# BERT-Based Code Learning for Exception Localization and Type Prediction

**Chongyu Zhang, Qiping Tao, Liangyu Chen*, Min Zhang**

Shanghai Key Laboratory of Trustworthy Computing
East China Normal University
lychen@sei.ecnu.edu.cn

**Abstract**

Exception handling is crucial but challenging in program development. It needs to identify and handle all potential exceptions within programs to ensure system security and stabilization. Traditional exception handling relies on the expertise and experience of programmers, which often leads to oversights. Therefore, identifying exceptional code and recommending handling solutions are hot research topics with significant practical value. This paper presents a model called CodeHunter for exception localization and type prediction. The model first utilizes BERT-based model to represent code features and then uses Bi-LSTM for sequence labeling to pinpoint exceptional code. Additionally, this model also considers contextual features of the exception code and learns weights for the code within the try block and its context through the self-attention mechanism. Subsequently, it performs exception localization and predicts exception types. We conduct experiments on three different datasets. The results demonstrate that in the task of exception localization, our model can achieve a maximum accuracy of 98.6%, exceeding SOTA baselines by 11.2%. In the task of exception type prediction, our model can surpass the accuracy of SOTA baselines by a maximum of 18.7%, achieving 92.0% Top-1 accuracy. The rationality of techniques used in our model is also proved by the ablation testing. The model is implemented as an IDE plugin for programming convenience.

## 1 Introduction

Exception handling plays an important role in software development, involving the anticipation, capturing, and handling of unexpected situations. An effective mechanism for exception handling is to improve code reliability, robustness, and maintainability (Tufano et al. 2019). However, as modern software systems continue to grow in complexity and scale, it becomes very challenging to accurately identify the code that might cause exceptions (Liu et al. 2021). In traditional software development, programmers need to handle exceptions in manual ways based on their expertise and experience. However, it may cause some problems, such as unclear exception classification, insufficient exception stack information, inconsistent exception handling, and missing exception explosion. Improper exception handling can lead the system into chaos and make code hard to maintain. Therefore, good exception handling is the base of software quality and performance (Wang et al. 2023a).

To address these problems above, researchers have proposed various methods and techniques for exception handling. Among them, early approaches based on code static analysis have achieved certain results. These methods can effectively extract and utilize syntax features, then yield favorable outcomes in exception handling. However, these methods face challenges in computation and storage resources, which make it difficult to maintain their generalization and effectiveness when dealing with large-scale code (Abdelaziz et al. 2023).

With the rise of artificial intelligence (AI) technology in recent years, the AI-based approaches (Allamanis et al. 2018) have gained widespread attention in the field of exception handling. These methods demonstrate promising performance by learning from massive sample data (Niu et al. 2022). They also have the analysis capacity for large-scale code (Li et al. 2022). However, most of them still have room to improve the representation of static and dynamic features from code. Additionally, the majority of these AI-based methods lack comprehensive consideration of contextual information surrounding exceptions and thus fail to fully understand the environment and conditions in which exceptions occur (Henkel et al. 2018).

To tackle these limitations, this paper proposes a model called CodeHunter for exception localization and type prediction. CodeHunter uses the BERT-based (Bidirectional Encoder Representations from Transformers) model (Devlin et al. 2018) for code representation enhancement. Moreover, in order to better capture the contextual information and sequence relationships, CodeHunter utilizes the Bi-LSTM (Bidirectional Long Short-Term Memory) network (Zhou et al. 2016), which serves as a sequence labeler for exception identification and classification. Additionally, CodeHunter incorporates code preprocessing techniques, to obtain more syntax features, aiming for better model performance.

The main contributions of this paper are listed as follows.

- We propose the CodeHunter model. For the exception localization task, it uses BERT-based model for feature representation and Bi-LSTM for sequence labeling to identify code blocks that may generate exceptions. For the task of exception type prediction, it inputs the code before the try blocks and the code in the try blocks separately into different BERT-based models for feature representation and learns weights through the self-attention mechanism to predict exception types ultimately.

- In order to extract more language-related code syntax information, we utilize AST (Abstract Syntax Tree) parsing and variable name substitution in the data preprocessing stage. These techniques of feature enhancement have been proved effective in model performance through ablation testing.
- We conduct experiments to compare CodeHunter with other baselines on three public datasets. The results indicate that our model is significantly better than other baselines. In the task of exception localization, CodeHunter has an improvement of $11.2\%$ in accuracy compared to other SOTA baselines. In the task of exception type prediction, CodeHunter achieves a maximum of $18.7\%$ improvement in accuracy. In addition, ablation experiments are conducted to validate the rationality of the techniques used in our model.

The remainder of this paper is organized as follows. We first summarize the related work (§2). Then we explain the structure of CodeHunter in detail (§3). Furthermore, we carry out experiments from several perspectives to validate our model (§4). Finally, we conclude this paper (§5).

## 2 Related Work

Exception handling is crucial in programming, and various methods and technologies have been proposed to enhance the efficiency and accuracy of exception handling. According to the differences in handling approaches, these works can be categorized into two types: methods based on code static analysis and methods based on artificial intelligence.

