Learning Uniform Semantic Features for Natural Language and Programming Language Globally, Locally and Sequentially

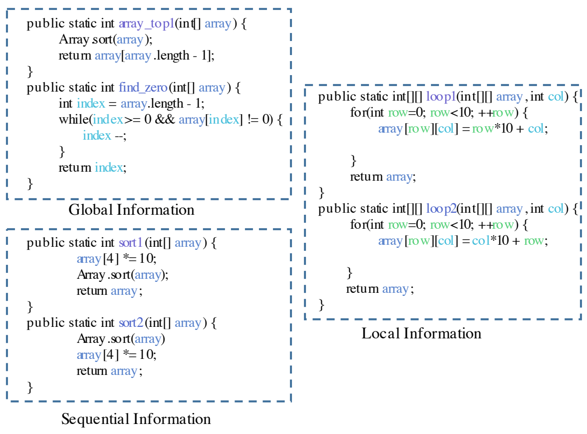
Yudong Zhang,[1] **Wenhao Zheng,**[1,3] **Ming Li**[1,2]
[1]National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, 210023, China
[2]Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing, 210023, China
[3]Search Product Center, WeChat Search Application Department, Tencent, China
yudongzhang@smail.nju.edu.cn, wenhaozheng@tencent.com, lim@nju.edu.cn

Abstract

Semantic feature learning for natural language and programming language is a preliminary step in addressing many software mining tasks. Many existing methods leverage information in lexicon and syntax to learn features for textual data. However, such information is inadequate to represent the entire semantics in either text sentence or code snippet. This motivates us to propose a new approach to learn semantic features for both languages, through extracting three levels of information, namely global, local and sequential information, from textual data. For tasks involving both modalities, we project the data of both types into a uniform feature space so that the complementary knowledge in between can be utilized in their representation. In this paper, we build a novel and general-purpose feature learning framework called UniEmbed, to uniformly learn comprehensive semantic representation for both natural language and programming language. Experimental results on three real-world software mining tasks show that UniEmbed outperforms state-of-the-art models in feature learning and prove the capacity and effectiveness of our model.

Introduction

In recent years, semantic representation learning (Bengio, Courville, and Vincent 2013) for languages has gained much attention and been widely applied to many software mining tasks. Among those tasks, natural language and programming language are two common data types, which require feature extraction as a preliminary step so that their representation can be processed by specific models. The extracted features have been used to address tasks such as bug localization (Huo, Li, and Zhou 2016; Huo and Li 2017; Hoang et al. 2018), code clone detection (White et al. 2016; Wei and Li 2017; 2018), code summarization (Allamanis, Peng, and Sutton 2016), etc. However, one main factor that determines the capacity of representation lies in the information exploited from data, since partial semantic information can result in features of poor quality, and thus prevent models from elevating performance. Therefore, figuring out a way to learn comprehensive semantic features for natural language and programming language has become a vital problem we need to resolve.



```
public static int array_top1(int[] array) {
        Array sort(array);
        return array[array .length - 1];
}
public static int find_zero(int[] array) {
        int index = array.length - 1;
        while(index>= 0 && array[index] != 0) {
              index --;
        }
        return index;
}
```

```
public static int sort1(int[] array) {
            array[4] *= 10;
            Array .sort(array);
            return array;
}
public static int sort2(int[] array) {
            Array .sort(array)
            array[4] *= 10;
            return array;
}
```

```
public static int[][] loop1(int[][] array , int col) {
        for(int row=0; row<10; ++row) {
              array [row][col] = row*10 + col;

        }
        return array;
}
public static int[][] loop2(int[][] array , int col) {
        for(int row=0; row<10; ++row) {
              array [row][col] = col*10 + row;

        }
        return array;
}
```

Figure 1: Examples of Java code snippet pairs differing in each level of information

As we observe, natural language and programming language, in spite of different modalities, uniformly share three levels of semantic information, namely global, local and sequential information. The three levels of information depict semantic meaning of textual contents from different aspects. In order to give a clean explanation below, we provide detailed instances and discussions to separately illustrate those semantic information in both languages.

In natural language, global information refers to the aggregation of every term's meaning within the sentence globally. All terms contribute to the sentence's semantics, while replacing any term can probably cause global information shift. For example, sentences as *"write a program in Python"* and *"debug the Java program"* are discussing disparate things because of different lexicons in the two sentences. Local information is implied in a local small group of words standing together as a conceptual unit, like the phrase *"Depth First Search"*. Only by considering those terms as an integrity rather than separate words can we understand it. Sequential information is included in the logic order of terms forming the sentence. For instance, changing the order of terms in *"apply quick sort to the array"* to *"quick to apply array the sort"* will alter the semantic meaning and make the sentence not understandable.

