AstBERT: Enabling Language Model for Financial Code Understanding with Abstract Syntax Trees

Rong Liang
Ant Group
liangrong.liang@antgroup.com

Tiehua Zhang[*]
Ant Group
zhangtiehua.zth@antgroup.com

Yujie Lu
Ant Group
lyj272836@antgroup.com

Yuze Liu
Ant Group
liuyuze.liuyuze@antgroup.com

Zhen Huang
Ant Group
hz101346@antgroup.com

Xin Chen
Ant Group
jinming.cx@antgroup.com

Abstract

Using the pre-trained language models to understand source codes has attracted increasing attention from financial institutions owing to the great potential to uncover financial risks. However, there are several challenges in applying these language models to solve programming language related problems directly. For instance, the shift of domain knowledge between natural language (NL) and programming language (PL) requires understanding the semantic and syntactic information from the data from different perspectives. To this end, we propose the AstBERT model, a pre-trained PL model aiming to better understand the financial codes using the abstract syntax tree (AST). Specifically, we collect a sheer number of source codes (both Java and Python) from the Alipay code repository and incorporate both syntactic and semantic code knowledge into our model through the help of code parsers, in which AST information of the source codes can be interpreted and integrated. We evaluate the performance of the proposed model on three tasks, including code question answering, code clone detection and code refinement. Experiment results show that our AstBERT achieves promising performance on three different downstream tasks.

1 Introduction

Programming language and source code analysis using deep learning methods have received increasing attention in recent years. Using pre-trained model such as such as BERT (Devlin et al., 2019), AlBERT (Lan et al., 2020) receive a great success on different NLP tasks. Inspired by that, some researchers attempt to apply this technique to comprehend source codes. For instance, CodeBERT (Feng et al., 2020) is a pre-trained model using six different programming languages from GitHub, demonstrating a good performance comparing with different embedding techniques.

[*]Corresponding Author

Although pre-trained models are now widely used for different purposes, it is rare to see how to apply such techniques to financial service codes. It is believed that re-training the model using financial service code could help uncover the code hazards before being released and circumvent any economic damage (Guo et al., 2021). Existing research points out that the use of domain knowledge is critical when it comes to training a well-performing model, and one way to solve this problem is to pretrain a model using specific domain corpora from scratch (Hellendoorn et al., 2019). However, pretraining a model is generally time-consuming and computationally expensive, and domain corpora are often not enough for pre-training tasks, especially in the financial industry, where the number of open-sourced codes is limited.

To this end, we propose an AstBERT model, a pre-trained language model aiming to better understand the financial codes using abstract syntax trees (AST). To be more specific, AST is a tree structure description of code semantics. Instead of using source code directly, we leverage AST as the prominent input information when training and tuning AstBERT. To overcome the token explosion problem that usually happens when generating the AST from the large-scale code base, a pruning method is applied beforehand, followed by a designated AST-Embedding Layer to encode the pruned code syntax information. To save the training time and resources, we adopt the pre-trained CodeBERT (Feng et al., 2020) as our inception model and continue to train on the large quantity of AST corpus. In this way, AstBERT can capture semantic information for both nature language (NL) and programming language (PL).

We train AstBERT on both Python and Java corpus collected from Alipay code repositories, which contains about 0.2 million functions in java and 0.1 million functions in python. Then we evaluate its performance on different downstream tasks. The main contributions of this work can be summarized as follows:

- We propose a simple yet effective way to enhance the pre-trained language model's ability to understand programming languages in the financial domain with the help of abstract syntax tree information.

- We conduct extensive experiments to verify the performance of AstBERT on code-related tasks, including code question answering, code clone detection and code refinement. Experiments results show that AstBERT demonstrates a promising performance for all three downstream tasks.

2 Related Work

In this part, we describe existing pre-trained models and datasets in code language interpretation in detail.