### 2.1 Methods Based on Code Static Analysis

These methods mainly focus on techniques including static code analysis, flow control, etc. They mainly conduct exception identification and handling from the perspective of program structure and syntax, focusing on recognizing and summarizing patterns. These methods are applied widely and achieve certain effectiveness in the early stages.

Barbosa et al. introduced a heuristic recommendation strategy that constructs a solution space by extracting information from code and its exception handling specification (Barbosa and Garcia 2018). This method then traverses the solution space to generate effective recommendations and ranks them. However, this approach cannot be easily applied to new projects without further customization. It requires building a solution space each time and is difficult to apply in large software systems.

Thummalapenta et al. mined association rules of method invocation sequences within try-catch statement blocks and applied them to detect behaviors that violate exception handling strategies (Thummalapenta and Xie 2009). Although this approach is helpful in improving the quality of exception handling, its effectiveness is limited because it assumes that try-catch blocks are entirely written by developers. Additionally, this approach cannot be directly applied to new projects without further customization. It requires mining association rules of method invocation sequences each time.

Nguyen et al. proposed a tool, ExAssist, for exception handling (Nguyen, Vu, and Nguyen 2019). This tool utilizes the fuzzy set theory and an N-gram model to predict potential exception types and recommend appropriate exception handling codes for some kinds of exceptions. The expected effect of the tool is similar to the goal of this paper, but it is unsuitable to be deployed in large-scale datasets.

### 2.2 Methods Based on Artificial Intelligence

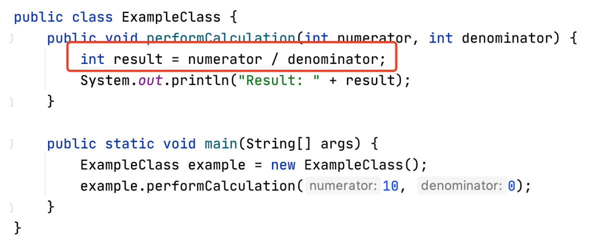
These methods utilize artificial intelligence techniques to learn modes and patterns automatically from massive data in order to achieve the automated handling of exceptions. Compared to traditional methods based on code static analysis, they have stronger generalization and higher accuracy.

Lee et al. proposed NPEX (Lee, Hong, and Oh 2022). This is a new technique that can fix *null pointer exceptions* (NPEs) without using test cases. NPEX can automatically infer the repair specifications for exceptions and use the inferred specifications to verify patches. The key idea is to learn a statistical model that predicts how developers will handle NPE by mining empty handling patterns from existing code repositories and using symbolic execution variants that can use the model to infer repair specifications from exceptional programs. However, this method is only used to fix NPEs and cannot effectively fix other exceptions.

Farima et al. proposed a prediction method D-REX (Farmahinifarahani et al. 2021) for Java runtime exceptions. The core of D-REX is a deep learning model that utilizes the representation learning ability of neural networks to infer a set of signals from code to predict related runtime exception types. The accuracy of D-REX in predicting runtime exception types reaches $81\%$, which is higher than multiple baselines without Transformer for at least $10\%$.

Zhang et al. proposed an exception handling model Nexgen (Zhang et al. 2020), which combines the attention mechanism and Bi-LSTM to handle exceptions in Java code. This model can determine the locations of possible exceptions and automatically generate the catch blocks. However, the accuracy of this model is not satisfied. More than $70\%$ of the generated catch blocks cannot be directly used. It is very difficult to generate catch blocks that completely meet the expectations of developers because each project has its unique requirements and each developer has his/her preference. Therefore, we believe that it is fully sufficient to inform developers of the possible locations and corresponding types of exceptions that may occur. The statements of catch blocks should be written by the developers according to their personal preferences and project requirements.

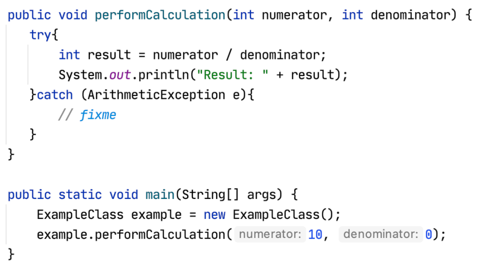
At the beginning of this research, we investigated LLMs for exception prediction. We used the Nexgen dataset to predict exception types on GPT3.5, which was most powerful at that time. However, the accuracy is only $24.08\%$, which is much lower than the existing SOTA models (over $70\%$). This indicates that GPT3.5 performs not very well in this problem, so we do not consider it as a baseline in the paper. Of course, if the prompts are adjusted carefully, the accuracy



```
public class ExampleClass {
    public void performCalculation(int numerator, int denominator) {
        int result = numerator / denominator;
        System.out.println("Result: " + result);
    }

    public static void main(String[] args) {
        ExampleClass example = new ExampleClass();
        example.performCalculation( numerator: 10,  denominator: 0);
    }
}
```

Figure 1: A code segment without try block.



```
public void performCalculation(int numerator, int denominator) {
    try{
        int result = numerator / denominator;
        System.out.println("Result: " + result);
    }catch (ArithmeticException e){
        // fixme
    }
}

public static void main(String[] args) {
    ExampleClass example = new ExampleClass();
    example.performCalculation( numerator: 10,  denominator: 0);
}
```

Figure 2: A code segment with try-catch block.

of GPT3.5 may increase, but we believe that it is difficult to bridge the performance gap.