Similar to natural language, programming language also contains three levels of information. Its semantics relates to the functionality of the code snippets. As the examples shown in Figure 1, function `array_top_1` and `find_zero` are manifestly different in the variables and terms within their bodies. This causes disparity in their global information and meanwhile leads to functionality difference. In the local information example, function `loop2` changes the substructures in the *for-loop* in `loop1` while it remains all the terms and the organization of statements (global and sequential information) unshifted. Nevertheless, the behaviors of the two functions differ due to the nuance in local information. In the sequential information example, two statements have reversed positions in `sort2` comparing with `sort1`. As a result, the dependencies between statements (sequential information) are changed and the functionality is mutated, despite the fact that the terms and substructures (global and local information) are nearly the same in two functions.

According to the discussions above, we observe that (1) both natural language and programming language contain three levels of information, i.e. global, local and sequential information, all of which are indispensable for understanding the semantic meaning; (2) the three levels of information are complementary rather than implicit to each other, as changing one level of information while keeping the other two can still cause semantic shift.

Many previous works focused on learning representation for both languages in software mining tasks by extracting semantic information. (Huo and Li 2017; Huo, Li, and Zhou 2016) model bug reports and source code by leveraging local and sequential information in their representation to address bug localization task, (Wei and Li 2017) learns hash code as code snippet's representation by exploiting global and sequential information, while (Allamanis, Peng, and Sutton 2016) generates features based only on local information to encode code snippet. However, those existing models failed to take all of the aforementioned three levels of information into account, so that their learned representations would cover only partial semantics.

In order to resolve this problem, we propose a novel feature learning framework UniEmbed to learn representations for both natural language and programming language, by exploiting global, local and sequential information in the textual contents. In particular, for certain software mining tasks involving both modalities, such as code annotation and bug localization, where text sentence and code snippet of a pair are semantically relevant, we project the representations of those cross-modal data pairs into a uniform feature space. Such approach assures that the complementary knowledge in between can be leveraged during learning process, as stated in (Kim et al. 2018), and alleviates *semantic gap* between modalities as well. Experimental results on three real-world software mining tasks demonstrate the capacity and effectiveness of UniEmbed, which significantly outperforms state-of-the-art feature learning methods in those tasks.

The contributions of our work are twofold:

- We introduce a approach to extract three levels of semantic information, namely global, local and sequential information, in learning features for textual data. Those learned representations can comprehensively cover the semantics in both natural language and programming language. To the best of our knowledge, no previous works adopt similar strategy in their feature learning method or achieve comparable performance as our framework.
- We propose a novel representation learning framework UniEmbed, which can learn uniform features for natural language and programming language when both modalities are involved in the same task. Such representations are capable and effective in helping address a set of real-world software mining tasks.

Related Work

Most of existing research involving feature learning in software mining tasks has focused on various methods to extract semantic information from textual data. (Lukins, Kraft, and Etzkorn 2010) considers each word with certain probabilities in bug reports and applies Latent Dirichlet Allocation (LDA) model for locating buggy files, (Gay et al. 2009) represents bug reports and source code files as feature vectors based on concept localization using Vector Space Model (VSM). More recently, besides global information (lexicon), local information (structure) within textual content is also taken into consideration. (Allamanis, Peng, and Sutton 2016) parses code token groups to extract local information with Convolutional Attention Model, (White et al. 2016; Wei and Li 2017) exploit both lexicons and syntax structures in learning features for source code. (Huo and Li 2017) even considers long-term dependencies in code snippet and thus combines sequential information in representation. However, all these models leverage only one or two types among global, local and sequential information, which may result in incomplete semantics in learned features.

For software mining tasks involving both natural language and programming language, such as bug localization and code annotation, it is prevailing to treat both modalities differently by learning representation in separate feature spaces among previous works. (Huo, Li, and Zhou 2016) encodes bug reports and source code through different structures and only fuse their features right before classification. (Hoang et al. 2018; Tantithamthavorn et al. 2018) directly calculate similarity scores between bug reports and code snippets after separately encoding and include them as a component of the data pair's combined features. However, treating text sentence and code snippet differently can neither well resolve the *semantic gap* between two modalities nor make use of the complementary knowledge in between. Our proposed model resolves this problem by projecting data of both types into a uniform feature space.