2.1 Datasets in Code Understanding

It is inevitable to leverage a high-quality dataset in order to pre-train a model that excels in code understanding. Some researchers have started to build up the dataset needed for the code search task in (Nie et al., 2016), in which different questions and answers are collected from Stack Overflow. Also, a large-scale unlabeled text-code pairs are extracted and formed from GitHub by (Husain et al., 2019). Three benchmark datasets are builed by (Heyman and Van Cutsem, 2020), each of which consists of a code snippet collection and a set of queries. An evaluation dataset developed by (Li et al., 2019) consists of natural language question and code snippet pairs. They manually check whether the questions meet the requirements and filter out the ambiguous pairs. A model trained by (Yin et al., 2018) on a human-annotated dataset is used to automatically mine massive natural language and code pairs from Stack Overflow. Recently, CoSQA dataset constructed by (Huang et al., 2021) that includes 20,604 labels for pairs of natural language queries and codes. CoSQA is annotated by human annotators and it is obtained from real-world queries and Python functions. It is rare to find open-sourced public source code in the financial domain, and we therefore retrieve both Python and Java code from the Alipay code repositories.

2.2 Models in Code Understanding

Using deep learning network to solve language-code tasks has been studied for years. A Multi-Modal Attention Network trained by (Wan et al., 2019) represents unstructured and structured features of source code with two LSTM. A masked language model(Kanade et al., 2019) is trained on massive Python code obtained from GitHub and used to obtain a high-quality embedding for source code. A set of embeddings (Karampatsis and Sutton, 2020) based on ELMo (Peters et al., 2018) and conduct bug detection task. The results prove that even a low-dimensional embedding trained on a small corpus of programs is very useful for downstream task. Svyatkovskiy et al. use GPT-2 framework and train it from scratch on source code data to support code generative task like code completion (Svyatkovskiy et al., 2020). CodeBERT (Feng et al., 2020) is a multi-PL (programming language) pretrained model for code and natural language, and it is trained with the new learning objective based on replaced token detection. C-BERT proposed by (Buratti et al., 2020) is pre-trained from C language source code collected from GitHub to do AST node tagging task.

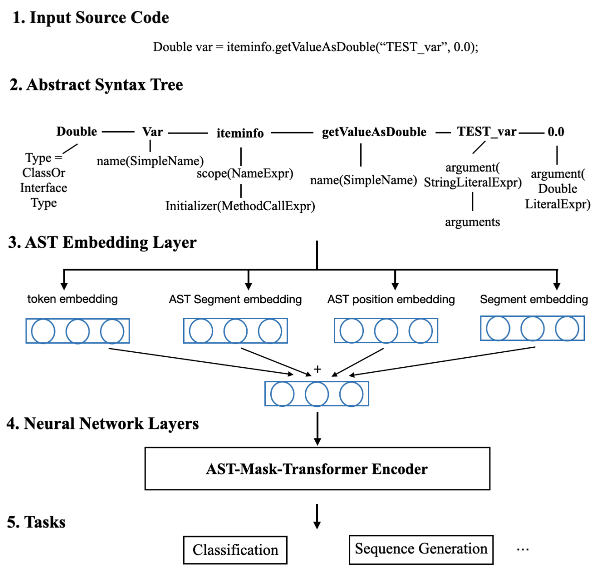
Different with previous work, AstBERT is a simple yet effective way to use pre-trained model in code interpretation field. Instead of training a large-scale model from scratch, it incorporates AST information into a common language model, from which the code understanding can be derived.

3 AstBERT

In this part, we describe the details about AstBERT.