## 2.3 Summary of Related Work

The traditional methods based on static code analysis mainly focus on program structure and syntax to improve exception handling. These methods often require a lot of human resources and cannot achieve good results on large datasets. AI-based methods utilize artificial intelligence techniques to learn modes and patterns from massive data in order to achieve automation and optimization of exception handling. As more datasets are constructed and more artificial intelligence technologies are invented, AI-based methods have more advantages for exception handling.

## 3 Methodology

## 3.1 Problem Formulation

To illustrate the problems handled by this model, we use a simple example to show exception handling. In Figure 1, the method *performCalculation* performs a division operation and prints the result. However, if the denominator is 0, an exception *ArithmeticException* will be thrown. If the exception is not handled correctly, it may cause the program to stop and lead to serious consequences. Experienced programmers usually write code as shown in Figure 2 to handle exceptions and prevent programs from problems. However, a large number of developers are unaware of the possibility of exceptions arising in certain locations. They may not handle the exceptions correctly. Alternatively, they may not know which type of exception may occur even though they know that an exception will occur, so the try-catch blocks written by them cannot effectively solve the exception. The main problems proposed in this paper are listed as follows:

- Exception Localization: which code may raise exceptions?
- Exception Type Prediction: which type of exception may be raised?

We design two following models (§3.2, §3.3) to solve the above problems respectively.

## 3.2 The Model for Exception Localization

**Overview** To solve exception handling problems, the first step is to predict which code is prone to generate exceptions. Given a code segment, every line may have the possibility to raise an exception. Thus, we transform the problem of exception localization into a sequence labeling problem. Each line of code will be predicted with a labeling, where 1 indicates that this line may raise an exception and 0 means no exception. In this way, it is possible to determine the lines that potentially raise exceptions. The exception localization model is shown in Figure 3 and explained as follows.

**Data Preprocessing** In the stage of data preprocessing, we use two techniques that can improve the quality of the dataset and enhance the model performance. The first technique is to parse code with AST (abstract syntax tree) notation and remove the unused code. Then, the corresponding AST tag is generated by preorder traversal on the AST tree and concatenated as the last token at the end of the statement, so that the model can perceive the syntax features of each statement. The process of AST parsing is shown in *supplementary material*. The second technique is to replace the variable names with their corresponding class names. In this way, the class information corresponding to the variable can be retained in the tokenizer stage of the BERT-based model.

**Feature Representation Layer** After the data preprocessing is completed, the feature representation layer of the model represents the code features by using CodeT5+ (Wang et al. 2023b). The feature representation layer fine-tunes CodeT5, inputs the tokens of each line of code obtained in the preprocessing stage, and outputs the tensor representation corresponding to each line of code. Since the tensors corresponding to each line of code are independent of others, it is necessary to associate each line of code in a method through a sequence labeling layer to make full use of context information and then predict whether each line of code will raise exceptions.

**Sequence Labeling Layer** By using Bi-LSTM as a sequence annotator, the input BERT hidden layer tensor is processed for sequence labeling to identify lines that may raise exceptions in the sequence labeling layer. During this process, each token in the method will be annotated to identify rows that may raise exceptions. As shown in Eq.(1) and Eq.(2), $h^0$ represents the sentence vector output by the BERT layer, LN represents the layer normalization operation of Bi-LSTM, MHAtt represents the multi-head attention mechanism, the superscript $\ell$ represents the $\ell$-th layer