The Proposed Framework: UniEmbed

In this section, we describe the details of our proposed model. We first give some notations which will be used in the following sections. Given a data triple (e, p, n), we consider $e, p, n \in T \cup C$, where T and C stand for the sets of raw text sentences and code snippets (We use text sentences and code snippets to represent natural language and

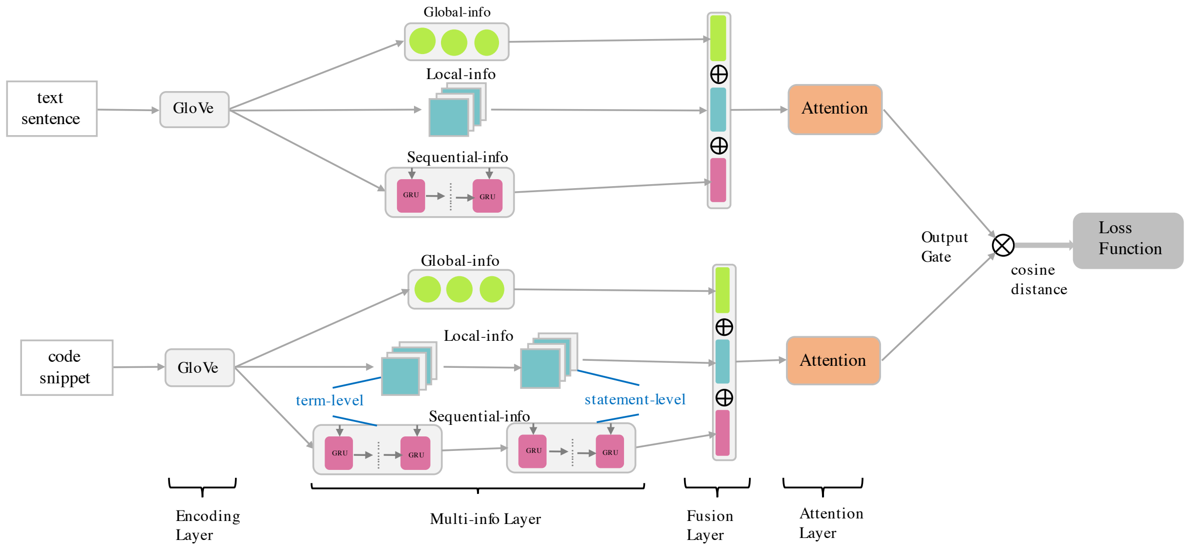


Figure 2: The Overall Structure of UniEmbed

programming language respectively). We define (e, p) as a similar data pair where e and p have similar semantic meanings, and (e, n) as a dissimilar data pair where e and n are semantically irrelevant. Note that e and p, e and n can come from different modalities, while p and n should belong to the same modality. We define $R(e)$ as the learned feature vector of e. The feature space where we project our data is a cosine distance-based space. Given a data pair (e_1, e_2), where $e_1, e_2 \in T \cup C$, we measure the similarity between e_1 and e_2 based on the cosine distance between their feature vectors $R(e_1)$ and $R(e_2)$ in the feature space. We note the cosine distance between two vectors as $D_c(\cdot, \cdot)$, which can be formalized as follows:

$$\begin{aligned} D_c(e_1, e_2) =& 1 - \cos\langle R(e_1), R(e_2)\rangle \\ =& 1 - \frac{e_1 \cdot e_2}{\|e_1\| \, \|e_2\|} \end{aligned} \tag{1}$$

General Framework

The structure of UniEmbed is shown in Figure 2. The whole framework is composed of two separate pipelines with similar structures. One is used to extract semantic features for text sentence while another one is for code snippet. We name them as *Textnet* and *Codenet*. Each pipeline contains four parts: encoding layer, multi-info layer, fusion layer, and attention layer. At the end of the framework, an output gate connects both pipelines and transforms the feature vectors learned in *Textnet* and *Codenet* to the cosine distance used for further similarity measurement.

UniEmbed takes raw text sentence and code snippet as input to corresponding pipeline. In the encoding layer, each term in text sentence or code snippet is encoded with the standard GloVe vectors (Pennington, Socher, and Manning 2014) into 300-dimensional representation. Then three levels of semantic information, namely **global**, **local** and **sequential** information, are extracted in multi-info layer with different architectures. We fuse the three levels of features into a multi-information representation by concatenating them in the order of [global, local, sequential] in the fusion layer. Finally, the attention layer learns different importance weights for each level of features and outputs the final representation, which is the projection of raw data in the feature space. Note that the structure of *Textnet* and *Codenet* is slightly different due to the discrepancy between text sentence and code snippet in their characteristics, which will be elaborated later.

Multi-information extraction

Multi-info layer consists of three differently structured network branches, which are used for exploiting global, local and sequential features accordingly. We name the three branches as *global-info* branch, *local-info* branch and *sequential-info* branch.

***Global-info* branch** Since understanding the semantic meaning of text sentence or code snippet depends on the information conveyed in every term, we follow the philosophy of *fastText* model (Joulin et al. 2016) to summarize the global information by averaging each term's representation.

