MulCode: A Multi-task Learning Approach for Source Code Understanding

Deze Wang, Yue Yu*, Shanshan Li*, Wei Dong, and Ji Wang
College of Computer Science
National University of Defense Technology
Changsha, China
{wangdeze14, yuyue, shanshanli, wdong, wj}@nudt.edu.cn

Liao Qing
College of Computer Science and Technology
Harbin Institute of Technology(Shenzhen)
Shenzhen, China
liaoqing@hit.edu.cn

***Abstract*—Recent years have witnessed the significant rise of Deep Learning (DL) techniques applied to source code. Researchers exploit DL for a multitude of tasks and achieve impressive results. However, most tasks are explored separately, resulting in a lack of generalization of the solutions. In this work, we propose MulCode, a multi-task learning approach for source code understanding that learns unified representation space for tasks, with the pre-trained BERT model for the token sequence and the Tree-LSTM model for abstract syntax trees. Furthermore, we integrate two source code views into a hybrid representation via the attention mechanism and set learnable uncertainty parameters to adjust the tasks' relationship.**

We train and evaluate MulCode in three downstream tasks: comment classification, author attribution, and duplicate function detection. In all tasks, MulCode outperforms the state-of-the-art techniques. Moreover, experiments on three unseen tasks demonstrate the generalization ability of MulCode compared with state-of-the-art embedding methods.

***Index Terms*—representation learning, deep learning, multi-task learning, attention mechanism**

I. Introduction

Deep neural networks [1] have achieved impressive performance in various software engineering tasks associated with source code, such as code search [2], [3], code summarization [4], [5], code clone detection [6], [7], and code completion [8], [9]. However, most studies are aimed at one problem and deal with it as an independent task. Those tasks are probably highly interrelated since they all need to understand and model source code with relating artifacts (*e.g.*, code comments). Thus, the rich direct and indirect correlations among those tasks are not fully considered and exploited. Also, the generalization ability of existing single-task models is subject to the specific task-related datasets and learning objectives. For example, Hong *et al.* [10] got a negative result of transferring a state-of-the-art code representation model (*i.e.*, code2vec [11] originally trained for predicting method names) to other tasks including code comment generation, code authorship identification, and code clones detection.

Given the above-mentioned limitations of single-task approaches, many studies tried to solve them by integrating multiple tasks, which have achieved impressive improvements for source code understanding in code summarization, code retrieval, and code change representation tasks [4], [12]–[14]. However, in their frameworks, each task is trained and fitted separately. Wei *et al.* [15] proposed a dual training network to optimize code generation and code summarization simultaneously, which requires expert knowledge to define the relationship between two tasks accurately. Hence, it is not easy to expand to multiple tasks.

In this paper, we present a multi-task learning approach for source code understanding to fill this gap. To improve the generalization ability of the model, we build unified code representation layers for all tasks. All tasks share the parameters of the representation layer; thus, there is no need to define a code representation layer for each task, which greatly reduces the parameters of the model. To build an understandable model, we design task-specific attention layers to focus on and extract important code representation information. To evaluate the performance of our model, we select three downstream tasks for source code, namely Comment Classification (CC) [16], Author Attribution (AA) [17] and Duplicate function Detection (DD) [18]. The reason for choosing these tasks is that their existing expert datasets are too small to design a single neural network for them. The use of multi-task learning can implicitly increase the data for a single task and expand the room for model improvement. We evaluate the performance of our model against models originally designed for the considered tasks.

There are three overarching challenges in building a multi-task model for source code: (a) Source code is rich in information, including sequence information and structure information. In order to understand and model the source code, multiple views of source code should not be ignored. (b) Various task models have different extracted features and convergence speeds. It is not easy to coordinate with them to complement each other during the modeling process. Although predefined parameters can define tasks' relationship, it is time-consuming and challenging to search the parameter space. (c) It is nontrivial to verify the model's generalization ability in various tasks since most models are only verified on the datasets of their own tasks.

To solve the aforementioned challenge (a), we extract information from the tokens sequence and the abstract syntax tree (AST), which contains the hierarchical syntactic structure

*Corresponding author

of the program. Although the parameters of sequence representation and structure representation are shared over tasks, each task pays different attention to sequence and structure. We assign different weights to sequence and structure representations for each task via the attention mechanism. To address the challenge (b), since the loss values for each task have different granularity, we apply the task uncertainty [19] on the calculation of the overall loss function to adjust the relationship of different tasks, and replace the predefined parameters with the learnable parameters, which greatly reduces the cost of parameter tuning. To solve the challenge (c), we design experiments to verify our model's generalization ability in three new tasks, namely library classification, algorithm classification, and bug detection. We fix the parameters of sequence and structure representation layers and generate embeddings for three new tasks. In the verification process, we do 10-fold cross-validation on the standard machine learning framework WEKA [20] to compare the performance with the state-of-the-art embeddings.

On corresponding public datasets of three downstream tasks, we show that our model significantly surpasses the state-of-the-art approaches with a multi-task learning framework, with 36.9%, 10.6%, and 8.1% improvement in author attribution, comment classification, and duplicated function detection, respectively. We also do an ablation study, and it demonstrates that structure representation and setting learnable parameters to balance the relationship of tasks can significantly improve the model's performance. More impressively, without training, the embeddings extracted from our model significantly surpass the state-of-the-art embeddings in tasks of algorithm classification and bug detection and have comparable performance in the library classification task.

The main contributions of this paper are as follows:

- We propose a multi-task learning MulCode model with generalization ability, which can be used to simultaneously solve multiple downstream tasks for source code. To the best of our knowledge, it is the first work in this direction.
- We build a sequence encoder with a pre-trained BERT model and a structure encoder with the Tree-LSTM model. Furthermore, we design the attention layer to merge different representations. Experiments on three unseen tasks demonstrate the generalization ability of our model when compared with state-of-the-art embedding methods.
- We evaluate MulCode in three downstream tasks. Experimental results show that our model achieves the best performance compared with the state-of-the-art models.

The rest of the paper is organized as follows. Section II introduces the background knowledge on multi-task learning and pre-trained models. Section III presents an overview of the MulCode model and describes the details of each component. Section IV presents the experimental settings and the evaluation results. Section V describes the strengths, threats to validity, and limitations of our model, followed by Section VI that presents the related work. Finally, Section VII concludes the paper.

II. Background

In this section, we will introduce the background knowledge on multi-task learning and pre-trained models.