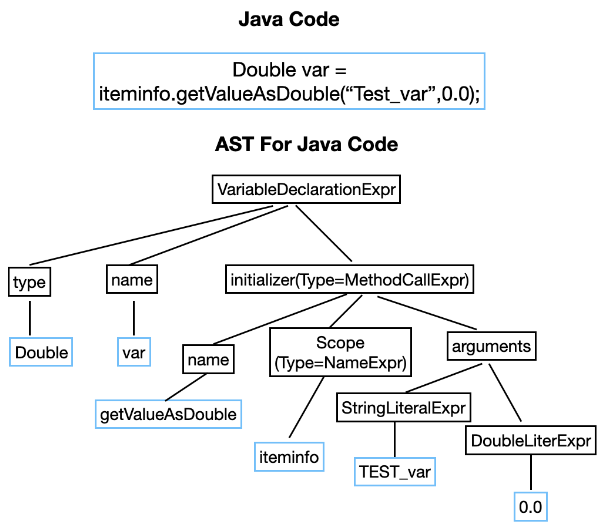
3.1 Model Architecture

Figure 1 shows the main architecture of AstBERT. Instead of using source code directly, the pruned AST information is used as the input. For each source code token, the AST information is attached in the front, and the position index is used to show the order of the input. There are four embedding modules at the AST embedding layer. Token embedding is similar to what is in BERT (Devlin et al., 2019), one key difference is that the vocabularies used are AST keywords. The token is then encoded to a vector format. Additionally, in AstBERT, AST-segment and AST-position are used to integrate the structure information of AST, the detail of their function will be introduced in subsection 3.3. After the AST embedding layer, the embedding vectors



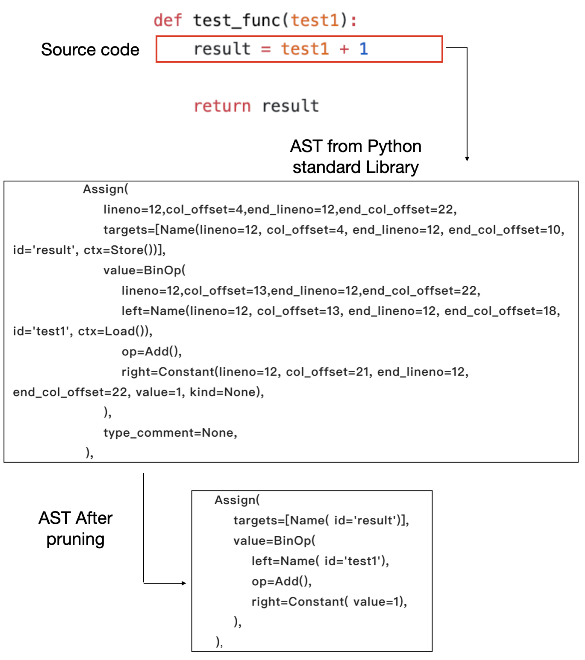
```
Double var = iteminfo.getValueAsDouble("TEST_var", 0.0);
```

Figure 1: The model structure of AstBERT: An easy and effective way to enhance pre-trained language model's ability for code understanding



```
Double var =
iteminfo.getValueAsDouble("Test_var",0.0);
```

Figure 2: AST-based code representation of a financial code snippet



```
def test_func(test1):
    result = test1 + 1

    return result
```

```
Assign(
    lineno=12,col_offset=4,end_lineno=12,end_col_offset=22,
    targets=[Name(lineno=12, col_offset=4, end_lineno=12, end_col_offset=10,
id='result', ctx=Store())],
    value=BinOp(
        lineno=12,col_offset=13,end_lineno=12,end_col_offset=22,
        left=Name(lineno=12, col_offset=13, end_lineno=12, end_col_offset=18,
id='test1', ctx=Load()),
        op=Add(),
        right=Constant(lineno=12, col_offset=21, end_lineno=12,
end_col_offset=22, value=1, kind=None),
    ),
    type_comment=None,
),
```

```
Assign(
    targets=[Name( id='result')],
    value=BinOp(
        left=Name( id='test1'),
        op=Add(),
        right=Constant( value=1),
    ),
),
```

Figure 3: AST pruning process

are then forwarded to a multi-layer bidirectional AST-Mask-Transformer encoder (Vaswani et al., 2017) to generate hidden vectors. The difference is that we use AST-Mask-Self-Attention instead of Self-Attention to calculate the attention score, the detail of which will be unveiled in subsection 3.4. In the output layer, the hidden vectors generated by AST-Mask-Transformer encoder will be used for classification or sequence generation tasks.

3.2 Input and Pruning

We introduce the pruning process in this part. As shown in Figure 2, the AST contains the complete information of the source code and provide the brief description for each token. For example, the *getValueAsDouble* is the name for *MethodCallExpr* (one of the AST node types) and the *TEST_var* is an argument for *MethodCallExpr*. We know the *Double* is a type of the variable *var* from AST. Such AST information reveals the semantic knowledge of the source code.