We feed each term into the same fully-connected neural network to exploit its semantic feature vector separately. Then we average all terms' feature vectors by calculating their mean vector. The mean vector contains summarized information of every term and constructs the global feature vector of text sentence or code snippet. We note the global feature vector of the input e as $G(e)$, the i^{th} term in e as $term_i$, and the fully-connected neural network as $FC(\cdot)$, the whole process can be formalized as:

$$G(e) = \frac{\sum_i FC(term_i)}{\#\text{terms}} \tag{2}$$

***Local-info* branch** Local information is usually covered in a small specific region of the textual content, for instance, a phrase in text sentence or a *for-loop* substructure in code snippet. In order to extract local information from the organized group of terms, we need to consider those terms as an integrity instead of understanding their meanings term-by-term. Regarding the properties above, we employ CNN-based neural network to extract local information. As each convolutional operation only looks into a small region and generates corresponding local features at a time, it perfectly coordinates the characteristics of local information.

Using CNN to extract features for natural language has been widely studied (Johnson and Zhang 2014), thus we follow the standard approach to process text sentence in *Textnet* pipeline. However, programming language differs from natural language in two aspects, which are the *atomicity* of statements in semantics and the ”structured” organization of statements groups (Huo, Li, and Zhou 2016). So we should follow the nature of program structure defined by programming language when selecting CNN configurations to extract local features from code snippet in *Codenet* pipeline.

We design our *local-info* branch for code snippet using two-level convolutional neural networks, namely term-level and statement-level CNN. The network structure is demonstrated in Figure 3. Term-level CNN and its pooling layer learns the semantics of a single statement based on the terms it contains, while statement-level CNN and its pooling layer extracts local information from the interaction between statements with respect to program structure. At the end of *local-info* branch, a fully-connected neural network is applied to refine the learned features and maintain the dimensions. We note the local feature vector of the input e as $L(e)$.

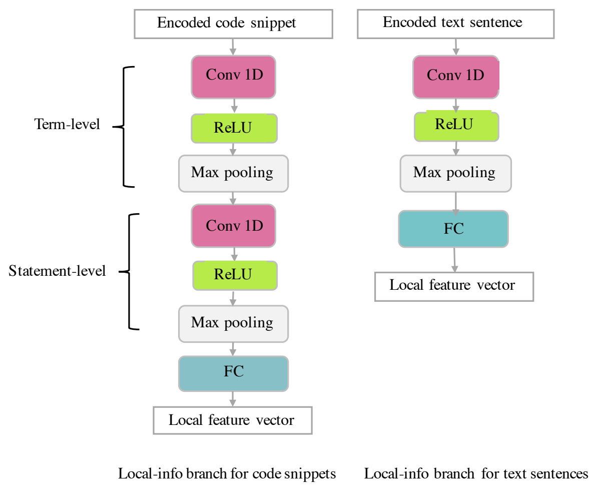
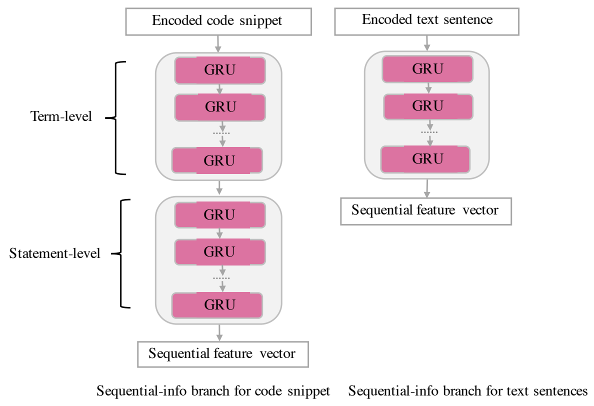


Figure 3: The structure of *local-info* branch

***Sequential-info* branch** Sequential information lies in the logic order of the components organized in data, for example, the order of terms in text sentence or the sequential dependencies between statements in code snippet. In order to extract sequential information from the data, we employ GRU-based (Chung et al. 2014) models to learn such features.

The structure of *sequential-info* branch is shown in Figure 4. With similar consideration of the discrepancy between characteristics of natural language and programming language, we apply standard GRU for text sentence, but term-level GRU and statement-level GRU for code snippet to learn sequential information first from the order of terms in single statement, and then from the interaction between statements based on program structure. We note the sequential feature vector with respect to the input e as $S(e)$.



