the code after change by packing them together, which is probably the first attempt to address this problem. However, it merely packs the code before change and the code after change into a fixed-length vector, which does not explicitly model the *change* itself. The fixed-length vector simply packs all relevant and irrelevant information about the change and it leaves a heavier burden on the following classifiers to capture the key information of the change. For example, if `codeA`, `codeB` and `codeC` have slight differences, it is difficult for a predictor to decide whether a change from `codeA` to `codeB` is better than the change from `codeA` to `codeC` using two fixed-length vectors which are almost the same. It indicates that the summarized vectors are under-representation for code changes. However, the expressivity and the ability to distinguish different changes are vital for modeling.

The key problem for learning the process of code change is to correctly guide the model to fit the converting directions. A straightforward solution is only to consider the converting process from the code before change to the code after change. However, such a solution may perform poorly because there is no constraint for the representation of the code after change. To address this problem, in this paper we propose a novel deep neural network called CCL (Code Change Learning) to learn code changes by exploiting bidirectional converting

deviation. Such a method mainly consists of two phases. The first phase is for context embedding, which combines CNN and LSTM to exploit local semantics within changed lines and contextual information about the functional semantics of the code. The second phase is for modeling a forward converting process from the code before change to the code after change and a backward converting process inversely, and the change representations of bidirectional converting processes can be learned. The converting deviation, which is measured by the difference between the two change representations, is used to evaluate the code change. Experimental results on open source projects indicate that modeling bidirectional converting processes and exploiting converting deviation are beneficial for learning code changes, and our method CCL significantly outperforms the compared methods.

The contributions of our work are summarized as follows:

- We highlight that code change learning is a major task in software mining, and we are the first to model the change process of code directly.
- We design a general deep framework named CCL for code change learning, where a forward converting process and a backward converting process are modeled and the change representations which capture the key of the change can be learned by exploiting the deviation of the bidirectional converting processes.

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

2. Method

In this work, we focus on the code change learning task. We are provided with a set of code pairs, which contain the code before change and the code after change. The goal of code change learning is to predict whether the change is good or bad (i.e., meeting the certain objective or not), which can be formulated as a binary classification problem.

Let $\mathcal{C}^O = \{c_1^O, c_2^O, \cdots, c_n^O\}$ denotes the set of old code (i.e., the source code before changes), $\mathcal{C}^N = \{c_1^N, c_2^N, \cdots, c_n^N\}$ denotes the set of new code (i.e., the source code after changes), and n is the number of instances. We attempt to learn a prediction function $f : \mathcal{C}^O \times \mathcal{C}^N \mapsto \mathcal{Y}$. $y_i \in \mathcal{Y} = \{+, -\}$ indicates the change from c_i^O to c_i^N is a good change which improves the quality of code when $y_i = +$ and we use (c_{i+}^O, c_{i+}^N) to denote the code pair, or is a bad change when $y_i = -$ and we use (c_{i-}^O, c_{i-}^N) to denote the code pair.

We instantiate this code change learning task by proposing a novel deep neural network CCL. The general framework of CCL is shown in Figure 2. The CCL model contains two phases. The first phase is for context embedding, which takes the encoded data of a code pair (c_i^O, c_i^N) as input after word embedding and combines CNN and LSTM to extract the local as well as contextual features from source code. The code before change and the code after change are processed separately by the network. The second phase is for exploiting bidirectional converting deviation, in which we use autoencoder to model a forward converting process and a backward converting process of code. By exploiting the bidirectional deviation, the change representations of the forward process and the backward process can be learned. Finally, we apply the prediction function defined by Equation 1

$$f(c_i^O, c_i^N) = \mathbb{I}(d(\phi_F(\psi(c_i^O)), \phi_B(\psi(c_i^N))) \leq \theta) \quad (1)$$

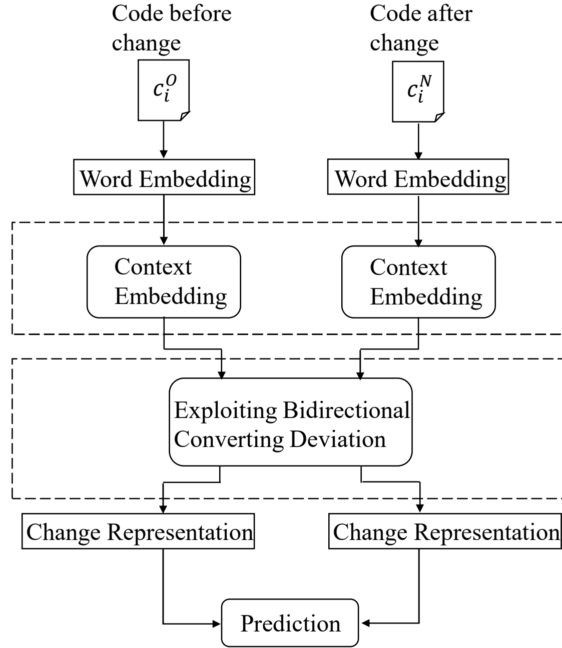

Figure 2: The general framework of CCL. The CCL model mainly contains two phases: context embedding and exploiting bidirectional converting deviation.