A. Multi-task Learning

Multi-Task Learning (MTL) is a machine learning method that is opposite to single-task learning. It puts multiple related tasks together and learns multiple tasks simultaneously. MTL provides an effective method that can utilize the supervised data in related tasks and reduce the dependence on the task-specific labeled data. The use of MTL can reduce over-fitting to specific tasks and play a role similar to regularization. Leveraging MTL to improve the performance of tasks has been explored in many scenarios.

Hard parameter sharing and soft parameter sharing are two commonly used multi-task learning methods based on deep neural networks. The hard parameter sharing mechanism is the most common method in multi-task learning. [21] It can be applied to hidden layers of all tasks while retaining the output layer related to the task. In this way, the hard parameter sharing mechanism reduces the risk of over-fitting. In the soft parameter sharing mechanism, each task has its own model and parameters. The similarity of the parameters is guaranteed by regularizing the distance of the model parameters.

In this paper, we employ a hard parameter sharing mechanism to build universal representation layers for source code, which is an intuitive idea that can easily be extended to multiple tasks.

B. Pre-trained Models

When encountering a new task, it is costly to train a model from scratch. An easy way is to employ transfer learning [22] to get a solution to a similar task. The pre-trained model is a saved network that was previously trained in similar tasks. A successful pre-trained model is ELMO [23]. ELMO applies the language model for pre-training and then extracts each layer's word embeddings as new features to improve performance in downstream tasks. GPT [24] employs a unidirectional transformer [25] model for pre-training, transforms downstream tasks for fine-tuning, and achieves excellent results in nine natural language tasks. BERT [26] designs the masked language model, and the next sentence prediction task for pre-training and obtains the state-of-the-art results in eleven natural language tasks.

The advantage of the pre-trained model is that it can be quickly transferred to new tasks with a small training cost. In this paper, we use the pre-trained BERT model to extract the sequence representation of source code and fine-tune it during the training process. Simultaneously, the transformer architecture used by the sequence encoder has a powerful capability of feature extraction and can model the long-term dependency in code.

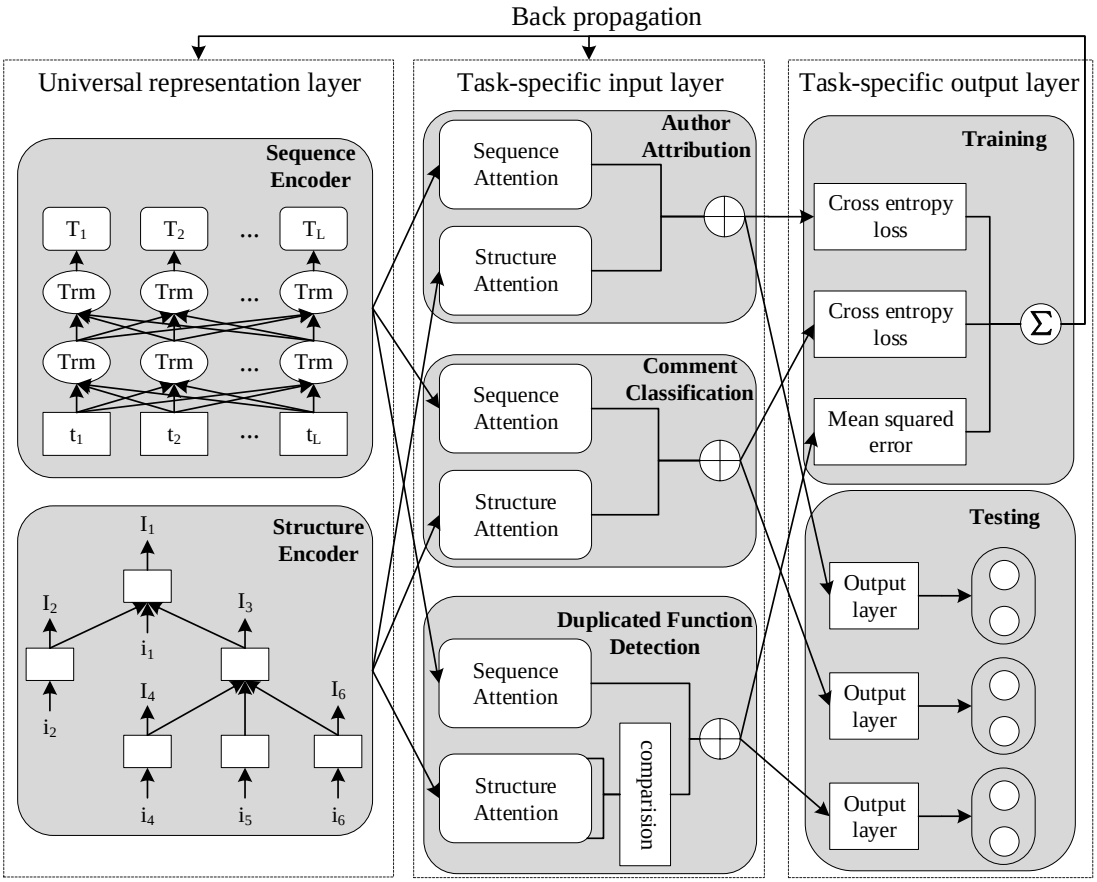


Fig. 1. The architecture of our proposed MulCode model.

III. Proposed Model

In this section, we first present an overview of our MulCode model. We then describe the details of each component of MulCode. Finally, we show how to learn multiple tasks jointly.

A. Overall Architecture

Figure 1 shows the architecture of our proposed MulCode model. We take three downstream tasks as examples, although we can incorporate various code tasks. The MulCode model consists of three submodules as follows:

- Universal representation layer. This submodule receives input from all tasks and uses a universal sequence encoder and structure encoder to generate representations for tasks.
- Task-specific input layer. Various tasks pay different attention to the sequence representation and structure representation; therefore, this submodule will extract and merge the above representations by attention mechanism.
- Task-specific output layer. This submodule is to calculate task-specific output based on the representation vectors and specific task objectives.

B. Universal Representation Layer

In this section, we extract features from multiple views of source code for different tasks. We design two universal encoders to represent these features, including a sequence encoder with a pre-trained BERT model and a structure encoder with the Tree-LSTM model [27].

a) Sequence encoder: The sequence encoder receives source code and other task input and returns sequence representation vectors for them. It processes the input from different tasks, including ⟨comment, code⟩ pairs for the comment classification task, ⟨code, author⟩ pairs for author attribution task, and ⟨func A, func B⟩ pairs for duplicated function detection task respectively. Each pair is input as a whole, and the two parts of the pair are separated by a special token "[SEP]".