Encoded code snippet
Encoded text sentence
Term-level
GRU
GRU
GRU
Statement-level
GRU
GRU
GRU
Sequential feature vector
GRU
GRU
GRU
Sequential feature vector
Sequential-info branch for code snippet
Sequential-info branch for text sentences

Figure 4: The structure of *sequential-info* branch

Feature fusion After extracting global, local and sequential information with *global-info* branch, *local-info* branch and *sequential-info* branch, we concatenate the three learned feature vectors in the order of [global, local, sequential] in the fusion layer to construct multi-information features of text sentence or code snippet.

Attention mechanism

Although all three levels of information are necessary in conveying semantic meaning, the portion of contribution of each information can differ. However, the multi-information features learned in previous layers treat global, local and sequential information equally, which may not be the gold representation.

To address this problem, we apply attention mechanism to learn different importance weights for each level of features. Due to different characteristics of text sentence and code snippet, the importance of each information may also differ between the two modalities. Thus we use separate attention layers to learn feature weights in *Textnet* and *Codenet* accordingly.

We note the k_{th} feature vector in multi-information features as $\mathrm{F}_k(e)$, where $\mathrm{F}_k(e) \in \{G(e), L(e), S(e)\}$ with respect to the input e. $\mathrm{WF}_k(e)$ stands for the weighted $\mathrm{F}_k(e)$ learned by attention layer, and $\mathrm{NWF}_k(e)$ stands for normalized $\mathrm{WF}_k(e)$. W_k is the learnable weights for each information. The attention mechanism can be formalized as follows:

$$\mathrm{WF}_k(e) = W_k * \mathrm{F}_k(e)$$
$$\mathrm{NWF}_k(e) = \frac{\exp(\mathrm{WF}_k(e))}{\sum_{i=1}^{3} \exp(\mathrm{WF}_i(e))} \quad (3)$$

After weighting each information with different importance, our final representation of input e can be specified as:

$$R(e) = [\mathrm{NWF}_1(e), \mathrm{NWF}_2(e), \mathrm{NWF}_3(e)], \quad (4)$$

Where $\mathrm{NWF}_1(e)$, $\mathrm{NWF}_2(e)$ and $\mathrm{NWF}_3(e)$ refer to the normalized weighted features with respect to $G(e)$, $L(e)$ and $S(e)$.

Similarity measurement

We attempt to learn semantic features for text sentence and code snippet in a mutual cosine distance-based feature space, where we measure the similarity between two representation vectors according to their cosine distance. Smaller cosine distance means higher similarity between a data pair.

Subsequently, we design a vanilla linear classifier and select the cosine distance, which is a scalar, as the only classification feature to predict whether a data pair is similar or not. We apply Cross-entropy loss to optimize the classifier. Give a data pair (e_1, e_2), the prediction process can be specified as:

$$\text{prediction} = \text{sigmoid}(a * D_C(e_1, e_2) + b) \quad (5)$$

Where a, b are learnable parameters.

Since our classifier is extremely simple, the performance of prediction heavily depends on how distinguishing the cosine distances between similar data pairs and dissimilar data pairs are. In return, high performance can empirically prove the capacity and effectiveness of UniEmbed in learning semantic features for text sentence and code snippet.

Optimization

It is possible that the importance weights for text sentence and code snippet can differ, which may result in scale difference between their representation. Such discrepancy in magnitude increases the difficulty to learn a unified absolute distance distribution over their feature vectors. The scale gap between the cosine distance of cross-modal data pair (text-code pair) and of single-modal data pair (text-text pair or code-code pair) can be hard to eliminate during learning process.

With the consideration above, we propose to learn a uniform relative distance distribution using triplet loss (Schroff, Kalenichenko, and Philbin 2015). By doing so we can optimize the cosine distance of single-modal and of cross-modal data pairs without worrying about scale difference. In triplet loss, we maximize the gap between cosine distance of similar pairs and of dissimilar pairs until it reaches a specific margin α. Given a data triple (e, p, n), we optimize our UniEmbed model by minimizing the following objective function:

$$\mathrm{Loss}(e, p, n) = \max(D_c(e, p) - D_c(e, n) + \alpha, 0) \quad (6)$$

Training

With respect to different real-world software mining tasks, we propose two training strategies: single training and joint training.

Single training: It learns representation for single task. This is a common case when we need to address a specific software mining task. For tasks involving single modality, such as code clone detection, we only train the corresponding module. While for tasks involving cross-modal data, like bug localization, we train both modules (*Textnet* and *Codenet*) synchronically.