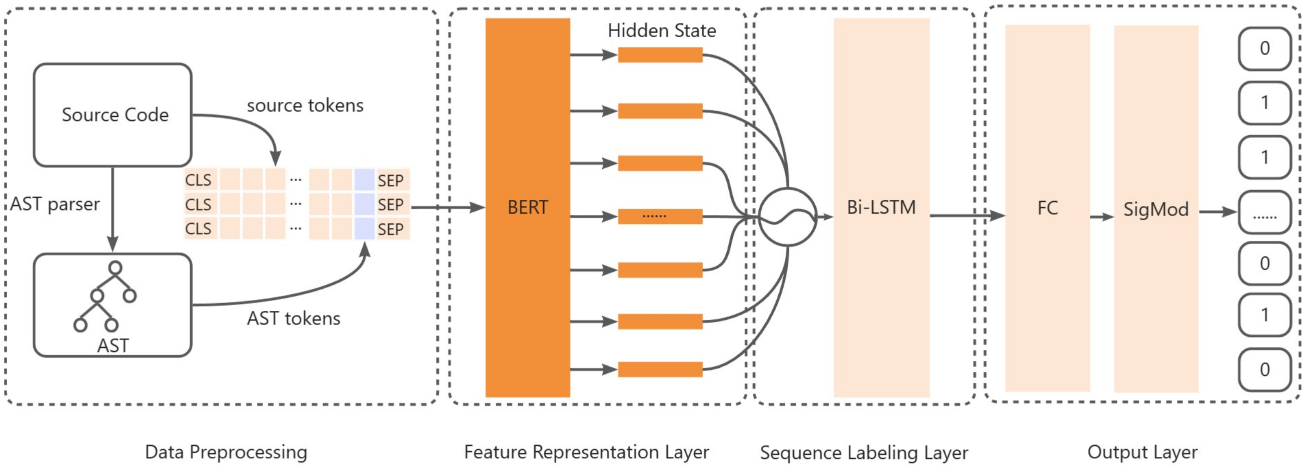


Figure 3: The structure of the model for exception localization.

of Bi-LSTM.

$$\tilde{h}^{\ell} = \mathrm{LN}\left(h^{\ell-1} + \mathrm{MHAtt}\left(h^{\ell-1}\right)\right) \tag{1}$$

$$h^{\ell} = \mathrm{LN}\left(\tilde{h}^{\ell} + \mathrm{FFN}\left(\tilde{h}^{\ell}\right)\right) \tag{2}$$

**Output Layer** The final output layer uses a Sigmod classifier, as shown in Eq.(3), where $h_i$ is the output vector of the last layer of Bi-LSTM and $i$ represents the line index in the code. The label $Y_i$ represents the final output, which is used to identify whether an exception has been raised.

$$\hat{Y}_i = \sigma\left(W_o h_i^{\ell} + b_o\right) \tag{3}$$

## 3.3 The Model for Exception Type Prediction

**Overview** After identifying which code lines may raise exceptions, we need to predict the type of potential exception. Essentially, this prediction is a classification problem. The overall solution adopted in this paper is to enrich the features of the dataset by using techniques such as AST parsing in the data preprocessing stage, similar to the exception localization model, and then process the dataset into the format of BERT-based model. The input of the model includes two main parts: the code before the try block and the code in the try block. These two parts are inputted into the model respectively, and finally, the model predicts a label of exception type will be generated. These labels represent the possible types of exceptions that the code may raise. The model structure is shown in Figure 4.

**Data Preprocessing** Similar to the processing of exception localization, we use AST parsing to generate new code tokens for feature enrichment and concatenate them with source code tokens. In addition, we divide all tokens into two main parts: code tokens before the try block and code tokens in the try block. The latter are very important because they raise the exceptions actually, while the former also cannot be ignored, because they have a certain relation with the exceptions and may indirectly cause them. Therefore, these two parts are considered in the model for training respectively.

**Feature Representation Layer** The feature representation layer uses BERT-based model for feature representation. Inspired by the Deep Structured Semantic Model (Huang et al. 2013), code tokens before the try block and code tokens in the try block are inputted into the BERT-based model respectively, to represent the code features. Then, the last hidden layers of the BERT-based model are assigned different weights. The weight of the hidden layer tensor of the code in the try block is higher, and the weight size is learned through the self-attention mechanism. Using the weighted feature representation as input to the next layer, which is the sequence labeling layer, can enable the model to fully obtain feature information and achieve better results.

**Sequence Labeling Layer** The sequence labeling layer uses two Bi-LSTM models for sequence labeling to identify potential exception types. It processes the input BERT hidden layer tensor, and two independent Bi-LSTM models process the BERT hidden layer tensor of the code before the try block and the BERT hidden layer tensor of the code in the try block. During this process, all tokens will be annotated. As shown in Eq.(4) and Eq.(5), $h^0$ represents the sentence vector output from the BERT layer, and LN denotes the layer normalization operation of Bi-LSTM. MHAtt represents the multi-head attention mechanism, and the superscript $\ell$ denotes the $\ell$-th layer of Bi-LSTM. As shown in Eq.(6), $h_1$ represents the last hidden layer output after the code in the try block is fed into Bi-LSTM, while $h_2$ represents the last hidden layer output after the code before the try block is fed into Bi-LSTM. Here, $\lambda$ represents the weight parameters learned with the self-attention mechanism.