To capture the long-range dependency in the input sequence, we use a bidirectional transformer as the sequence encoder to map the above input into a representation vector $C^{seq} \in R^{L\times d}$, where L is the sequence length and d is the embedding dimension. The sequence representations of three tasks are calculated as follows:

$$C^{seq}_{AA}, C^{seq}_{CC}, C^{seq}_{DD} = Transformer(I^{seq}_{AA}, I^{seq}_{CC}, I^{seq}_{DD}) \quad (1)$$

where $I^{seq}_{task} = \{t_1, ..., t_L\}$ is the token sequence of length L and represents the sequence input of the specific task.

It is worth noting that the bidirectional transformer's weights are pre-initialized using the pre-trained model of BERT and will be updated with a lower learning rate during the training process. In this way, there is enough model space to represent all tasks while not converging too slowly.

b) Structure encoder: In this subsection, we will extract the structure representation for the source code input. We first extract the AST for the source code using Eclipse's JDT compiler[1] and calculate the embeddings for each node of the AST before building the structure representation. To avoid introducing new embedding parameters and ensure that the same embedding parameters are used as the sequence encoder, we employ the pre-trained BERT model for embedding. In this way, the initial embeddings of sequence and structure encoders are in the same model space, making it easier for the model to establish the relationship between the two representations. Then we apply Tree-LSTM to represent the AST information. For standard LSTM, only the previous step's hidden output will be used when calculating the output of the current step. The Tree-LSTM will calculate the sum of the hidden outputs of the child nodes. Simultaneously, to filter important information among child nodes, it will construct a forget gate for each child node. The structure representations are calculated as follows:

$$\begin{aligned} \widetilde{h}_j &= \sum_{k \in Child(j)} h_k \\ h_j &= TreeLSTM(\widetilde{h}_j, I^{str}) \end{aligned} \tag{2}$$

where k is one of the child nodes of j, and the hidden state $\widetilde{h}_j$ is the sum of the child nodes' hidden outputs. I^{str}, which denotes the structure input of tasks, contains a list of node tokens and an adjacency list representing the relationship of nodes. h_j is the structural representation of the node j.

In this paper, in order to pay attention to the global information of all nodes, we do not take the hidden state of root node as the structure representation, but keep the hidden vectors of all nodes. Therefore, the structure representations of different tasks are represented as follows:

$$\begin{aligned} C^{str}_{AA} &= TreeLSTM(I^{str}_{AA}) \\ C^{str}_{CC} &= TreeLSTM(I^{str}_{CC}) \\ C^{str}_{DD}(Func\ A) &= TreeLSTM(I^{str}_{DD}(Func\ A)) \\ C^{str}_{DD}(Func\ B) &= TreeLSTM(I^{str}_{DD}(Func\ B)) \end{aligned} \tag{3}$$

where $C^{str} \in R^{N \times d}$ is the structure representation vectors and N is the number of nodes. Since the DD task has the input of two code segments, Func A and Func B, there are two structure representation vectors in the result.

C. Task-specific Input Layer

Since various tasks pay different attention to sequence representation and structure representation, this section introduces the attention layer to calculate the task-specific input for each task.

[1]http://www.eclipse.org/jdt

a) Sequence Attention: Different tokens contribute differently to the overall representation due to their locations, tasks and information contained. Therefore, we design the sequence attention layer for each task. The layer is represented as follows:

$$\begin{aligned} \alpha^{seq}_j &= \frac{exp(w^{seq} C^{seq}_j)}{\sum_i exp(w^{seq} C^{seq}_i)} \\ C^{seq} &= \sum \alpha^{seq}_j C^{seq}_j \end{aligned} \tag{4}$$

where α^{seq}_j is the attention score of j-th token, C^{seq}_j is the sequence representation of j-th token and w^{seq} is the task-specific parameter matrix. Finally, each task has a C^{seq}_{task} vector as its sequence representation.

b) Structure Attention: In the previous section, we calculate the representations of all nodes, but not all nodes have the same contribution to the final representation. Therefore, we design the attention layer to extract and merge the structure representations. The structure attention layer is represented as follows:

$$\begin{aligned} \alpha^{str}_j &= \frac{exp(w^{str} C^{str}_j)}{\sum_i exp(w^{str} C^{str}_i)} \\ C^{str} &= \sum \alpha^{str}_j C^{str}_j \end{aligned} \tag{5}$$

where α^{str}_j is the attention score for j-th node. C^{str}_j is the structure representation of j-th node. w^{str} is the task-specific parameter matrix and is randomly initialized and learned during the training process. Finally, each task has a C^{str}_{task} vector as its structure representation.

Different from the author attribution and comment classification task, the duplicated function detection task will get two structure representation vectors due to the input. We design a comparison layer to capture their differences after the attention layer.

$$C^{str}_{DD} = ReLU(W[C^{str}_{DD}(Func\ A); C^{str}_{DD}(Func\ B)] + b) \tag{6}$$

where $W \in R^{2d \times d}$ and $b \in R^d$ are trainable parameters. ";" denotes the concatenation operator.

c) Concatenation: We merge the above representations into a representation vector. The output is represented as follows:

$$x_{task} = [C^{seq}_{task}; C^{str}_{task}] \tag{7}$$

where x_{task} is the final representation vector of the task.

D. Task-specific Output Layer

In this section, we describe the output layers of the three downstream tasks. Although our MulCode model can be applied to various code tasks, we choose three tasks with small public datasets. Therefore, we can utilize the characteristics of multi-task learning to increase data for each task implicitly and to explore better performance in these tasks.

a) Comment Classification Task: Given the input comment X_1 and the code snippet X_2, the final representation is represented as x_{CC}. We design a layer of logistic regression network to convert it into a vector with a dimension of the category c and apply softmax to normalize the vector. The equation is represented as follows:

$$P(c \mid X_1, X_2) = softmax(W_{CC}^T \cdot x_{CC}) \tag{8}$$

where $c \in \{0, 1\}$, the value of 0 means that the comment is inconsistent with the code, and the value of 1 means that the comment is consistent with the code. W_{CC}^T is the parameter matrix of the logistic regression network.

b) Author Attribution Task: Given the code snippet X, the final representation is represented as x_{AA}. The task is to identify the author of the code snippet from the candidate set. Since this task is a multi-classification task, we built a multi-layer perceptron (MLP) to calculate the probability that X is labeled with class c. The formula is represented as follows:

$$P(c \mid X) = softmax(MLP(x_{AA})) \tag{9}$$

where c represents the category of authors, and there are 13 categories in total.

c) Duplicate Function Detection Task: Given a pair of code snippets $\langle X_1, X_2 \rangle$, the final representation is represented as x_{DD}. The task is to judge whether the two code snippets implement the duplicated function. We use the similarity to measure the relationship between two input code snippets:

$$Sim(X_1, X_2) = W_{DD}^T \cdot x_{DD} \tag{10}$$

where W_{DD}^T is the task-specific parameter matrix. Assuming that c is the label of the input, we use the mean squared error $(c - Sim(X_1, X_2))^2$ as the optimization objective.