Joint training: It learns uniform features for a set of relevant tasks simultaneously. There are some cases in real world when we need to resolve multiple related tasks at the same time. For example, we want to address duplicate programming question retrieval, code annotation and code clone detection, where both modalities are involved in data. Under such circumstances, we can apply joint training. Firstly we pre-train *Textnet* on duplicate question data (single-modal). Then we tune down the learning rate for *Textnet* while remain it unchanged for *Codenet*, and train both modules on code annotation data (cross-modal). Finally we tune down the learning rate to train *Codenet* on code clone detection data (single-modal). We use triplet loss with the same margin value to optimize our model in all tasks. Rich knowledge in different tasks' datasets is shared during joint training process, which enables model to learn more informative features and perform better on all these tasks than under single training.

Experiment

Datasets

In this section, we evaluate our UniEmbed model on three datasets, covering three relevant tasks, which are code clone detection, code annotation, and duplicated programming question retrieval.

All three datasets are collected from StackOverflow website. The duplicated programming question dataset (abridged as *dup-question*) consists of programming question pairs labelled as duplicated by users. We extract the code snippets within the best answer and label the respective question as annotation to the code. Code snippet and its annotation are matched as a pair, and such pairs compose of the code annotation dataset (abridged as *code-anno*). We regard the best answers of duplicated questions as similar answers, thus the code snippets extracted from such answer pairs are considered as cloned code, which belongs to Type-4 clone according to (Sridhara et al. 2010). The cloned code pairs are included in the code clone detection dataset (abridged as *code-clone*). Note that all code snippets are in Java programming language. The original datasets are available on Stack Exchange website[1].

Since all the collected data are similar pairs, we generate the same amount of dissimilar pairs by randomly sampling on text sentence set T ($\sim$720k) and code snippet set C

[1]https://archive.org/details/stackexchange

Table 1: Evaluation results on multiple software mining tasks

	dup-question		*code-clone*		*code-anno*	
Methods	AUC	F-measure	AUC	F-measure	AUC	F-measure
SourcererCC	–	–	53.21%	4.80%	–	–
CDLH	–	–	62.03%	59.89%	–	–
AP	72.23%	68.13%	57.20%	54.86%	62.18%	53.79%
MaLSTM	70.83%	58.67%	50.56%	55.08%	60.95%	50.29%
UniEmbed(ST)	92.64%	83.34%	70.50%	66.50%	75.15%	68.82%
UniEmbed(JT)	**95.44%**	**87.70%**	**87.69%**	**80.31%**	**78.21%**	**71.85%**

(∼340k) based on the collected data. Note that we delete the generated pairs which are already labelled as similar in original datasets. All the data in each dataset are organized in triple format (e, p, n), where (e, p) is a similar data pair and (e, n) is a dissimilar one. The statistics of all three datasets are shown in Table 2.

Table 2: Statistics of datasets

Dataset	#Instances	#Train	#Valid	#Test
dup-question	535,254	495,254	20,000	20,000
code-anno	693,026	653,026	20,000	20,000
code-clone	33,690	27,690	3,000	3,000

Baseline models

For fair comparison, we re-implement or directly use the existing baseline models as follows:

- AP: (Santos et al. 2016): It is a representation learning model based on two-way attention mechanism and LSTM for pair-wise text classification task.
- MaLSTM (Mueller and Thyagarajan 2016): A siamese adaptation of LSTM relying on Manhattan metric to learn semantic features for text sentences.
- SourcererCC (Sajnani et al. 2016): It is an unsupervised lexical-based clone detection model for code snippets.
- CDLH (Wei and Li 2017): An AST-based LSTM code clone detection model. It efficiently leverages lexical and syntactical information to learn hash codes as representation for code snippet.

Experiment settings

In experiments, we evaluate our UniEmbed model with two different training strategies. UniEmbed (ST) represents training our model on single task using corresponding dataset. For example, we train our model merely on *code-clone* dataset and evaluate on code clone detection task. While UniEmbed (JT) stands for jointly training our model on all three datasets and evaluate on each task. We compare the performance of our model trained under the above two strategies with the aforementioned baseline models on AUC and F-measure metrics.

We use pre-trained 300-dimensional GloVe vectors (Pennington, Socher, and Manning 2014) to encode the text sentence and code snippet in initialization. For the terms not found in the pre-trained vectors, we set their encoding randomly according to standard normal distribution. We apply Adam (Kingma and Ba 2014) to optimize our model with the learning rate set to 0.001. The margin in triplet loss is set as 1.0. We train all the models (including our model and baselines) on 4 Titan X Pascal GPUs for 20 epochs, with batch size set to 64.

Evaluation results

We evaluate our UniEmbed model and baselines on three real-world software mining tasks: duplicated programming question retrieval, code clone detection, code annotation. Since CDLH and SourcererCC are specially designed for processing code snippet, we test them only on code clone detection task. For the other baselines, we extend their original framework to fit all three tasks.