In general, the length of AST from the compiled codes is greater than the plain source code, as shown in Figure 3, the AST from Python standard library contains a number of nodes such as *lineno*, *endlineno* and so on. Taking the snippet *result* = *test1* + *1* as an example, both the original and pruned AST trees can be seen in Figure 3. It is clearly noticed that there exists a large amount of redundant information such as line number and code column offset in the original AST tree, leading to intractable AST exploration problem for large code corpus (Wan et al., 2019). Therefore, after generating AST, we will prune this tree by removing the meaningless and uninformed node to avoid unintended input for the model.

3.3 AST Embedding Layer

As mentioned above, we use the pruned AST as the input for model, and it will pass AST embedding layer first. The details of the AST embedding layer are unveiled in Figure 4, from which token-embedding vectors, AST-segment embedding vectors, AST-position embedding vectors and segment

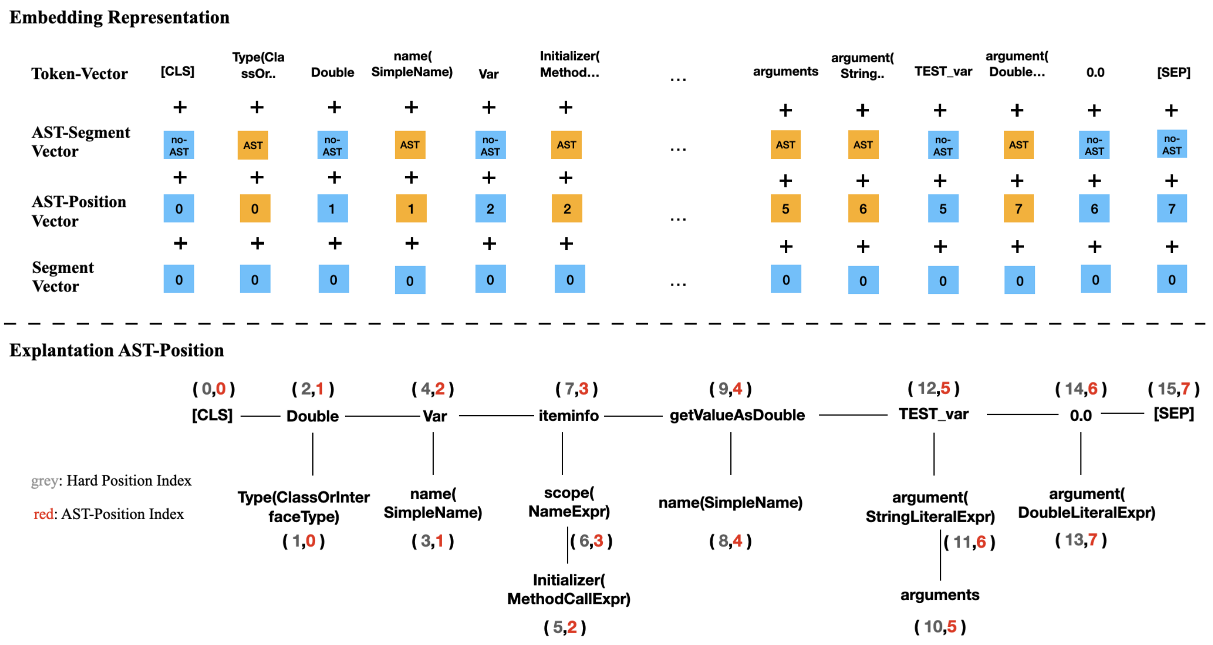


Figure 4: The overview of AST embedding representations

embedding vectors are generated. Taking the code snippet in Figure 2 as an example, we can see the additional AST information account for most of the tokens in the input, which unexpectedly causes changes in the meaning of the original code. To prevent this from happening, we use AST-segment embedding to distinguish between AST tokens and source code tokens. It is known that in BERT all the order information for input sequence is contained in the position embedding, allowing us to add different position information for input. Here, except for the AST-segment, we use an index combination of hard-position and AST-position to convey the order information. As seen in Figure 4, the index combination of *name(SimpleName)* is *(3,1)*, which means it locates at the 3rd position in the input sequence dimension while being the 1st AST token. In the front of *name(SimpleName)*, there is only one extra AST token named *Type(ClassOrInterfaceType)*. Segment embedding is similar to BERT. The output of the embedding layer is simply the sum of all embedding vectors from these four parts. The result is then passed into the AST-Mask transformer encoder to generate hidden vectors.