E. Model Training

In order to learn multiple tasks jointly, it is necessary to balance each task, ensuring that the loss of each task is not too large or small. In this way, the task-specific parameters can be updated normally. Inspired by the research of Kendall *et al.* [19], we set weight parameters for different tasks according to the uncertainty of the task. The overall loss function of the model is represented as:

$$L(W, \sigma) = \sum_{i=1}^{N} (\frac{1}{2\sigma_i^2} * L_i(W_i) + log\sigma_i) \tag{11}$$

where σ_i is the uncertainty scalar of each subtask and is the trainable parameter. N is the number of tasks. W_i is the task-specific trainable parameters. When σ_i increases, the contribution of the corresponding task loss $L_i(W_i)$ will be reduced, and vice versa. At the same time, σ_i will be restricted by the last term to avoid being too large.

During training, we calculate the loss for a specific task in a batch. In an epoch, we traverse all training data of tasks so that the model will be updated according to the overall objective of all tasks.

IV. EXPERIMENTS AND ANALYSIS

In this paper, we consider three downstream tasks for source code: comment classification, author attribution, and duplicated function detection. We will introduce the objectives of each task, the datasets we use, the corresponding baseline methods, and the experimental settings. Finally, we propose four research questions and design experiments to answer them.

A. Datasets and Baselines

This section briefly describes the objectives of each task and the datasets we use, and the corresponding baseline methods. We first introduce three downstream tasks used in the experiments.

a) Comment Classification Task: This task is to decide if the quality of a code comment is reliable. It is not only the comment itself but also the relationship between the comment and the corresponding source code that determines the quality of the comment. This task has many implications for code understanding and software maintenance. In order to complete the comment classification task, it is required to not only model the source code but also understand comments in the natural language form and the relationship between comments and code.

We compare with the state-of-the-art method [28] in comment classification and use the dataset they released. They adopt the Support Vector Classification (SVM) technique based on Radial Basis Function (RBF) kernel to classify the comments. Their dataset contains a manual assessment of the comments and the implementation of 3636 methods in three open-source software applications.

b) Author Attribution Task: This task is to identify the author of the code snippet. Authorship Attribution has been developed substantially in natural language processing [29], such as applications on e-mail messages, online forum messages, and blogs. However, in the field of source code research, there are fewer studies focusing on this task. Identifying the author of a program is a challenging but essential task in computer security. The source code usually contains features that range from simple artifacts in comments, code layout to habits in the use of syntax and control flow. [30] These characteristics reflect various coding styles and can be used to identify programmers.

In order to compare existing methods, we choose the latest method [31] and use the dataset they built. They design approaches for reducing reliance on variable names by training a code2vec model on the obfuscated data, which forces the model to look at the structure information in the code. The dataset is built based on Leetcode[2], which is a website designed for people to improve their coding skills by solving questions posted on the site. The dataset contains 13 categories and 1062 instances.

[2]https://leetcode.com/

c) Duplicated Function Detection Task: The last task is to determine if a pair of code fragments implements the duplicated function. Duplicated functions will increase the overhead of software maintenance and increase program complexity. Therefore, this task has many implications for software development and maintenance. This task is to determine the relationship between a pair of code segments. In this paper, we consider the sequence and structure relationship of a pair of code segments together.

In order to balance the size of datasets used by these tasks, we use a hand labelled dataset [32] on GitHub shared by software company Source{d} and compare our model with the corresponding method. The dataset contains 1277 instances.

We also choose three downstream tasks for the validation of the generalization ability of our model. These tasks have datasets of different scales and are not suitable for joint learning with the above three tasks of a similar scale. Moreover, these tasks are different in purpose and difficulty, which can thoroughly test the performance of our model in the unseen data.

d) Library Classification Task: This task is to distinguish two kinds of code snippets that use different libraries. One uses OpenCV, an open-source image processing library, and the other uses Spring, a popular Java web application framework. This task is quite simple and is a direct way to check whether the embedding method is effective. This dataset was created manually using files from GitHub and other website tutorials by Compton *et al.* [31]. The dataset contains two categories and 305 instances.

e) Algorithm Classification Task: This task is to identify the algorithm implemented in the code snippet. Princeton University has published many implementations of algorithms on its Java Algorithms and Clients page[3], which are divided into their respective categories (sorting, searching, graphs *etc.*). The dataset contains seven categories and 182 instances.

f) Bug Detection Task: This task is to predict whether a given code snippet has a bug. The Public Unified Bug Dataset for Java [33] is collected from five public datasets and hand-labelled with the number of bugs. The dataset is converted into two categories for evaluation, one with bugs and the other without bugs. The dataset contains 31135 instances.

We compare our model with state-of-the-art methods code2vec and the obfuscated code2vec [31] in the above three tasks. The code2vec model is trained on the java-large dataset, a larger dataset with 15.3m examples, sourced from the code2seq GitHub repository[4]. In order to train the obfuscated code2vec model, the variable names in the dataset are obfuscated randomly. Reducing reliance on variable names may gain an improvement of performance in some tasks.

We employ accuracy and F1-score to compare the performance of the approaches. Accuracy is the ratio of the number of correct predictions to that of the total predictions. F1 is the harmonic mean between precision and recall. In the experiment to verify the generalization ability, we also use the metric kappa consistent with the original paper of the obfuscated code2vec. Kappa coefficient is used to measure classification accuracy based on the calculation of the confusion matrix. When dealing with unbalanced datasets, kappa can reflect the experimental results more truly than accuracy. [34]

B. Experimental Setup

The implementation of our model is based on the PyTorch implementation of BERT. The word embedding size and the embedding size of the tree node are set to 768. We set the batch size to 8 and use Adamax [35] as the optimizer with a learning rate of 5e-5. To avoid overfitting, we adopt dropout [36] with a drop probability of 0.1. We randomly select 80% samples for each dataset as the training set, 10% samples as the test set, and 10% for validation. In the evaluation of the bug detection task, we apply downsampling to keep the balance of the two categories. To train the model, we employ the cross-entropy loss for comment classification and author attribution tasks and employ the mean squared error for the duplicated function detection task. We run our experiments on a Linux server with the NVIDIA v100 GPU and 92 GB memory.