to decide whether the change from $c_i^O$to $c_i^N$is good. $\mathbb{I}(\cdot)$ is the indicator function which returns $+$ if the condition is satisfied and returns $-$ otherwise. $\psi(\cdot)$ is the mapping function specific to contex embedding. $\phi_F(\cdot)$ and $\phi_B(\cdot)$ are the mapping functions specific to the forward converting process and the backward converting process respectively. The parameter θ can be learned in the training set. The function $d(\cdot,\cdot)$ defined by Equation 2

$$d(\phi_F(\psi(c_i^O)),\phi_B(\psi(c_i^N))) = \|\phi_F(\psi(c_i^O)),\phi_B(\psi(c_i^N))\|_2 \tag{2}$$

returns the distance between the representation of forward converting process and the representation of backward converting process, which is used to measure the converting deviation.

The key of the CCL model lies in how to learn the features for context embedding and how to learn change representations by exploiting bidirectional converting deviation, which will be introduced in the following subsections.

2.1. Context Embedding

After the processing of the word embedding, we obtain the embedding of each token in source code. To learn code changes, firstly we need to extract context features to represent the code before change and the code after change separately. Context embedding for code change learning faces some challenges: local semantics within changed lines should be preserved, and contextual information should also be considered because most of the changes are so small that only the local semantics about the change is insufficient. The context provides abundant correlated knowledge of the changed code.

To exploit semantics of code, Huo and Li (2017) proposed a method that combines the LSTM and CNN models to enhance features by exploiting the sequential nature of source code, where CNN is particularly designed for source code to extract the local semantics within statements, and LSTM is used to exploit semantic features reflecting sequential nature and handle long-term dependency between statements, such that the functional semantics of the code can be embedded. Inspired by this method, we leverage CNN for source code (Huo et al., 2016) and Long Short-Term Memory (LSTM) (Sutskever et al., 2014) to extract the context embedding.

An illustration of the structure for context embedding is shown in Figure 3. The first convolutional layer aims to represent the semantics of a statement based on the tokens within the statement, which utilizes multiple filters sliding within statements and converts statements into new feature vectors. The filters in the convolutional layer have different sizes to capture local semantics within statements in different granularity. The following pooling layer aims to extract the most informative information of each feature map. The network generates T feature maps $\mathbf{z} \in \mathbb{R}^d$ after convolutional and pooling operations, where T is the number of statements in the code.

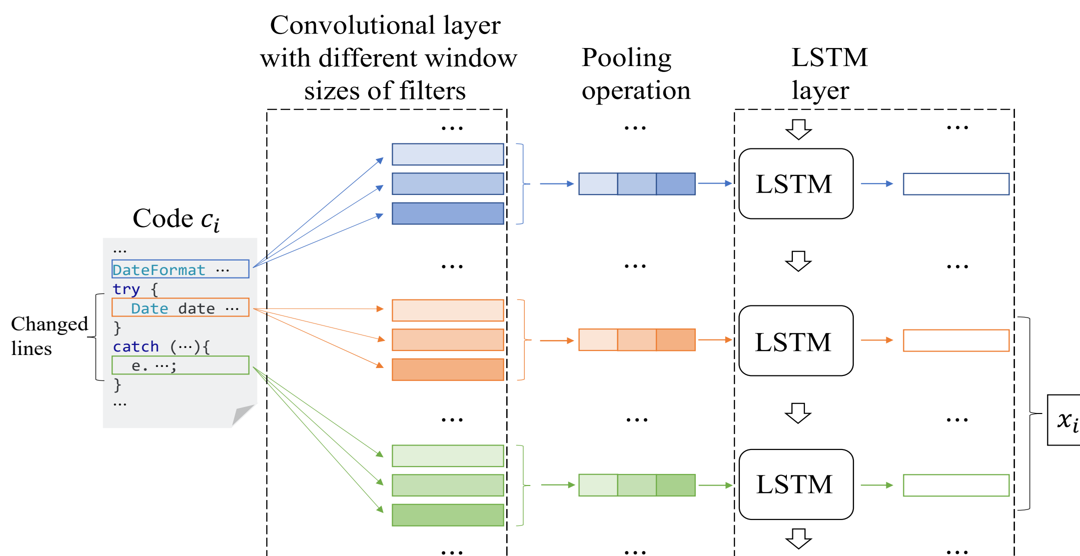

Figure 3: The structure for context embedding. The convolutional layer is used to extract semantics of a statement based on the tokens within the statement, the pooling layer aims to extract the most informative information, and the LSTM layer aims to capture contextual information. The output x_i is the context embedding of changed lines in code c_i.