3.4 AST-Mask Transformer

Since the branch in AST contains the specific semantic knowledge to describe the role of the code token, it is rational to make AST tokens only contribute to the code tokens on the same branch. For example, in

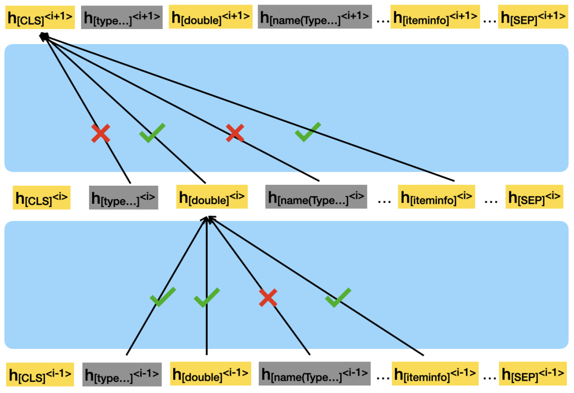


Figure 5: The explanation of AST-Mask-Transformer

Figure 2, [*Type=ClassOrInterfaceType*] only describe the role of the [*Double*] and has nothing to do with [*Var*]. Therefore, the embedding of [*Var*] should not be affected by [*Type=ClassOrInterfaceType*]. As demonstrated in Figure 5, the *Type=ClassOrInterfaceType* should not make a contribution to the embedding of [*CLS*] tag that often used for classification bypass the [*Double*]. This is because that the [*Type=ClassOrInterfaceType*] is a tag in the branch of [*Double*] and should only correlate to [*Double*]. To prevent the AST information injection from changing the semantic of the input, AstBERT employs Mask-Self-Attention(Xu et al., 2021) to limit the self-attention region in Transformer(Vaswani et al., 2017). We use AST matrix *M* to describe whether the AST token and code token are on the same branch, M_{AST} is defined as follow:

$$M_{AST_{i,j}} = \begin{cases} 1 & w_i \oplus w_j \\ 0 & w_i \otimes w_j \end{cases} \quad (1)$$

where, $w_i \oplus w_j$ indicates that w_i and w_j are on the same AST branch, while $w_i \otimes w_j$ are not. i and j are the AST-position index. The AST mask matrix is then used to calculate the self-attention scores. Formally, the AST-mask-self-attention is defined as follow:

$$Q^{i+1}, K^{i+1}, V^{i+1} = h^i W_q, h^i W_k, h^i W_v \quad (2)$$

$$S^{i+1} = softmax(\frac{{K^{i+1}}^T Q^{i+1} M_{AST}}{\sqrt{d_k}}) \quad (3)$$

$$h^{i+1} = S^{i+1} V^{i+1} \quad (4)$$

where W_q, W_k, W_v are trainable model parameters. h^i is the hidden state from the ith AST-mask-self-attention blocks. d_k is the scaling factor. If h^i_k and h^i_j are not in same AST branch, the $M_{AST_{kj}}$ will make the attention score S^{i+1}_{kj} to 0, which means h^i_k makes no contribution to the hidden state of h^i_j.

We collect massive Python and Java codes from Alipay code repositories and generate the AST for these source code (Python code using standard AST API, Java code using Javaparser). We use these processed AST information to continue the pre-train of the model. The technique of pre-training is inspired by the masked language modeling (MLM), which is proposed by (Devlin et al., 2019) and proven effective.

4 Experiments

We test the performance of our proposed model on different code understating tasks using the different released test datasets. We also look into the ablation studies.