For baseline methods, code2vec and the obfuscated code2vec are trained with default hyperparameters using the training script supplied in the code2vec GitHub repository[5]. In the experiment to verify the model's generalization ability, we use the standard machine learning framework WEKA and its implementation of sequential minimal optimization (SMO) [37] to classify all embeddings from our model and baseline methods. The SMO classifier is set in the default mode with a poly kernel. The tolerance parameter is 0.001, and the batch size is set to 100.

C. Research Questions and Results

To evaluate our proposed MulCode model, in this section, we conduct experiments to answer the following research questions:

RQ1: How does MulCode perform in different tasks when compared with state-of-the-art models?

To answer this research question, we compare MulCode with state-of-the-art models of different downstream tasks, which are set as the baseline models in the experiment. To demonstrate the effectiveness of learning multiple tasks together, we also compare MulCode with the single-task models. The single-task model adopts the same architecture as the multi-task model, including universal representation layers, a task-specific attention layer, and an output layer. The single-task model is trained on the dataset of a single task. Compared with the single-task model, our MulCode model only uses data from other tasks without increasing the parameters for each task. The results are shown in Table I.

As can be seen from Table I, our MulCode model outperforms all the baselines in three downstream tasks, especially in the author attribution task. The author attribution task is a multi-class classification problem with 13 categories and a

[3]https://algs4.cs.princeton.edu/code/

[4]https://github.com/tech-srl/code2seq

[5]https://github.com/tech-srl/code2vec

TABLE I
COMPARISON OF THE OVERALL PERFORMANCE BETWEEN MULCODE, BASELINES AND THE SINGLE-TASK MODEL.

Model	Author Attribution		Comment Classification		Duplicated Function Detection	
	ACC	F1	ACC	F1	ACC	F1
Baseline	32.1%	35.0%	81.6%	70.3%	85.0%	84.0%
Single	70.7%	68.1%	84.9%	79.6%	87.7%	91.1%
MulCode	**73.0%**	**71.9%**	**85.7%**	**80.9%**	**89.1%**	**92.1%**

TABLE II
COMPARISON OF THE OVERALL PERFORMANCE BETWEEN MULCODE, CODE2VEC AND THE OBFUSCATED CODE2VEC.

Model	Algorithm Classification			Library Classification			Bug Detection		
	ACC	F1	Kappa	ACC	F1	Kappa	ACC	F1	Kappa
code2vec	74.6%	74.2%	0.695	98.0%	97.4%	0.948	66.1%	65.7%	0.309
obfuscated code2vec	83.4%	81.8%	0.779	**99.7%**	**99.7%**	**0.993**	65.0%	64.7%	0.288
MulCode	**87.1%**	**86.5%**	**0.836**	98.4%	98.4%	0.967	**72.8%**	**72.6%**	**0.453**

small dataset. Although it is a difficult task, MulCode achieves the F1-score of 71.9%, which improves the baseline model by 36.9%. For the comment classification task, the improvement to the baseline model is 10.6%. For the duplicated function detection task, MulCode has an improvement of 8.1% over the baseline. The last two tasks are binary classification tasks. The tasks are simple compared to the first task, but it is not easy to improve over the baselines.

Compared with the single-task models, MulCode has a small improvement over them. Our model does not expand the model capacity and introduce new parameters for each task compared to the single-task models. Just by adding the training data from other tasks, MulCode shows improvements over the single-task models. We can observe that learning multiple tasks together indeed has a positive effect. The reason may be that the representation layer constructed from data of multiple tasks can balance the noise in single task data and avoid falling into the local optimal solution on the training set of a single task.

RQ2: How about the generalization ability of MulCode model in other tasks?

To answer this question, we conduct experiments on three new downstream tasks: library classification, algorithm classification, and bug detection. We train code2vec and the obfuscated code2vec to generate embeddings for the above three tasks. To generate embeddings for three tasks using our model, we fix the trained MulCode model's parameters. We use the pre-trained sequence encoder and structure encoder to generate two representation vectors for each task and then concatenate them as their embeddings.

To further measure the quality of these embeddings, we convert these embeddings and their corresponding labels into a data format that can be evaluated by the framework WEKA. Using the existing standard framework can eliminate manual intervention and ensure the fairness and reproducibility of comparative experiments. In the experiment, we choose the SMO classifier implemented by WEKA, which has the best classification performance on these embeddings. The SMO algorithm is an improved algorithm of the original support vector machine, and its parameter settings are shown in IV-B. Finally, the metrics calculated by the SMO classifier reflect the quality of embeddings. The experimental results are shown in Table II.

As can be seen from Table II, our reproduced results of code2vec and the obfuscated code2vec in three tasks are almost the same as that of the original paper by Compton *et al.* [31]. Our model greatly surpasses code2vec and the obfuscated code2vec in algorithm classification and bug detection tasks and has comparable performance to baseline models in the library classification task. Although we have not trained in these tasks, MulCode achieves better performance than other models, indicating the generalization ability of MulCode. Compared to other models, MulCode requires less data to train. Furthermore, in the experiment, code2vec and the obfuscated code2vec use 2304-dimensional vectors for representation, while MulCode uses 1536-dimensional vectors. Obviously, our representation vectors are denser and more informative. As for the slightly lower performance than the obfuscated code2vec in the library classification task, the reason may be that our model is misled by the typos in the variable names, which have been processed by the obfuscated code2vec.

RQ3: What is the effectiveness of each component for our MulCode model?

In this section, we design an experiment to test the effects of our model's two components: the structure encoder and the balance mechanism that balances the relationship between different tasks. The experimental results are shown in Table III.

We conduct experiments with and without considering the structure of our model and single-task models. We also conduct an experiment by removing the balance mechanism from the full model. The first two rows show the single-task model results without (w/o) and with (w/) the structure encoder. The results of MulCode without and with the structure encoder

TABLE III
EFFECTIVENESS OF EACH COMPONENT IN MULCODE MODEL.