As shown in Table 1, we observe general facts that: (1) our proposed model trained under both strategies clearly outperforms all state-of-the-art baseline models on three chosen tasks, which confirms the capacity of our model in learning representation for both text sentence and code snippet; (2) all jointly trained models achieve better results than singly trained ones, which empirically proves that rich knowledge between tasks is shared during joint training, helping model learn more powerful features than under single training.

Duplicated question retrieval & Code annotation On duplicated programming question retrieval task, our UniEmbed models exceed both AP and MaLSTM by more than 20% on AUC and 15% on F-measure; on code annotation task, our models exceed both AP and MaLSTM by more than 13% on AUC and 15% on F-measure. It's reasonable for UniEmbed to outperform the two strong baseline models because UniEmbed extracts three levels of information (global, local and sequential information) from text sentence and code snippet to construct the final representation, while MaLSTM considers only sequential information and AP considers global and sequential information. We also find that joint training strategy didn't bring much improvement on these two tasks. One possible reason is that *dup-question* and *code-anno* datasets are large enough for UniEmbed to learn high-quality representation, while the knowledge shared between tasks has only small effects compared to the knowledge learned in the enormous datasets.

Code clone detection UniEmbed achieves even more dominant performance in code clone detection task, espe-

Table 3: Ablation studies on the validation set (* means loss does not converge)

	dup-question		*code-clone*		*code-anno*	
Methods	AUC	F-measure	AUC	F-measure	AUC	F-measure
w/o *glb-info*	60.75%	50.05%	50.00%*	33.33%*	50.02%*	33.35%*
w/o *loc-info*	64.81%	51.25%	50.00%*	33.33%*	51.15%	47.54%
w/o *seq-info*	69.74%	63.88%	50.00%*	33.33%*	49.95%	48.89%
w/o attention	82.66%	81.29%	76.48%	74.40%	65.59%	65.55%
full model	**95.44%**	**87.70%**	**87.69%**	**80.31%**	**78.21%**	**71.85%**

cially the one trained under joint training strategy. It exceeds all baseline models by more than 25% on AUC and 20% on F-measure. The results confirm that exploiting certain three levels of information from code snippet do help model learn effective semantic features. Even the state-of-the-art model CDLH couldn't achieve the same performance, because it only leverages lexical and syntactical information. Another possible reason is that CDLH can only process AST-parsable code snippet, while not all data in our dataset satisfy this condition. Thus CDLH's performance is affected, whereas our model doesn't have such restriction. Moreover, note that *code-clone* dataset has a much smaller size than the other two datasets, this may explain why the baseline models and even the singly trained UniEmbed perform poorly on this task. Yet, jointly trained UniEmbed maintains high performance, it indicates that the knowledge shared between related tasks can be helpful in representation learning when training data is insufficient.

Model analysis

What is learned in representation As mentioned in former sections, we attempt to learn uniform features for both text sentence and code snippet and we want the cosine distance between two feature vectors to directly reflect their similarity. Evaluated on test set, we can observe from the results in Table 4 that for all types of data, the gaps between the average cosine distance of similar data pair and of dissimilar data pair are distinguishable, with values 0.4565, 0.4422, 0.2305 corresponding to text pair, code pair, and text-code pair. The results verify the success of our attempt to learn semantic representation, in which cosine distance between two feature vectors reflects their semantic similarity. They also explain why our model achieves high performance on the chosen tasks.

Table 4: Cosine distance of learned representation

Dataset	$D_c(e, p)$	$D_c(e, n)$	Gap
dup-question (text pair)	0.5277	0.9842	0.4565
code-clone (code snippet pair)	0.3366	0.7788	0.4422
code-anno (text-code snippet pair)	0.7773	1.0078	0.2305

Effects of each information We conduct ablation study to investigate the effects of each level of information. We disable one level of information each time by setting the corresponding values in feature vector to all 0s. "w/o *glb-info*", "w/o *loc-info*", "w/o *seq-info*" represent disabling global, local, sequential information respectively. As shown in Table 3, on both AUC and F-measure metrics, full model outperforms all ablated models by more than 23% on *dup-question* dataset, 37% on *code-clone* dataset and 24% on *code-anno* dataset. It indicates that all three levels of information are necessary in constructing the final representation, removing any one component will make the model fail to learn capable semantic features, especially when training data is insufficient (such as *code-clone* dataset). The results can also be explained from another aspect that since we use the simplest linear classifier to measure similarity, the performance will highly depend on the quality of the learned representation. While ablated models extract features of worse quality than full model.