The subsequent LSTM layer is employed to extract contextual information of code. LSTM is able to exploit sequential nature from source code to capture high-level semantic features because LSTM overcomes the difficulty in learning long-term dynamics. Since LSTM is specified for sequential inputs, the T feature maps generated by CNN can be fed into LSTM. At time t, LSTM takes the vector $\mathbf{z}_t$ as input, which represents the feature

maps of the t-th statement in the code. After processing from LSTM, the network generates T feature maps $\mathbf{h} \in \mathbb{R}^m$. By taking the L changed lines' representation as output, which contains the contextual and helpful information outflowed from memory cells of LSTM, we can obtain a richer feature representation that captures both local semantics within changed lines and the contextual semantics of the surrounding code. For a code pair (c_i^O, c_i^N), we process c_i^O and c_i^N separately and use (x_i^O, x_i^N) to denote the output of context embedding.

2.2. Exploiting Bidirectional Converting Deviation

In this subsection, we introduce the structure for exploiting bidirectional converting deviation, where a novel framework is designed to model the forward converting process and the backward converting process of code and learn the representations of code change.

Code changes can be mapped to some converting directions. The reason why a change improves the quality of code is that it follows some specific converting directions which lead to a better version of code. For instance, the example in Figure 1 improves the quality of code because it follows the converting direction that adds the exception handling mechanism to the source code. To model the process of code change and map the code change to the trace of converting process, a straightforward solution is only to consider the converting process from the code before change to the code after change, which can be solved by transforming the context embedding of old code to the context embedding of new code. However, such a solution may perform poorly because there is no constraint for the representation of the code after change, so it can change arbitrarily in the `After Change Space`. There are infinitely many representations of the code after change given the representation of code before change. Aligning the representation after the converting process to that arbitrarily changed representation causes that it is hard to optimize and the solution is intractable .

To address this problem, we consider adding more structure, where both the forward converting process from the `Before Change Space` to the `After Change Space` and the backward converting process inversely are modeled and the change representations for the bidirectional converting processes are exploited. Moreover, the converting directions which lead to good changes are captured by the model, which means that the converting processes modeled by the network are consistent with good changes. Specifically, for good changs, the forward converting process and the backward converting process are the inverse of each other, so they share the same change representation. Inversely, a bad change that does not fit the directions deviates from the good changes, and the forward converting process for the code before change and the backward converting process for the code after change are inconsistent, which results in that the distance between the two change representations is large. Therefore, the change can be evaluated by the bidirectional converting deviation which is measured by the distance between the change representation of the forward converting process and the representation of the backward converting process. The bidirectional converting processes with large deviation are more likely to be bad changes.

Based on the considerations mentioned above, we propose the structure responsible for exploiting bidirectional converting deviation, which is specified in Figure 4. The left part is to build a forward converting process for the code before change and a backward converting process for the code after change. The right part is used to reconstruct the code before

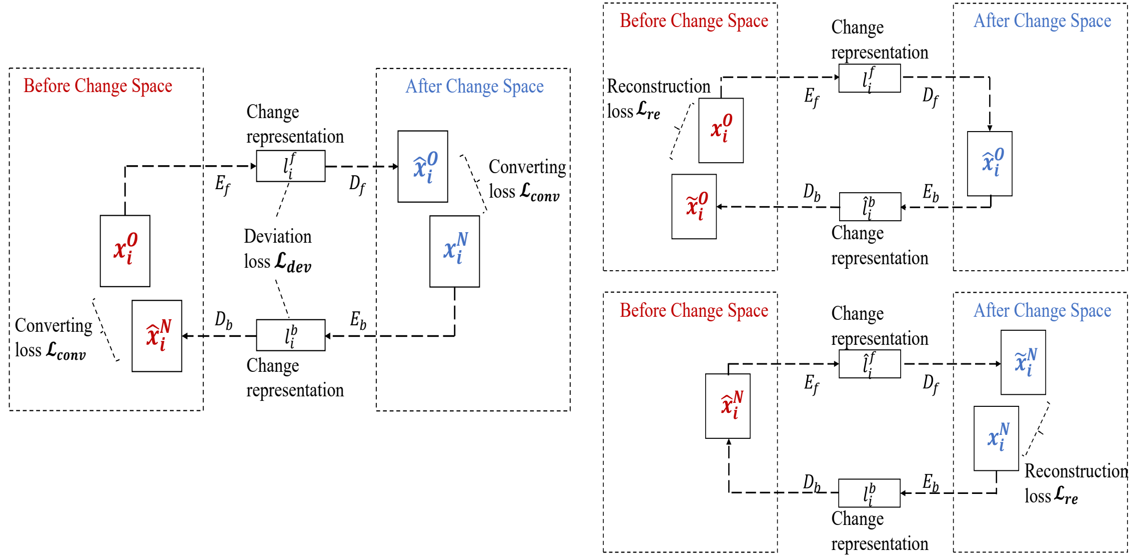

Figure 4: The structure for exploiting bidirectional converting deviation. The left part is to build a forward converting process from the context embedding of old code x_i^O to $\hat{x}_i^O$, and a backward converting process from the context embedding of new code x_i^N to $\hat{x}_i^N$. The right part is to reconstruct the code before change and after change respectively, where $\tilde{x}_i^O$,and $\tilde{x}_i^N$ are reconstructions.