Model	Author Attribution		Comment Classification		Duplicated Function Detection	
	ACC	F1	ACC	F1	ACC	F1
Single w/o struct.	66.0%	62.8%	83.3%	77.0%	85.9%	90.0%
Single w/ struct.	70.7%	68.1%	84.9%	79.6%	87.7%	91.1%
MulCode w/o struct.	70.7%	66.8%	84.7%	80.1%	87.0%	90.6%
MulCode w/ struct.	**73.0%**	**71.9%**	**85.7%**	**80.9%**	**89.1%**	**92.1%**
MulCode w/o balance	70.7%	70.8%	84.9%	80.0%	82.8%	86.8%

are shown in the next two rows. The results of removing the balance mechanism from the full model are shown in the last row. It can be seen from the results that removing the structure encoder from the single-task model and MulCode leads to a drop in the accuracy and F1-score, which demonstrates that in the field of source code research, structure information is necessary for feature extraction.

When removing the balance mechanism from the entire model, we notice a significant drop in accuracy and F1-score. Especially in the DD task, the performance of MulCode without the balance mechanism is even worse than that of the single-task model. One possible reason is that the DD task employs the mean squared error loss function, which is different from the other two tasks that employ the cross-entropy loss function. The granularity of the two functions' loss values is different, and the loss value of the DD task may be too small compared to the other two tasks. As a result, the task-specific parameters of the DD task cannot be updated normally in the training process of MulCode without the balance mechanism. The above results show that blindly adding multiple tasks to a single-task model cannot effectively improve performance in specific tasks. It is of great importance to design an understandable balance mechanism to balance the relationship between multiple tasks. We balance the granularity of the loss values by setting a learnable parameter for each task and design a penalty term to prevent our model from falling into the overfitting of single task data.

RQ4: What is the difference of internal behaviour between the MulCode model and the single-task models in the learning process?

In order to find the difference of internal behaviour between MulCode and the single-task models, we conduct experiments to compare their performance changes during the training process. Figure 2 shows the changes of F1-score after each epoch during the training process for two types of models in three downstream tasks. In order to highlight the trend of change, we save the trained results until the epoch where the best performance is obtained on the test set.

As seen from the three sub-graphs, the F1-score of single-task models is higher than that of MulCode from the beginning to the 5th epoch. At this time, single-task models receive single task data and optimize the single-task objectives that are easier to converge to better performance. In contrast, MulCode receives mixed data from multiple tasks and optimizes towards the joint direction at a slower speed. Although the multi-task model converges slightly slower than the single-task models, it achieves higher performance than single-task models in the next epochs. The difference in the three sub-graphs is that MulCode clearly outperforms the single-task models in the author attribution task (Figure 2(a)) and duplicated function detection task (Figure 2(c)). At the same time, it has no obvious advantage over the single-task model in the comment classification task (Figure 2(b)). The possible reason is that the model of comment classification task needs to learn the relationship between source code and natural language, and there is less benefit gained from other source code learning tasks.

V. DISCUSSION

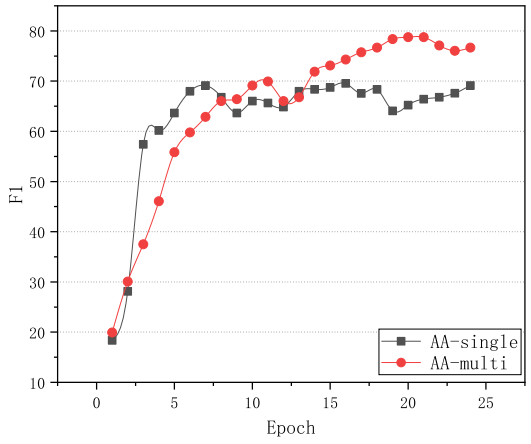
A. Strength

Our proposed model has three apparent advantages in the learning of source code task:

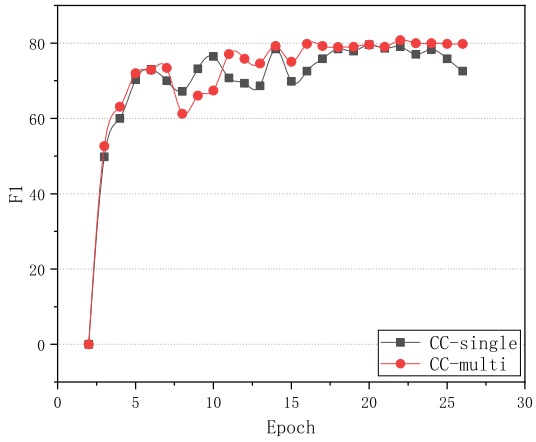
(1) A general language learning way for source code. The universal sequence encoder and structure encoder learn a general vector space from the token sequence and AST of source code. All tasks share the knowledge embedded in the vector space, helping reduce the overfitting of a single task data during the learning process. This is also the main reason why our model can improve performance over single-task models only by adding data from other tasks without increasing the model capacity for each task.

(2) An understandable neural network for multiple tasks. Sequence representation and structure representation contribute differently to the final representation for various tasks. Therefore our model introduces the task-specific attention layers to assign weights to different representations. Since the choice of loss function may lead to a different granularity of the loss value, some task-specific parameters cannot be updated normally. We design learnable weight parameters to adjust the relationship between different tasks.

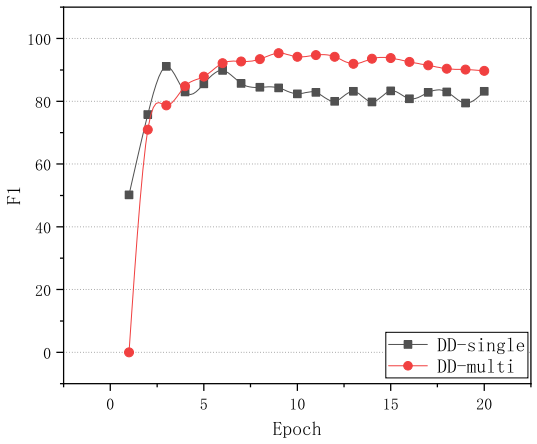
(3) It is easy to expand to multiple tasks. Our model is built with the token sequence and AST, which exist in the most programming language. When a new task arrives, the model only needs to add a task-specific output layer without adding a specific embedding layer. Since the representation layer has embedded general knowledge learned from other tasks, the new task requires little training data to learn a usable representation. More impressively, we extract embeddings from the fixed sequence encoder and structure encoder and



(a) Author Attribution



(b) Comment Classification



(c) Duplicated Function Detection

Fig. 2. The F1-score on the training set for the single-task models and the MulCode model.

compare with the state-of-the-art approaches, and the results show the generalization ability of our model.