$$\tilde{h}^{\ell} = \mathrm{LN}\left(h^{\ell-1} + \mathrm{MHAtt}\left(h^{\ell-1}\right)\right) \tag{4}$$

$$h^{\ell} = \mathrm{LN}\left(\tilde{h}^{\ell} + \mathrm{FFN}\left(\tilde{h}^{\ell}\right)\right) \tag{5}$$

$$h = h_1 + \lambda h_2 \tag{6}$$

**Output Layer** As shown in Eq.(7), the final output layer includes a fully connected layer and a Softmax activation

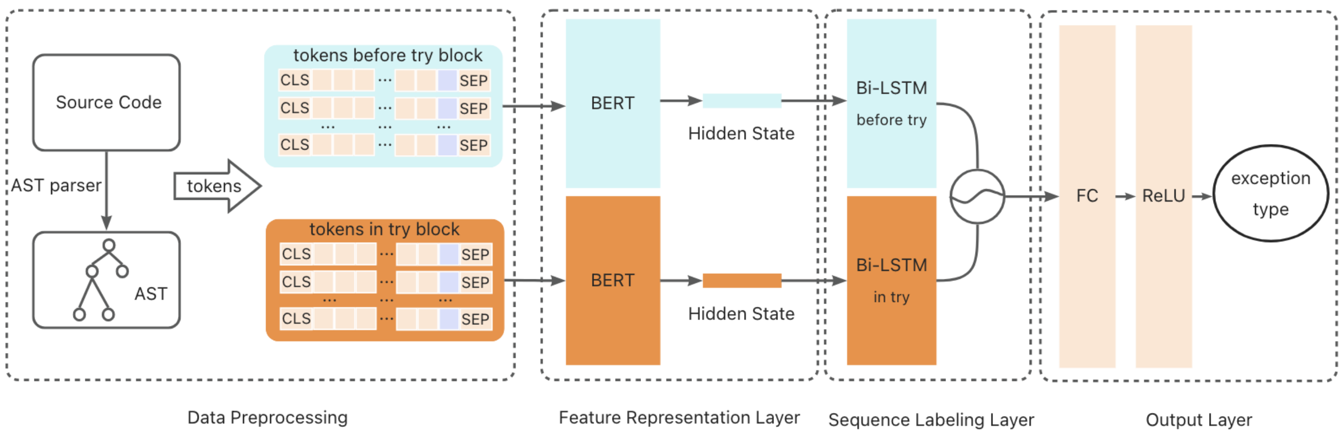


Figure 4: The structure of the model for exception type prediction.

function. The main effect of the fully connected layer is to map the feature space calculated from the previous layers to the sample labeling space. Typically, the output of the fully connected layer is processed through the Softmax function to map the output into probability. In Eq.(7), $\mathbf{z}$ represents the output vector from the fully connected layer, and $N$ is the number of output categories. For each output $i$, the softmax function computes the ratio of its exponentiated value to the sum of exponentiated values for all outputs. As shown in Eq.(8), the output vector $\mathbf{z}$ from the fully connected layer is denoted as $\mathbf{W} \cdot \mathbf{h} + \mathbf{b}$, where $\mathbf{h}$ is the output vector from the preceding layer. Subsequently, the Softmax function is applied to $\mathbf{z}$, such that $\text{Output}_i$ represents the probability of the output of the $i$-th category. The model selects the category with the highest probability in the output as the final exception type.

$$\textbf{Softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}} \tag{7}$$

$$\textbf{Output}_i = \frac{e^{(\mathbf{W}\cdot\mathbf{h}+\mathbf{b})_i}}{\sum_{j=1}^{N} e^{(\mathbf{W}\cdot\mathbf{h}+\mathbf{b})_j}} \tag{8}$$

## 4 Experiments

In this section, we conduct experiments to evaluate the performance of our models on exception localization and type prediction. Specifically, we answer the following three research questions.

- RQ1: How does the performance of the proposed model for exception localization?
- RQ2: How does the performance of the proposed model for exception type prediction?
- RQ3: What is the effect of applying the semantic enhancement techniques of AST and variable name substitution in the data preprocessing?

### 4.1 Datasets

**Nexgen Dataset** This dataset comes from **Nexgen** (Zhang et al. 2020), which includes $377,923$ methods in $2,000$ of the most popular Java projects from GitHub.

**D-REX Dataset** This dataset comes from **D-REX** (Farmahinifarahani et al. 2021), which consists of $442,446$ methods in $200,000$ Java projects from GitHub. Note that this dataset is designed for exception type prediction because it contains code tokens instead of source code, which cannot be applied to the exception localization experiments.

**Neurex Dataset** This dataset comes from **Neurex** (Cai et al. 2024), which includes $307,646$ methods in $5,726$ Java projects with the highest ratings from GitHub.

### 4.2 Baselines

**D-REX** This model is proposed in (Farmahinifarahani et al. 2021). It is based on Transformer architecture for representation learning and has achieved good results in the task of exception type prediction.

**Nexgen** This model is presented in (Zhang et al. 2020). It combines the attention mechanism and Bi-LSTM and has performed well in the field of Java exception handling.

**M-E-F** This model is proposed in (Nguyen, Vu, and Nguyen 2020). It uses a statistical approach to gather all method invocation information and uses a fuzzy union approach to recommends potential exceptions based on the predicted probabilities.

**Neurex** This model is presented in (Cai et al. 2024). It uses CodeBert for word embedding and conducts joint training on whether exceptions occur, the location of exception statements and the recommendation of exception types.

**CodeHunter(~~AST~~)** This model is a simplified version of the CodeHunter model without using AST to extract additional code features. For ablation testing, this model is designed to verify the effect of AST parsing.