change and the code after change respectively. The converting processes are modeled by autoencoder. The autoencoder for the forward converting process is composed of an encoder E_f and a decoder D_f. E_f is responsible for encoding x_i^O, the context embedding of old code, to a change representation l_i^f, which captures the key of the change. D_f is responsible for decoding from that representation to $\hat{x}_i^O$. Autoencoder can also be used to model the backward converting process. Analogously, let E_b denotes the encoder for the backward converting processes, D_b denotes the decoder, l_i^b denotes the change representation.

For all old code $x_i^O, i = 1, 2, \cdots, n$, we can get code $\hat{x}_i^O$, which is higher-quality than x_i^O, after the forward converting process which models the transformation from the code before good change to the code after good change. For all new code $x_i^N, i = 1, 2, \cdots, n$, we can get code $\hat{x}_i^N$, which is lower-quality than x_i^N, after the backward converting process which models the transformation from the code after good change to the code before good change.

Specifically, for a code pair (x_{i+}^O, x_{i+}^N), which means the change from x_{i+}^O to x_{i+}^N is good, the forward converting process modeld by the encoder E_f and the decoder D_f can be expressed as: $x_{i+}^O \xrightarrow{E_f} l_{i+}^f \xrightarrow{D_f} \hat{x}_{i+}^O$, while the backward converting process modeled by E_b and D_b can be expressed as: $\hat{x}_{i+}^N \xleftarrow{D_b} l_{i+}^b \xleftarrow{E_b} x_{i+}^N$. Since the forward converting process and the backward converting process are consistent with good changes, either the difference between $\hat{x}_{i+}^O$ and x_{i+}^N or the difference between $\hat{x}_{i+}^N$ and x_{i+}^O should be as small as possible.

Moreover, the distance between l_{i+}^f and l_{i+}^b should be as small as possibel because the two inverse processes share the same change representation.

Similarly, for a code pair (x_{i-}^O, x_{i-}^N), which means the change from x_{i-}^O to x_{i-}^N is low-quality, the forward converting process can be represented as: $x_{i-}^O \xrightarrow{E_f} l_{i-}^f \xrightarrow{D_f} \hat{x}_{i-}^O$, while the backward converting process can be expressed as: $\hat{x}_{i-}^N \xleftarrow{D_b} l_{i-}^b \xleftarrow{E_b} x_{i-}^N$. Since x_{i-}^N is lower-quality than x_{i-}^O, $\hat{x}_{i-}^O$ which is higher-quality than x_{i-}^O is not similar to x_{i-}^N. Furthermore, because the two converting processes are not the inverse of each other, and they do not share the same change representation, the distance between l_{i-}^f and l_{i-}^b should be large, which indicates that the change deviates from good changes.

We define the following loss functions:

$$\mathcal{L}_{conv} = \sum_{i+}[d(\hat{x}_{i+}^O, x_{i+}^N) + d(\hat{x}_{i+}^N, x_{i+}^O)], \tag{3}$$

$$\mathcal{L}_{dev+} = \sum_{i+} d(l_{i+}^f, l_{i+}^b), \tag{4}$$

$$\mathcal{L}_{dev-} = \sum_{i-} d(l_{i-}^f, l_{i-}^b), \tag{5}$$

$$\mathcal{L}_{dev} = \mathcal{L}_{dev+} + \gamma\mathcal{L}_{dev-}^{-1}, \tag{6}$$

where $i+ \in \{i|1 \le i \le n, y_i = +\}$, $i- \in \{i|1 \le i \le n, y_i = -\}$, $d(\cdot,\cdot)$ is the distance function defined by Equation 2. $\mathcal{L}_{conv}$ is the converting loss to ensure that the converting processes modeld by our model are consistent with good changes. $\mathcal{L}_{dev+}$ is the deviation loss to minimize the distance between l_{i+}^f and l_{i+}^b because they are the inverse of each other and they share the same change representation when the change is good. $\mathcal{L}_{dev-}$ is the deviation loss to maximize the distance between l_{i-}^f and l_{i-}^b because they do not share the same representation when the change is bad. $\mathcal{L}_{dev}$ is the combination of $\mathcal{L}_{dev+}$ and $\mathcal{L}_{dev-}$ in which minimize $\mathcal{L}_{dev-}^{-1}$ is equivalent to maximize $\mathcal{L}_{dev-}$, and γ is the trade-off parameter balancing these two terms.