B. Threats to Validity and Limitations

One threat to validity is the quality of the datasets we use. All datasets are obtained from previous researches. We implement MulCode based on the existing datasets and reproduce the state-of-the-art approaches. However, the Java datasets currently used are of small scale, and we need to do further research on more datasets of various tasks and languages. In the process of reproducing the obfuscated code2vec, we contact the authors and confirm that their scripts could not compile a small number of constructor functions. Therefore, our model is slightly different from their model in the number of data instances. Another limitation is time overhead. Our proposed model takes about 24.5 hours for three tasks. If it is extended to tasks with larger datasets, reliable computing power is required.

VI. Related Work

A. Representation Learning for Source Code

In the process of multi-task learning, it is necessary to learn the code representation for each task. There are many studies on source code representation. Allamanis *et al.* [38] did an understandable survey of source code representation. DeFreez *et al.* [39] learned program embedding through the control flow graphs of the source code. Gu *et al.* [2] combined the embedding of plain texts and source code to improve code search. Iyer *et al.* [4] combined the attention mechanism with an LSTM network model for code summarization, and Yin *et al.* [40] trained their models on natural language texts and corresponding code from Stack Overflow. Hu *et al.* [41] proposed a method of generating comments for method-level code using a sequence-to-sequence translation model. They also added the AST structure information of the code as the input to the model. Alon *et al.* [11] used paths in the AST to learn the representation of code. They verified their model in the task of predicting the method name of the code fragment. They also proposed code2seq [5]. The difference from previous work is that this model generates a series of word sequences instead of individual words. Wan *et al.* [3] introduced the multi-modal representation of source code in their research, that is, using token sequence, abstract syntax tree, and control flow graph to construct code representations. They achieved excellent results in the code search task. These researches have gradually made full use of multiple views of programming languages. However, LSTM is always used when extracting sequence features. It cannot capture long-term dependency and is time-consuming.

In recent years, large-scale pre-trained models have made significant progress. The most representative neural network is Transformer [25], which uses a multi-head attention mechanism to perform the end-to-end learning under the encoder-decoder framework. The GPT [24] model employs language modeling to predict the next word according to a given context and to learn a more general context representation. BERT [26]

uses the mask mechanism for language modeling, predicts randomly masked words in a given context, and achieves the state-of-the-art results in 11 different natural language tasks. Although the pre-trained model has a large enough model capacity, it only pays attention to the input sequence information. In this paper, we explore the fusion of a pre-trained model for sequence and source code model that focuses on structure.

B. Code Modeling for Multiple Tasks

In the field of source code research, many studies only focus on a single task. However, there are also some studies on learning of multiple tasks through reinforcement learning, joint learning, dual learning, and adversarial learning. He *et al.* [42] proposed a dual learning framework to learn machine translation from English to French and from French to English simultaneously. Chen *et al.* [12] and Iyer *et al.* [4] proposed that their model can be directly applied to code retrieval and code annotation tasks with slight modifications. However, their training goals only consider one of the two tasks. Yao *et al.* [13] employed reinforcement learning to combine tasks of code comment generation and code retrieval and designed rewards based on the above tasks to update the entire network. Wang *et al.* [43] also combined the two tasks of code retrieval and code comment generation, and the difference is that they established a framework based on the transformer. The goal of Hoang *et al.* [14] is to learn the distributed representation of code changes and to optimize the three tasks of log information generation, patch identification, and defect detection. Although this work can achieve good results in multiple tasks, each task is trained and fitted separately, which is different from the joint learning of multiple tasks. Feng *et al.* [44] built a pre-trained model based on the transformer using source code and natural language as the corpus, and their model surpassed the original natural language pre-trained model in two tasks. Wei *et al.* [15] proposed a dual training network to optimize the two tasks of code generation and code summarization simultaneously. They designed regular term constraints based on the relationship between the two tasks. This work requires expert knowledge to define the relationship between the two tasks accurately, and it is not easy to expand to multiple tasks. Liu *et al.* [9] successfully solved the code completion task with the help of the multi-task learning method for the first time. This research focused on selecting auxiliary tasks to help the main task achieve good results. The auxiliary task predicts the node type and value of the AST and will help understand the hierarchical structural information of the AST.

C. Multi-task Learning

Multi-task learning has been applied to various domains, such as language [45]–[47], robotics [48], [49], video [50], [51], and speech recognition [52]. Luong *et al.* [53] used syntactic parsing and image caption generation as auxiliary tasks to improve translation quality. Liu *et al.* [54] combined pre-trained language models and multi-task learning and have achieved state-of-the-art results in ten natural language processing tasks. Sun *et al.* [55] proposed the ERNIE 2.0 framework, which is built with multiple pre-trained tasks incrementally, and then trained the model through continuous multi-task learning. Experimental results show that the ERNIE 2.0 model outperforms existing researches in multiple English and Chinese tasks.

In this paper, we employ hard parameter sharing to build an end-to-end neural network model for multiple tasks. Compared with previous studies in natural language and other fields, we extract code features from its multiple views, which adapts the representation to the structural characteristics of the programming language. Furthermore, we integrate the attention mechanism and multi-task learning network to build a more understandable model.

VII. Conclusion and future work

Good code models should not only be competent for a single task but also be applied to multiple tasks universally, which requires that the model has sufficient capacity and scalability to be easily extended to new tasks. To build a code model with generalization ability and interpretability, we propose the MulCode model based on the attentional multi-task neural network. We employ the pre-trained BERT model and tree-LSTM model to capture the sequence and structure information of source code to ensure that MulCode has sufficient capacity. We design the attention layer and balance mechanism that adjust the relationship between different tasks to obtain an understandable model and improve the overall performance. Our results have shown that MulCode is effective and outperforms all state-of-the-art approaches for multiple downstream tasks. Furthermore, our experimental results show that the MulCode model can be easily extended to multiple tasks, which will help solve many tasks in the field of source code research.

In the future, we plan to expand our model to more multi-domain tasks and multi-language tasks and solve unlabeled tasks with expensive labeling costs. We believe that it is promising to build the source code model with generalization ability in multiple tasks.

We have released the data and code used in this study on https://github.com/wangdeze18/Mtl

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments. This work was substantially supported by National Natural Science Foundation of China (No. 62032019, 61872373, 61702534, and 61702134); National Key Research and Development Program of China (No. 2018YFB0204301) and National Grand RD Plan (No. 2020AAA0103504).

References

[1] P. Devanbu, M. Dwyer, S. Elbaum, M. Lowry, K. Moran, D. Poshyvanyk, B. Ray, R. Singh, and X. Zhang, "Deep learning & software engineering: State of research and future directions," *arXiv preprint arXiv:2009.08525*, 2020.