### 4.3 Evaluation Metrics

Since the datasets contains different types of data, we use different evaluation metrics for two tasks on three datasets.

| Model | Accuracy | Precision | Recall | F1 |
|---|---|---|---|---|
| Nexgen | 74.7% | 80.9% | 74.3% | 77.5% |
| V+Nexgen | 75.3% | 81.2% | 74.8% | 77.9% |
| A+V+Nexgen | 76.6% | 79.1% | **76.3%** | 77.7% |
| CodeHunter(~~AST~~) | 76.9% | 82.4% | 75.9% | 79.0% |
| CodeHunter | **79.7%** | **86.2%** | 75.0% | **80.2%** |

Table 1: The results of **exception localization** experiments on the **Nexgen** dataset. Here, V stands for the substitution of variable names, and A means using AST to generate additional code tokens.

**Exception Localization** In the experiments of exception localization, common metrics of sequence labeling problems are used, including accuracy, precision, recall, and F1 score. For the Neurex dataset, the accuracy metric is row accuracy, which is the number of correctly predicted code lines divided by the total number of code lines. As to the Nexgen dataset, the accuracy metric is method accuracy, which is the number of correctly predicted methods divided by the total number of methods. A method will be considered correct only when each line in the method is correctly predicted.

**Exception Type Prediction** To evaluate the effectiveness of exception type prediction, we use the *Top-K* accuracy metric for the Nexgen dataset and the D-REX dataset, as shown in Eq.(9). Here, *Top-K Accuracy* denotes the accuracy of whether the true label is contained in the *Top-K* predictions, *Top-K Hits* represents the number of correct predictions by the model in the top $K$ predictions, and *Total Samples* is the total number of samples. For the values of $K$, we select 1, 2, 3, 5, and 10 respectively. For the Neurex dataset, we use the same metrics as in exception localization, namely accuracy, precision, recall, and F1 score.

$$\textit{Top-K Accuracy} = \frac{\textit{Top-K Hits}}{\textit{Total Samples}} \tag{9}$$

## 4.4 Experimental Results

**The Experimental Results of Exception Localization** The experiments of exception localization are conducted only on the Nexgen dataset and the Neurex dataset because the D-REX dataset contains only code tokens instead of the original source code. The experimental results on the Nexgen dataset are presented in Table 1. Among the models in Table 1, the Nexgen model serves as the basis for this comparison and achieves an accuracy of 74.7%, since it combines the attention mechanism and the Bi-LSTM model. If the basic Nexgen model is combined with AST parsing and variable name substitution techniques to enhance the feature representation, it can achieve an accuracy of 76.6%, which suggests that two optimization techniques can improve the model's performance to a certain extent.

The CodeHunter model shows excellent performance with an accuracy of 79.7%. This model integrates abstract syntax tree representation, BERT-based feature representation, and Bi-LSTM sequence labeling, taking advantage of

| Model | Accuracy | Precision | Recall | F1 |
|---|---|---|---|---|
| Neurex | 87.4% | **100.0%** | 62.0% | 76.5% |
| CodeHunter | **98.6%** | 93.6% | **92.4%** | **93.0%** |

Table 2: The results of **exception localization** experiments on the **Neurex** dataset.

| Top-K | M-E-F | D-REX | CodeHunter |
|---|---|---|---|
| Top-1 Acc | 70.95% | 81.09% | **87.26%** |
| Top-2 Acc | 83.06% | 88.91% | **92.71%** |
| Top-3 Acc | 87.83% | 91.70% | **94.86%** |
| Top-5 Acc | 93.89% | 94.52% | **96.70%** |
| Top-10 Acc | 97.36% | 97.08% | **98.23%** |

Table 3: The results of **exception type prediction** experiments on the **D-REX** dataset. Note that the results of D-REX model are copied from (Farmahinifarahani et al. 2021), because we cannot get the source code of D-REX.

BERT-based model's powerful semantic representation ability to effectively enhance the accuracy of identifying exception lines. Comparatively, the CodeHunter without AST, which removes AST parsing from CodeHunter, achieves an accuracy of 76.9%, which is lower than the complete CodeHunter model, indicating the effectiveness of AST.

The experimental results on the Neurex dataset are shown in Table 2. From Table 2, we can find that the accuracy of CodeHunter is 98.6%, which is much higher than Neurex's 87.4% accuracy. In terms of recall and F1 score, CodeHunter is also much higher than Neurex. This is because although the Neurex model uses CodeBert for word embedding, our model not only uses CodeT5+ for word embedding, but also uses Bi-LSTM for sequence labeling to further extract code context information, and uses AST parsing to extract more information from the code data. As a result, it performs better. CodeHunter is only lower than Neurex in precision, which may be related to the training set. Due to the fact that the authors of Neurex (Cai et al. 2024) did not provide the processed training set, we constructed an equal number of methods from the original dataset as the training set ourselves. Therefore, there is a certain gap between the two training sets. Note that Neurex has a high precision but a low recall, indicating that there are few false positives but many false negatives in the prediction results. However, in CodeHunter, these two are relatively even, and the difference between training datasets may lead to such results.