4.1 Dataset

Code Question Answering CoSQA (Huang et al., 2021) consists of 20,604 query-code pairs collected from the Microsoft Bing search engine. We randomly split CoSQA into 20,000 training and 604 validation examples. We also build AliCoQA dataset based on the code collected from the Alipay code repositories. We use the search logs from AntCode search engine as the source of queries and manually design heuristic rules to find the queries of code searching intent. For example, queries with the word of *tutorial* or *example* are likely to locate a programming description rather than a code function, so we remove such queries. Then, we use the CodeBERT matching model (Feng et al., 2020) to retrieve high-confidence codes for every query and manually check 5,000 query-code pairs to construct AliCoQA. We randomly split AliCoQA into 4,500 training and 500 validation samples.

Code Clone Detection We use BigCloneBench dataset (Svajlenko et al., 2014) and discard samples with no labels. Finally, we randomly split it into 901,724 training set and 416,328 validation set.

Code Refinement BFP (Tufano et al., 2019) dataset constains two subsets based on the code length. For BFP_small dataset, the numbers of training and validation are 46,680 and 5,835, respectively. For the BFP_medium dataset, the numbers of training and validation are 52,364 and 6,545. We collect code from Alipay code repositories and build AliCoRF dataset. Firstly, we identify commits having a message containing the words, such as *fix*, *solve*, *bug*, *problem* and *issue*. Following that, for each bug-fixing commit, we extract the source code before and after the bug-fix. Finally, we manually check 9,000 bug-fix pairs to construct AliCoRF and randomly split it into 8,000 training set and 1,000 validation set.

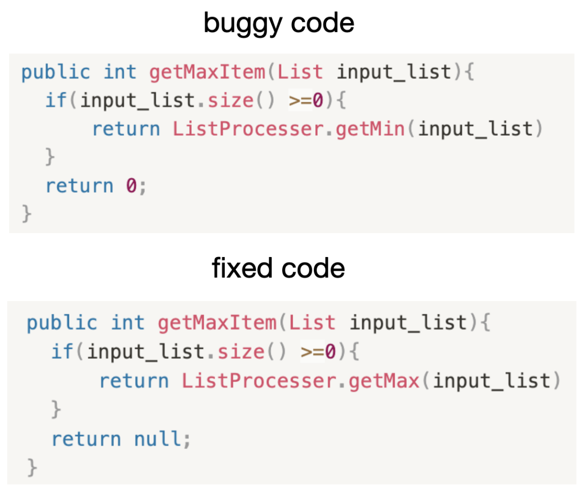
Evaluation Metric Following the settings in the previous work, we use accuracy as the evaluation metric on code question answering, and F1 score on code clone detection. We also use accuracy as the evaluation metric on code refinement, in which only the example being detected and fixed properly will be considered successfully completing the task. We give one example case for this task in Figure 6. In this example, the model successfully fixes the method name from *getMin* to *getMax*.

4.2 Parameter Settings

We follow the similar parameter settings in previous works (Huang et al., 2021; Svajlenko et al., 2014; Tufano et al., 2019). On code question answering task, we set dropout rate to 0.1, maximum sequence length to 512, learning rate to 1e-5, warm-up rate to 0.1 and batch size to 16. On code clone detection task, learning rate is set to be 2e-5, batch size to be 16 and maximum sequence length to be 512. On code refinement task, we set learning rate to 1e-4, batch size to 32 and maximum

		Datasets					
		ACC					F1
TASK	Model	AliCoQA	CoSQA	BFP_small	BFP_medium	AliCoRF	BigCloneBench
Question Answering	BERT	0.402	0.399	\	\	\	\
	RoBERTA	0.434	0.421	\	\	\	\
	CodeBERT	0.532	0.526	\	\	\	\
	AstBERT	**0.588**	**0.571**	\	\	\	\
Code Refinement	LSTM	\	\	0.100	0.025	0.111	\
	Transformer	\	\	0.147	0.037	0.152	\
	CodeBERT	\	\	0.164	0.052	0.176	\
	GraphCodeBERT	\	\	0.173	**0.091**	0.182	\
	AstBERT	\	\	**0.176**	0.089	**0.183**	\
Code Clone	CDLH	\	\	\	\	\	0.820
	ASTNN	\	\	\	\	\	0.930
	FA-AST-GMN	\	\	\	\	\	0.950
	RoBERTa	\	\	\	\	\	0.957
	CodeBERT	\	\	\	\	\	0.965
	GraphCodeBERT	\	\	\	\	\	0.971
	AstBERT	\	\	\	\	\	**0.973**