[2] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.

[3] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. Yu, "Multi-modal attention network learning for semantic source code retrieval," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 13–25.

[4] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 2073–2083.

[5] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[6] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018, pp. 542–553.

[7] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 141–151.

[8] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, 2018, pp. 4159–25.

[9] F. Liu, G. Li, B. Wei, X. Xia, Z. Fu, and Z. Jin, "A self-attentional neural architecture for code completion with multi-task learning," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 37–47.

[10] H. J. Kang, T. F. Bissyandé, and D. Lo, "Assessing the generalizability of code2vec token embeddings," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1–12.

[11] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[12] Q. Chen and M. Zhou, "A neural framework for retrieval and summarization of source code," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 826–831.

[13] Z. Yao, J. R. Peddamail, and H. Sun, "Coacor: code annotation for code retrieval with reinforcement learning," in *The World Wide Web Conference*, 2019, pp. 2203–2214.

[14] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: distributed representations of code changes," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 518–529.

[15] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," in *Advances in Neural Information Processing Systems*, 2019, pp. 6563–6573.

[16] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *2013 21st international conference on program comprehension (icpc)*. Ieee, 2013, pp. 83–92.

[17] I. Krsul and E. Spafford, "Authorship analysis: identifying the author of a program," *Comput. Secur.*, vol. 16, pp. 233–257, 1997.

[18] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 1998, pp. 368–377.

[19] A. Kendall, Y. Gal, and R. Cipolla, "Multi-task learning using uncertainty to weigh losses for scene geometry and semantics," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7482–7491.

[20] G. Holmes, A. Donkin, and I. H. Witten, "Weka: A machine learning workbench," in *Proceedings of ANZIIS'94-Australian New Zealnd Intelligent Information Systems Conference*. IEEE, 1994, pp. 357–361.

[21] M. Crawshaw, "Multi-task learning with deep neural networks: A survey," *arXiv preprint arXiv:2009.09796*, 2020.

[22] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2009.

[23] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *Proceedings of NAACL-HLT*, 2018, pp. 2227–2237.

[24] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018.

[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[26] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[27] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *arXiv preprint arXiv:1503.00075*, 2015.

[28] A. Corazza, V. Maggio, and G. Scanniello, "Coherence of comments and method implementations: a dataset and an empirical investigation," *Software Quality Journal*, vol. 26, no. 2, pp. 751–777, 2018.

[29] E. Stamatatos, "A survey of modern authorship attribution methods," *Journal of the American Society for information Science and Technology*, vol. 60, no. 3, pp. 538–556, 2009.

[30] E. Quiring, A. Maier, and K. Rieck, "Misleading authorship attribution of source code using adversarial learning," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 479–496.

[31] R. Compton, E. Frank, P. Patros, and A. Koay, "Embedding java classes with code2vec: Improvements from variable obfuscation," *arXiv preprint arXiv:2004.02942*, 2020.

[32] vmarkovtsev, "Duplicates dataset," [EB/OL], https://github.com/src-d/datasets/tree/master/Duplicates Accessed June 3, 2020.

[33] R. Ferenc, Z. Tóth, G. Ladányi, I. Siket, and T. Gyimóthy, "A public unified bug dataset for java," in *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2018, pp. 12–21.

[34] A. Ben-David, "Comparison of classification accuracy using cohen's weighted kappa," *Expert Systems with Applications*, vol. 34, no. 2, pp. 825 – 832, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957417406003435

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

[36] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[37] J. Platt, "Sequential minimal optimization: A fast algorithm for training support vector machines," 1998.

[38] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[39] D. DeFreez, A. V. Thakur, and C. Rubio-González, "Path-based function embedding and its application to error-handling specification mining," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 423–433.

[40] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018, pp. 476–486.

[41] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 200–20 010.

[42] D. He, Y. Xia, T. Qin, L. Wang, N. Yu, T.-Y. Liu, and W.-Y. Ma, "Dual learning for machine translation," in *Advances in neural information processing systems*, 2016, pp. 820–828.

[43] W. Wang, Y. Zhang, Z. Zeng, and G. Xu, "Transˆ 3: A transformer-based framework for unifying code summarization and code search," *arXiv preprint arXiv:2003.03238*, 2020.

[44] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[45] P. Liu, X. Qiu, and X. Huang, "Adversarial multi-task learning for text classification," *arXiv preprint arXiv:1704.05742*, 2017.

[46] D. Dong, H. Wu, W. He, D. Yu, and H. Wang, "Multi-task learning for multiple language translation," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2015, pp. 1723–1732.

[47] B. McCann, N. S. Keskar, C. Xiong, and R. Socher, "The natural language decathlon: Multitask learning as question answering," *arXiv preprint arXiv:1806.08730*, 2018.

[48] M. Wulfmeier, A. Abdolmaleki, R. Hafner, J. T. Springenberg, M. Neunert, T. Hertweck, T. Lampe, N. Siegel, N. Heess, and M. Riedmiller, "Regularized hierarchical policies for compositional transfer in robotics," *arXiv preprint arXiv:1906.11228*, 2019.

[49] K. Hausman, J. T. Springenberg, Z. Wang, N. Heess, and M. Riedmiller, "Learning an embedding space for transferable robot skills," in *International Conference on Learning Representations*, 2018.

[50] A. Diba, M. Fayyaz, V. Sharma, M. Paluri, J. Gall, R. Stiefelhagen, and L. Van Gool, "Holistic large scale video understanding," *arXiv preprint arXiv:1904.11451*, 2019.

[51] R. Pasunuru and M. Bansal, "Multi-task video captioning with video and entailment generation," *arXiv preprint arXiv:1704.07489*, 2017.

[52] L. Deng, G. Hinton, and B. Kingsbury, "New types of deep neural network learning for speech recognition and related applications: An overview," in *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 8599–8603.

[53] M.-T. Luong, Q. V. Le, I. Sutskever, O. Vinyals, and L. Kaiser, "Multi-task sequence to sequence learning," *arXiv preprint arXiv:1511.06114*, 2015.

[54] X. Liu, P. He, W. Chen, and J. Gao, "Multi-task deep neural networks for natural language understanding," *arXiv preprint arXiv:1901.11504*, 2019.

[55] Y. Sun, S. Wang, Y.-K. Li, S. Feng, H. Tian, H. Wu, and H. Wang, "Ernie 2.0: A continual pre-training framework for language understanding." in *AAAI*, 2020, pp. 8968–8975.