In summary, the CodeHunter model performs best, achieving an accuracy of 79.7% on the Nexgen dataset and an accuracy of 98.6% on the Neurex dataset. It also outperforms other baselines on the metrics of precision, recall, and F1 score. By integrating AST, BERT-based feature representation and Bi-LSTM, the CodeHunter model is able to capture the features of exception code better, leading to superior localization results.

**The Experimental Results of Exception Type Prediction** We conduct comparative experiments on three dif-

| Top-K | M-E-F | CodeHunter(~~AST~~) | CodeHunter |
|---|---|---|---|
| Top-1 Acc | 64.91% | 89.21% | **93.86%** |
| Top-2 Acc | 85.13% | 97.25% | **98.14%** |
| Top-3 Acc | 93.66% | 98.39% | **98.79%** |
| Top-5 Acc | 98.26% | 99.20% | **99.42%** |

Table 4: The results of **exception type prediction** experiments on the **Nexgen** dataset. Since the number of all exception types in this dataset is 10, the values of the Top-10 accuracy are all 100%. Thus, we ignore them here.

| Model | Accuracy | Precision | Recall | F1 |
|---|---|---|---|---|
| Neurex | 73.3% | **97.4%** | 56.9% | 71.8% |
| CodeHunter | **92.0%** | 94.9% | **92.0%** | **92.9%** |

Table 5: The results of **exception type prediction** experiments on the **Neurex** dataset.

ferent datasets. On the D-REX dataset, three models are compared in four Top-K accuracy metrics. The experimental results are presented in Table 3. The CodeHunter model performs best by achieving a Top-1 accuracy of 87.26%, which is significantly higher than the 70.95% of M-E-F and the 81.09% of D-REX. Furthermore, the CodeHunter model also outperforms M-E-F and D-REX in Top-2, Top-3, and Top-5 accuracy metrics.

On the Nexgen dataset, we compare CodeHunter with M-E-F. Additionally, a set of ablation experiments is designed to simulate the effect of CodeHunter without AST. The number of exception types is set to 10. The experimental results are presented in Table 4. CodeHunter performs the best by achieving a Top-1 accuracy of 93.86%, which is significantly higher than that of M-E-F and CodeHunter(~~AST~~). This means AST can create better code features for model performance. Furthermore, the CodeHunter model also outperforms M-E-F and CodeHunter(~~AST~~) in Top-2, Top-3, and Top-5 accuracy metrics, respectively.

On the Neurex dataset, we compare the accuracy, precision, recall, and F1 score of CodeHunter and Neurex. The experimental results are shown in Table 5. CodeHunter achieves an accuracy of 92.0%, which is much higher than Neurex's 73.3%. In terms of recall and F1 score, CodeHunter is also much higher than Neurex. Note the precision of CodeHunter is lower than that of Neurex. It is likely to be related to differences in the training set, which is the same reason in the experiment of exception localization.

Overall, the CodeHunter model achieves optimal results on all three datasets, outperforming other baseline models in accuracy, precision, recall, and F1 score.

### 4.5 Analysis of the Experimental Results

Regarding the three research questions in Section 4, we answer them as follows.

**RQ1: How does the performance of the proposed model for exception localization?** From Table 1 and Table 2, we verify that CodeHunter achieves significant results in exception localization, reaching an accuracy of 79.7% and 98.6% respectively on two datasets. The experimental results indicate that by using BERT-based model's powerful semantic representation ability and the sequence labeling ability of Bi-LSTM, the model can capture the features of exceptional code to provide valuable references for exception handling.

**RQ2: How does the performance of the proposed model for exception type prediction?** From Table 3, Table 4 and Table 5, we verify that CodeHunter achieves significant results in exception type prediction, reaching a Top-1 accuracy of 87.26% on the D-REX dataset, a Top-1 accuracy of 93.86% on the Nexgen dataset and an accuracy of 92.0% on the Neurex dataset. The results indicate that by employing a structure like the Deep Structured Semantic Model (Huang et al. 2013), where the code before the try block and in the try block are fed into distinct BERT-based models separately, the model can better capture the features of exceptional code. By using BERT-based model's powerful semantic representation ability for feature extraction, then expressing the importance of code in the try block through the attention mechanism, and finally combining Bi-LSTM's sequence labeling ability, CodeHunter is able to identify exception types more accurately compared to the baselines.

**RQ3: What is the effect of applying the semantic enhancement techniques of AST and variable name substitution in the data preprocessing?** In Table 1, CodeHunter, which combines AST representation with BERT-based features, outperforms other models. Simultaneously, the variable name substitution technique also enhances the model's accuracy. Particularly for the modified Nexgen model attached with the AST and variable name substitution techniques, the accuracy reaches 76.6%. This indicates that in the task of exception localization, considering both semantic information in the code and the impact of variable names is crucial. From Table 4, in the exception type prediction experiment on the Nexgen dataset, the Top-1 accuracy without using AST is 89.21%. However, the Top-1 accuracy with AST reaches 93.86%, which is significantly better than that without AST parsing. This also indicates that applying the technique of abstract syntax trees is very important in the task of exception type prediction.