Effects of attention mechanism To inspect whether attention mechanism helps to construct better representation, we conduct ablation study on disabling attention layer in UniEmbed to see how it works. From Table 3, on both AUC and F-measure metrics, we observe that UniEmbed without attention mechanism performs worse than full model by 6% on *dup-question* dataset, 5% on *code-clone* dataset and 6% on *code-anno* dataset. One reasonable explanation is that attention layer learns different importance for each level of information, so that more important information takes heavier proportion in final representation to better reflect semantics.

Conclusion

In this paper, we propose a novel feature learning framework UniEmbed to learn uniform representation for both natural language and programming language. We develop specific structures to extract three levels of semantic information (global, local and sequential) from text sentence and code snippet. When both modalities are involved in the task, we project them into a uniform feature space so that the complementary knowledge in between can be leveraged in their representation. Ablation study proves the effectiveness of each component in our model. Experimental results on three real-world software mining tasks shows that UniEmbed outperforms state-of-the-art methods and reflects the capacity of our model in representation learning.

UniEmbed is a general-purpose representation learning approach. Apart from the three chosen tasks mentioned in this paper, UniEmbed is easily extended to other software mining tasks involving single or both modalities, such as bug localization, code summarization, code search and program translation, etc. Pre-trained UniEmbed can also be used to transform raw textual contents into input semantic features for other models so that improves their performance in certain software mining tasks.

Acknowledgments

This research was supported by National Key Research and Development Program (2017YFB1001903) and NSFC (61751306). The authors would like to thank Zheng Xie, Hui-Hui Wei, Marina Romani for proof-reading this paper, as well as Tencent Corporation for the necessary support.

References

Allamanis, M.; Peng, H.; and Sutton, C. 2016. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33rd International Conference on Machine Learning*, 2091–2100.

Bengio, Y.; Courville, A.; and Vincent, P. 2013. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence* 35(8):1798–1828.

Chung, J.; Gulcehre, C.; Cho, K.; and Bengio, Y. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *Proceedings of the 27th International Conference on Neural Information Processing Systems, Workshop on Deep Learning*.

Gay, G.; Haiduc, S.; Marcus, A.; and Menzies, T. 2009. On the use of relevance feedback in ir-based concept location. In *Proceedings of the 25th International Conference on Software Maintenance*, 351–360. IEEE.

Hoang, T. V.-D.; Oentaryo, R. J.; Le, T.-D. B.; and Lo, D. 2018. Network-clustered multi-modal bug localization. *IEEE Transactions on Software Engineering*.

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 26th International Joint Conference on Artificial Intelligence*, 1909–1915. AAAI Press.

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 25th International Joint Conference on Artificial Intelligence*, 1606–1612.

Johnson, R., and Zhang, T. 2014. Effective use of word order for text categorization with convolutional neural networks. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 103–112. Association for Computational Linguistics.

Joulin, A.; Grave, E.; Bojanowski, P.; and Mikolov, T. 2016. Bag of tricks for efficient text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, 427–431.

Kim, J.; Koh, J.; Kim, Y.; Choi, J.; Hwang, Y.; and Choi, J. W. 2018. Robust deep multi-modal learning based on gated information fusion network. *arXiv preprint arXiv:1807.06233*.

Kingma, D. P., and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Lukins, S. K.; Kraft, N. A.; and Etzkorn, L. H. 2010. Source code retrieval for bug localization using latent dirichlet allocation. *Information and Software Technology* 52(9):972–990.

Mueller, J., and Thyagarajan, A. 2016. Siamese recurrent architectures for learning sentence similarity. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, 2786–2792.

Pennington, J.; Socher, R.; and Manning, C. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, 1532–1543.

Sajnani, H.; Saini, V.; Svajlenko, J.; Roy, C. K.; and Lopes, C. V. 2016. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, 1157–1168. IEEE.

Santos, C. d.; Tan, M.; Xiang, B.; and Zhou, B. 2016. Attentive pooling networks. *arXiv preprint arXiv:1602.03609*.

Schroff, F.; Kalenichenko, D.; and Philbin, J. 2015. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 815–823.

Sridhara, G.; Hill, E.; Muppaneni, D.; Pollock, L.; and Vijay-Shanker, K. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the 25th IEEE/ACM international conference on Automated software engineering*, 43–52. ACM.

Tantithamthavorn, C.; Abebe, S. L.; Hassan, A. E.; Ihara, A.; and Matsumoto, K. 2018. The impact of ir-based classifier configuration on the performance and the effort of method-level bug localization. *Information and Software Technology*.

Wei, 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 26th International Joint Conference on Artificial Intelligence*, 3034–3040.

Wei, H., and Li, M. 2018. Positive and unlabeled learning for detecting software functional clones with adversarial training. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 2840–2846.

White, M.; Tufano, M.; Vendome, C.; and Poshyvanyk, D. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 87–98. ACM.