Moreover, when we convert the code from the `Before Change Space` to the `After Change Space` and convert the code from the `After Change Space` to the `Before Change Space`, it should be able to reconstruct x_i^O after the process $x_i^O \xrightarrow{E_f} l_i^f \xrightarrow{D_f} \hat{x}_i^O \xrightarrow{E_b} \hat{l}_i^b \xrightarrow{D_b} \tilde{x}_i^O$, where $\tilde{x}_i^O$ is the reconstruction of x_i^O. Similarly, we denote $\tilde{x}_i^N$ as the reconstruction of x_i^N after the process $x_i^N \xrightarrow{E_b} l_i^b \xrightarrow{D_b} \hat{x}_i^N \xrightarrow{E_f} \hat{l}_i^f \xrightarrow{D_f} \tilde{x}_i^N$. The following reconstruction loss defined by Equation 7 should be small as possible:

$$\mathcal{L}_{re} = \sum_{i}[d(\tilde{x}_i^O, x_i^O) + d(\tilde{x}_i^N, x_i^N)]. \tag{7}$$

The reconstruction loss is independent of labels, therefore it is an unsupervised loss.

Specifically, the parameters of the structure for context embedding can be denoted as Θ_{CE}, and the parameters of the structure for exploiting bidirectional converting deviation can be denoted as Θ_{ECD}. The parameters $\Theta = \{\Theta_{CE}, \Theta_{ECD}\}$ of our model can be

learned by minimizing the following objective function defined by Equation 8 based on SGD (Stochastic Gradient Descent):

$$\min \mathcal{L} = \mathcal{L}_{conv} + \alpha\mathcal{L}_{dev} + \beta\mathcal{L}_{re}, \tag{8}$$

where $\mathcal{L}$ is the loss function implied in CCL and α, β are the trade-off parameters.

3. Experiments

To evaluate the effectiveness of the proposed method CCL, we conduct experiments on open source projects and compare the CCL with several methods.

3.1. Datasets and Experiment Settings

The code data used in the experiments is extracted from the publicly available dataset SOTorrent (Baltes et al., 2018), which has been widely used. SOTorrent is an open dataset based on the data from the official Stack Overflow data dump and the Google BigQuery GitHub dataset. SOTorrent provides access to all the version history of Stack Overflow content which has collected large quantities of code changes, therefore we choose SOTorrent as the datasets. We extract data about code changes from SOTorrent and divide our datasets into five repositories as shown in Table 1. The labels are obtained by votes of users. Namely, if users accept the answer for a code change, we regard the change as good. More details about the data can be referred in (Baltes et al., 2018). Our framework is not designed for a particular programming language. Although we conduct experiments on JAVA projects, the framework can be adapted and applied to the code change learning tasks with other programming languages (e.g., Python, C/C++, etc.).

Table 1: Statistics of our datasets. There are in total five different repositories. In each repository, the number of code pairs and the number of good changes are listed.

Datasets	# code pairs	# good changes
A	3,285	1,241
B	1,551	586
C	3,308	1,372
D	3,821	1,566
E	4,635	1,630

In the data preprocessing, we extract the information of the code change, for example, line numbers of the changed code lines. There are situations that the code change is only adding lines of code or deleting lines of code, which causes half of the code pair to be empty set and is unable to be fed into the network. Therefore, we only consider the situation that both the code before change and the code after change are nonempty set. To obtain word embeddings, first we split the each token in the code into subtokens (e.g., `ClassType` will be split into two subtokens `Class` and `Type`). The unseen subtokens will be replaced by `UNSEEN`. Then we use word2vec (Mikolov et al., 2013) to encode each subtoken in the raw data as vector, which has been shown effective in processing textual data and widely used.

We are concerned about whether the model can prioritize good changes over bad ones. Therefore we use AUC (Area Under ROC Curve) and Accuracy to measure the effectiveness of the proposed CCL model. The AUC value is equivalent to the probability that a randomly chosen positive example is ranked higher than a randomly chosen negative example (Fawcett, 2006). We use AUC to evaluate whether the model can choose the higher-quality code changes from large quantities of code.

Since there are no previous studies that employed machine learning methods in code change learning, we firstly compare CCL with several traditional models on software engineering. One of the most commonly used is to employ the Vector Space Model (VSM) to represent the source code and then train a classifier to predict (Gay et al., 2009). Furthermore, we assess whether the proposed method CCL can outperform the state-of-the-art software minging model DACE (Shi et al., 2019), which learns the correlation between the code before change and the code after change by packing them together. Finally, we evaluate whether the structure for exploiting bidirectional converting deviation and the loss functions proposed in Section 2.2 improve the performance by comparing the results with some variants of CCL.