Table 1: Experiment results of different tasks on different dataset



```
public int getMaxItem(List input_list){
  if(input_list.size() >=0){
      return ListProcesser.getMin(input_list)
  }
  return 0;
}
```

```
public int getMaxItem(List input_list){
  if(input_list.size() >=0){
      return ListProcesser.getMax(input_list)
  }
  return null;
}
```

Figure 6: One case of AstBERT output for code refinement task.

sequence length to 256. For all experiments, we use the Adam optimizer to update model parameters (Kingma and Ba, 2015).

4.3 Results of Code Question Answering

We use CoSQA (Junjie Huang et al. 2021) dataset to verify the code question answering task. In this task, the test sample is the query-code pair and labeled as either "1" or "0", indicating whether the code can answer the query. These query-code pairs are collected from Microsoft Bing search engine and annotated by human. We train different benchmark models using our dataset and evaluate the performance of each on CoSQA for code question answering:(i) BERT proposed by (Devlin et al., 2019); (ii) RoBERTA proposed by (Liu et al., 2019); (iii) CodeBERT proposed by (Feng et al., 2020); and (iv) AstBERT. From Table 1, we can see the BERT and RoBERTA achieve a similar yet relative low Acc score in this task. This is because these two models are pre-trained by natural language corpus and not integrated with any code-related domain knowledge. CodeBERT achieves a better performance than the RoBERTA, similar to the results published by (Huang et al., 2021). Our AstBERT achieves the best performance compared with all benchmarks. This clearly demonstrates that the integration of AST information into the model can further improve model's ability for understanding semantic and syntactic information in the codes. We also evaluate our AstBERT on AliCoQA and the results show that in financial domain dataset AstBERT also achieves the best performance.

	Datasets				
	ACC				F1
Model	CoSQA	AliCoQA	BFP	AliCoRF	BigCloneBench
AstBERT	**0.571**	**0.588**	**0.176**	**0.183**	**0.973**
-w/o AST-position	0.552	0.558	0.174	0.181	0.970
-w/o AST-Mask-Self-Attention	0.539	0.544	0.165	0.178	0.966

Table 2: Ablation study

4.4 Results of Code Clone Detection

Code clone detection is an another task when it comes to measuring the similarity of code-code pair, which can help reduce the cost of software maintenance. We use BigCloneBench (Svajlenko et al., 2014) dataset for this task and treat this task as a binary classification to fine-tune AstBert. The experimental results are also shown in the Table 2. The **CDLH** model is proposed by (Wei and Li, 2017) to learn representations of code by AST-based LSTM and use hamming distance as optimization objective. The **ASTNN** model (Zhang et al., 2019) encodes AST subtrees by RNNs to learn representation for code. The **FA-AST-GMN** model (Wang et al., 2020) uses a flow-augmented AST as the input and leverages GNNs to learn the representation for a program. The **GraphCodeBERT** (Guo et al., 2021), which is a pre-trained model using data flow at the pre-training stage to leverage the semantic-level structure of code, learns the representation of code. The experiment shows that our AstBERT achieves the best results in code clone detection task.

4.5 Results of Code Refinement

In general, code refinement is the task of locating code defects and automatically fixing them, which has been considered critical to uncovering any financial risks. We use both BFP_small and BFP_medium datasets (Tufano et al., 2019) to verify the performance of all models and show results in the Table 1. This is a Seq2Seq task, and we record relevant accuracy for each benchmark model. We take the results of **LSTM** and **Transformer** as recorded in (Guo et al., 2021). It is observed in the table that **Transformer** outperforms **LSTM**, which indicates that **Transformer** has a better ability of learning the representation of code. Both **CodeBERT** and **GraphCodeBERT** are pre-trained models, which present state-of-the-art results at their time. Our **AstBERT** achieves a better performance than other pre-trained models on BFP_small dataset, while obtaining the competitive result on BFP_medium dataset. This again demonstrates the effectiveness of incorporating the AST information in the pre-trained model is helpful to the code understanding, including the code refinement task.