## 5 Conclusion

Exception handling is complex and important in program development. In this paper, we propose CodeHunter, a model based on code learning for exception localization and exception type prediction. CodeHunter uses BERT-based model to represent code information with the optimization techniques AST and variable name substitution. The model also utilizes Bi-LSTM to recognize the exceptional points and predict exception types based on the exceptional code and its context semantics. Finally, the accuracy and effectiveness of our model are verified through experiments on three public datasets. The ablation testing is also executed to prove the techniques used are critical. Our model is further implemented as an IDE plugin for programmers.

## Acknowledgments

This work is supported by NSFC (No. 62272416, 62372176) and the National Key Research Project of China(No. 2023YFA1009402).

## References

Abdelaziz, I.; Dolby, J.; Khurana, U.; Samulowitz, H.; and Srinivas, K. 2023. SemFORMS: Automatic Generation of Semantic Transforms By Mining Data Science Code. In *Proceedings of the 32nd International Joint Conference on Artificial Intelligence, IJCAI 2023*, 7106–7109.

Allamanis, M.; Barr, E. T.; Devanbu, P. T.; and Sutton, C. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.*, 51(4): 81:1–81:37.

Barbosa, E. A.; and Garcia, A. 2018. Global-Aware Recommendations for Repairing Violations in Exception Handling. *IEEE Trans. Software Eng.*, 44(9): 855–873.

Cai, Y.; Yadavally, A.; Mishra, A.; Montejo, G.; and Nguyen, T. N. 2024. Programming Assistant for Exception Handling with CodeBERT. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024*, 94:1–94:13.

Devlin, J.; Chang, M.; Lee, K.; and Toutanova, K. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR*, abs/1810.04805.

Farmahinifarahani, F.; Lu, Y.; Saini, V.; Baldi, P.; and Lopes, C. V. 2021. D-REX: Static Detection of Relevant Runtime Exceptions with Location Aware Transformer. In *Proceedings of the 21st IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021*, 198–208.

Henkel, J.; Lahiri, S. K.; Liblit, B.; and Reps, T. W. 2018. Code Vectors: Understanding Programs through Embedded Abstracted Symbolic Traces. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018*, 163–174.

Huang, P.; He, X.; Gao, J.; Deng, L.; Acero, A.; and Heck, L. P. 2013. Learning Deep Structured Semantic Models for Web Search Using Clickthrough Data. In *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management, CIKM 2013*, 2333–2338.

Lee, J.; Hong, S.; and Oh, H. 2022. NPEX: Repairing Java Null Pointer Exceptions without Tests. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering, ICSE 2022*, 1532–1544.

Li, R.; Chen, B.; Zhang, F.; Sun, C.; and Peng, X. 2022. Detecting Runtime Exceptions by Deep Code Representation Learning with Attention-Based Graph Neural Networks. In *Proceedings of IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022*, 373–384.

Liu, K.; Li, L.; Koyuncu, A.; Kim, D.; Liu, Z.; Klein, J.; and Bissyandé, T. F. 2021. A Critical Review on the Evaluation of Automated Program Repair Systems. *J. Syst. Softw.*, 171: 110817.

Nguyen, T.; Vu, P.; and Nguyen, T. 2019. Recommending Exception Handling Code. In *Proceedings of 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*, 390–393.

Nguyen, T.; Vu, P.; and Nguyen, T. 2020. Code recommendation for exception handling. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020*, 1027–1038.

Niu, C.; Li, C.; Luo, B.; and Ng, V. 2022. Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code. In *Proceedings of the 31st International Joint Conference on Artificial Intelligence, IJCAI 2022*, 5546–5555.

Thummalapenta, S.; and Xie, T. 2009. Mining Exception-handling Rules as Sequence Association Rules. In *Proceedings of the 31st International Conference on Software Engineering, ICSE 2009*, 496–506.

Tufano, M.; Watson, C.; Bavota, G.; Penta, M. D.; White, M.; and Poshyvanyk, D. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4): 19:1–19:29.

Wang, S.; Huang, L.; Gao, A.; Ge, J.; Zhang, T.; Feng, H.; Satyarth, I.; Li, M.; Zhang, H.; and Ng, V. 2023a. Machine/Deep Learning for Software Engineering: A Systematic Literature Review. *IEEE Trans. Software Eng.*, 49(3): 1188–1231.

Wang, Y.; Le, H.; Gotmare, A.; Bui, N. D. Q.; Li, J.; and Hoi, S. C. H. 2023b. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023*, 1069–1088.

Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Pu, Y.; and Liu, X. 2020. Learning to Handle Exceptions. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*, 29–41.

Zhou, P.; Shi, W.; Tian, J.; Qi, Z.; Li, B.; Hao, H.; and Xu, B. 2016. Attention-Based Bidirectional Long Short-Term Memory Networks for Relation Classification. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016*.