The compared methods are shown below:

- TFIDF+LR (Gay et al., 2009): a model that uses TFIDF to represent the code before change and the code after change and concatenates the two vectors for predicting using Logistic Regression (LR) as the classifier.

- TFIDF+SVM: a model that uses TFIDF to represent code, which is the same as TFIDF+LR, and Support Vector Machine (SVM) is used for prediction.

- DACE (Shi et al., 2019): a state-of-the-art deep software mining model for code review, which summarizes information about the code before change and the code after change into a fixed-length vector to predict whether the change is approved.

- CCL^{-rcd}: a variant of CCL which only employs the structure for context embedding without modeling the bidirectional converting processes of code. Multilayer Perceptron (MLP) is employed for prediction.

- CCL^{-r}: a variant of CCL which does not consider the unsupervised reconstruction loss $\mathcal{L}_{re}$, and only use the converting loss $\mathcal{L}_{conv}$ and the deviation loss $\mathcal{L}_{dev}$.

- CCL^{-c}: a variant of CCL without considering the converting loss $\mathcal{L}_{conv}$, and only use the reconstruction loss $\mathcal{L}_{re}$ and the deviation loss $\mathcal{L}_{dev}$.

- CCL^{-rd}: a variant of CCL which only models the forward converting process without the backward converting process and does not employ the reconstruction loss $\mathcal{L}_{re}$ and the deviation loss $\mathcal{L}_{conv}$. It uses the distance between $\hat{x_i}^O$ and x_i^N to evaluate the change.

For TFIDF+LR and TFIDF+SVM, we consider the top 300 frequency words in all source code when calculating TFIDF. For DACE, we follow the same parameter settings as suggested in (Shi et al., 2019). For CCL and its variants, we employ the most commonly used ReLU $\sigma(x) = max(x, 0)$ as active function, the filter windows size in CNN is set as 2,

3, 4, with 100 feature maps each, the number of neuron dimension in LSTM is set as 300, and the encoder and decoder used in the network are GRUs with the cell size of 320.

3.2. Experiment Results

The experimental results are clearly shown in this section. In our experiments, 10-fold cross validation is repeated 10 times for each dataset. We also conduct Mann-Whitney U-test at 95% confidence level to evaluate the significance of CCL. In each result table, if the proposed method CCL significantly outperforms a compared method, the inferior performance of the compared method would be marked with "$\circ$", and the performance would be marked with "$\bullet$" if it is significantly better than CCL. Besides, the best performance on each data set is boldfaced in all tables.

3.2.1. Experiment Results Compared with Other Methods

We compare CCL with several traditional methods and the state-of-the-art method DACE (Shi et al., 2019). The average performance of the compared methods with respect to AUC and Accuracy are displayed in Table 2 and Table 3 respectively.

Table 2: Experiment results compared with other methods in terms of AUC on all datasets.

Method	A	B	C	D	E	Avg.
TFIDF+LR	0.553∘	0.503∘	0.524∘	0.597∘	0.573∘	0.550
TFIDF+SVM	0.564∘	0.542∘	0.503∘	0.635∘	0.614∘	0.572
DACE	0.807	0.893∘	0.753∘	0.848∘	0.890∘	0.838
CCL	**0.808**	**0.924**	**0.795**	**0.869**	**0.901**	**0.859**

Table 3: Experiment results compared with other methods in terms of Accuracy on all datasets.

Method	A	B	C	D	E	Avg.
TFIDF+LR	0.610∘	0.621∘	0.599∘	0.661∘	0.698∘	0.638
TFIDF+SVM	0.620∘	0.651∘	0.584∘	0.699∘	0.728∘	0.656
DACE	0.715∘	0.826∘	0.668∘	0.767∘	0.799∘	0.755
CCL	**0.745**	**0.872**	**0.708**	**0.789**	**0.821**	**0.787**

Compared with traditional methods TFIDF+LR and TFIDF+SVM, CCL improves the average AUC of TFIDF+LR by 56.2%, the average AUC of TFIDF+SVM by 50.2%, and CCL improves the average Accuracy of TFIDF+LR by 23.4%, the average Accuracy of TFIDF+SVM by 20.0%. It is because that CCL costs more time and space to extract features of code and model the change process of code. TFIDF technique which is used for representing code cannot capture the local and contextual information about the functional semantics of the code, which is significant for the code representation. Without a particular design to capture the semantics in the lexicon and program structure and the information

about the code change, it will suffer from the loss of information, which causes a poor performance for code change learning. These results suggest that the proposed method CCL can outperform traditional models.

Compared with the state-of-the-art method DACE, CCL also improves the average AUC of DACE by 2.5% and improves the average Accuracy of DACE by 4.2%, which can be regarded that only summarizing information into a fixed-length vector that indirectly represents the change is under-representation for code changes, and is difficult to capture diverse changes efficiently. The fixed-length vector simply packs all relevant and irrelevant information about the change and it leaves a heavier burden on the following classifiers to capture the key information of the change. Modeling the change process directly and exploiting the deviation of the bidirectional converting processes are more effective for code change learning, thus leading to better performance. The results indicate that CCL can extract the key of the code change and can significantly outperform the state-of-the-art software mining method DACE.