4.6 Ablation Studies

In this subsection, we explore the effects of the AST-position and AST-Mask-Self-Attention for AstBERT on three tasks. "**w/o AST-position**" refers to fine-tuning AstBERT without AST-position. "**w/o AST-Mask-Self-attention**" means that each token in input, regardless of its position in the AST tree, calculates the attention scores with other tokens. As shown in Table 2, we have made the following observations: (i) Without AST-position or AST-Mask-Self-Attention, the performance of AstBERT on code question answering has shown a clear decline; (ii) It also can be seen that the model without AST-Mask-Self-Attention demonstrates an even worse performance than without AST-position, which confirms sufficient AST tokens can help incorporate the syntactic structures of the code. The same trend can also be observed on code clone detection and code refinement. We can conclude that the AST-position and the AST-Mask-Self-Attention play a pivotal role in incorporating the AST information into the model.

5 Conclusion

In this paper, we propose AstBERT, a simple and effective way to enable pre-trained language model for financial code understanding by integrating semantic information from the abstract syntax tree (AST). In order to encode the structural information, AstBERT uses a designated AST-Segment and AST-Position in the embedding layer to make model incorporate such AST information. Following that, we propose the AST-Mask-Self-Attention to limit the region when calculating attention scores, preventing the input from deviating from its original meaning. We conduct three different code understanding related tasks to evaluate the performance of the AstBERT. The experiment results show that AstBERT outperforms baseline models on both code question answering and clone detection. For code refinement task, the model achieves state-of-the-art performance on BFP_small dataset and competitive performance on BFP_medium dataset.

References

Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, et al. 2020. Exploring software naturalness through neural language models. *arXiv preprint arXiv:2006.12641*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, pages 4171–4186.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. In *EMNLP*, pages 1536–1547.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. Graphcodebert:

Pre-training code representations with data flow. In *ICLR*, pages 1–21.

Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2019. Global relational models of source code. In *ICLR*, pages 1–12.

Geert Heyman and Tom Van Cutsem. 2020. Neural code search revisited: Enhancing code snippet retrieval through natural language intent. *arXiv preprint arXiv:2008.12193*.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. Cosqa: 20,000+ web queries for code search and question answering. *arXiv preprint arXiv:2105.13239*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2019. Pre-trained contextual embedding of source code. In *arXiv preprint arXiv:2001.00059*.

Rafael-Michael Karampatsis and Charles Sutton. 2020. Scelmo: Source code embeddings from language models. *arXiv preprint arXiv:2004.13214*.

Diederik P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *ICLR*, pages 1269–1272.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. Albert: A lite bert for self-supervised learning of language representations. In *ICLR*, pages 1–17.

Hongyu Li, Seohyun Kim, and Satish Chandra. 2019. Neural code search evaluation dataset. *arXiv preprint arXiv:1908.09804*.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Liming Nie, He Jiang, Zhilei Ren, Zeyi Sun, and Xiaochen Li. 2016. Query expansion based on crowd knowledge for code search. *IEEE Transactions on Services Computing*, 9(5):771–783.

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*.

Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *ICSME*, pages 476–480.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. In *ESEC/FSE*, pages 1433–1443.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology*, 28(4):1–29.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NeurIPS*, pages 5998–6008.

Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. In *ASE*, pages 13–25.

Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *SANER*, pages 261–271.

Huihui Wei and Ming Li. 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *IJCAI*, pages 3034–3040.

Wenwen Xu, Mingzhe Fang, Li Yang, Huaxi Jiang, Geng Liang, and Chun Zuo. 2021. Enabling language representation with knowledge graph and structured semantic information. In *CCAI*, pages 91–96.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *MSR*, pages 476–486.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *ICSE*, pages 783–794.