3.2.2. Experiment Results Compared with Variants of CCL

We further evaluate the effectiveness of the structure for exploiting bidirectional converting direction, which is the key part of the CCL model. The comparison results in terms of AUC and Accuracy are shown in Table 4 and Table 5 respectively. It can be observed that CCL outperforms CCL^{-rcd}, which is a variant of CCL without modeling bidirectional converting processes of code and only concatenating context embedding of the code before change and the code after change for prediction, 6.8% in terms of Accuracy and 6.3% in

Table 4: Experiment results in terms of AUC compared with variants of CCL on all datasets.

Method	A	B	C	D	E	Avg.
CCL^{-rcd}	0.711∘	0.903∘	0.697∘	0.853∘	0.875∘	0.808
CCL^{-r}	0.757∘	0.856∘	0.749∘	0.864	0.861∘	0.817
CCL^{-c}	0.735∘	0.878∘	0.721∘	0.816∘	0.854∘	0.801
CCL^{-rd}	0.608∘	0.875∘	0.690∘	0.783∘	0.873∘	0.766
CCL	**0.808**	**0.924**	**0.795**	**0.869**	**0.901**	**0.859**

Table 5: Experiment results in terms of Accuracy compared with variants of CCL on all datasets.

Method	A	B	C	D	E	Avg.
CCL^{-rcd}	0.687∘	0.814∘	0.671∘	0.712∘	0.799∘	0.737
CCL^{-r}	0.709∘	0.821∘	0.672∘	0.788	0.808∘	0.760
CCL^{-c}	0.670∘	0.808∘	0.663∘	0.747∘	0.813	0.740
CCL^{-rd}	0.640∘	0.795∘	0.669∘	0.731∘	0.793∘	0.726
CCL	**0.745**	**0.872**	**0.708**	**0.789**	**0.821**	**0.787**

terms of AUC. The results suggest that although the model CCL^{-rcd} also employs CNN and LSTM to extract the features of code, CCL^{-rcd} does not capture the key information of change, and only concatenating the context embedding of the code before change and the code after change is insufficient for code change learning. The CCL model captures the key for evaluating code changes by modeling the process of code change.

Moreover, to evaluate the effectiveness of the loss functions defined in CCL, we compare CCL with some other variants: CCL^{-r}, CCL^{-c} and CCL^{-rd}. It can be noticed from Table 4 and Table 5 that CCL achieves the best performance among the compared methods on all datasets. CCL improves the average AUC of CCL^{-r} by 5.1% and CCL improves the average Accuracy of CCL^{-r} by 3.6%. It means that the unsupervised reconstruction loss $\mathcal{L}_{re}$ is helpful because it ensures that after a forward and a backward converting process, the input can be reconstructed, which is a kind of regularization for the network. Besides, CCL outperforms CCL^{-c} 7.2% in terms of AUC and 6.4% in terms of Accuracy. It indicates that the converting loss $\mathcal{L}_{conv}$ makes sense because it guarantees that the distance between $\hat{x}_{i+}^{O}$ and x_{i+}^{N} should be as small as possible, which means that the converting processes model by the network are consistent with good changes. Compared with CCL^{-rd}, CCL improves the AUC by 12.1% and the Accuracy by 8.4%. CCL^{-rd} does not consider the backward converting process and only uses the distance between $\hat{x}_i{}^{O}$ and x_i^{N} to predict. It causes that there is no constraint for the representation of the code after change so it can change arbitrarily in the `After Change Space`. There are infinitely many representations of the code after change given the representation of code before change. Therefore, there are infinitely many mappings that can model the change from x_i^{O} to x_i^{N} and the solution is intractable and hard to optimize.

In summary, the experimental results demonstrate the effectiveness in modeling the bidirectional converting processes of source code and exploiting the converting deviation to improve code change learning.

4. Related Work

Many previous empirical studies aimed to help researchers and practitioners to understand code changes from different perspectives. For example, Tao et al. (2012) indicated that understanding code change is an indispensable task performed by engineers in software development process. Nguyen et al. (2013) indicated that repetitiveness of changes could be as high as 70-100% at small sizes and learning code changes and recommending them in software evolution is beneficial. Ray et al. (2015) presented an empirical study of unique changes and investigated where they occur along the project. Different from those previous studies, we focus on modeling the change process of code and evaluating the quality of the changes automatically, and we design a deep learning approach for code change learning.

To maintain software quality assurance, many previous studies have focused on the evolution of software systems. For example, many studies aimed to improve the effectiveness of code review, which is the process of manual inspection of code changes. Kononenko et al. (2015) investigated code review quality and explored the relationships between the code inspections and a set of personal factors. Zanjani et al. (2016) presented an approach to automatically recommend reviewers who are the best suited to participate in a given review. There are also some previous studies using deep learning methods to improve the

effectiveness of code review. For example, Shi et al. (2019) employed a special network to learn the revision feature which summarizes all information into a vector that indirectly represents the change. Besides, many studies focus on just-in-time defect prediction, which aims to be an earlier step of continuous quality control because it can be invoked as soon as a developer commits code. Kamei et al. (2013) indicated that just-in-time defect prediction provides an effort-reducing way to focus on the riskiest changes and thus reduce the costs of developing high-quality software. Another related task is called continuous build outcome prediction, which aims to predict whether a build will pass or fail upon the software change is committed. Hassan and Zhang (2006) employed a tree-based method and statistical information about code and commits to predict the result of a build. Xie and Li (2018) proposed a semi-supervised online AUC optimization method for build outcome prediction with a group of features. To the best of our knowledge, we are the first to directly model the change process of code, and we propose a general deep framework for learning the software code change.

Recently, deep learning has been applied in many software engineering problems. For example, Wang et al. (2016) leveraged the Deep Belief Network (DBN) to automatically learn semantic features of code for defect prediction. Tan et al. (2015) studied the online defect prediction problem for imbalanced data. White et al. (2015) proposed a novel deep learning model for code suggestion. Mou et al. (2016) designed a novel tree-based convolutional neural network for the programming language, which is effective in several software problems. Huo and Li (2017) combined LSTM and CNN to learn unified features from bug reports and source code for bug localization tasks. White et al. (2016) used autoencoder to learn latent features for source codes and designed learning-based clone detection techniques.

5. Conclusion

In this paper, we highlight that code change learning is a major task in software mining and propose a novel deep method called CCL to directly model the change process of code. In CCL, the local and contextual information about code is extracted by CNN and LSTM, subsequently a forward converting process and a backward converting process are designed for capturing the key of code change, and the deviation of the bidirectional converting processes is exploited to evaluate the change. Experimental results on five open source repositories indicate that modeling the bidirectional converting processes and exploiting the converting deviation are beneficial, and the CCL approach significantly outperforms the compared methods in code change learning.

In the future, incorporating additional information such as descriptions of code changes written in natural language to enrich the structure of CCL will be investigated. Additionally, extending CCL to more tasks where code changes often will be another meaningful future work.

Acknowledgments

This research was supported by National Key Research and Development Program (2017YFB1001903) and NSFC (61921006).

References

Robert S Arnold. *Software change impact analysis.* IEEE Computer Society Press, 1996.

Sebastian Baltes, Lorik Dumani, Christoph Treude, and Stephan Diehl. Sotorrent: reconstructing and analyzing the evolution of stack overflow posts. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 319–330, 2018.

Tom Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006.

Gregory Gay, Sonia Haiduc, Andrian Marcus, and Tim Menzies. On the use of relevance feedback in ir-based concept location. In *Proceedings of the 25th IEEE International Conference on Software Maintenance*, pages 351–360, 2009.

Ahmed E. Hassan and Ken Zhang. Using decision trees to predict the certification result of a build. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 189–198, 2006.

Xuan Huo and Ming Li. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 1909–1915, 2017.

Xuan Huo, Ming Li, and Zhi-Hua Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, pages 1606–1612, 2016.

Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. DECKARD: scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, pages 96–105, 2007.

Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineerin*, 39(6):757–773, 2013.

Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W. Godfrey. Investigating code review quality: Do people and participation matter? In *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution*, pages 111–120, 2015.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems 26*, pages 3111–3119, 2013.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, pages 1287–1293, 2016.

Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, and Hridesh Rajan. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 180–190, 2013.

Baishakhi Ray, Meiyappan Nagappan, Christian Bird, Nachiappan Nagappan, and Thomas Zimmermann. The uniqueness of changes: Characteristics and applications. In *Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories*, pages 34–44, 2015.

Shu-Ting Shi, Ming Li, David Lo, Ferdian Thung, and Xuan Huo. Automatic code review by learning the revision of source code. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, pages 4910–4917, 2019.

Ian Sommerville. *Software engineering, 8th Edition*. Addison-Wesley, 2007.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27*, pages 3104–3112, 2014.

Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering*, pages 99–108, 2015.

Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes?: an exploratory study in industry. In *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, page 51, 2012.

Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, pages 297–308, 2016.

Martin White, Christopher Vendome, Mario Linares Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories*, pages 334–345, 2015.

Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98, 2016.

Zheng Xie and Ming Li. Cutting the software building efforts in continuous integration by semi-supervised online AUC optimization. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 2875–2881, 2018.

Motahareh Bahrami Zanjani, Huzefa H. Kagdi, and Christian Bird. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering*, 42(6):530–543, 2016.