Learning to Handle Exceptions

Jian Zhang*
SKLSDE Lab, Beihang University,
China
zhangj@act.buaa.edu.cn

Xu Wang*†
SKLSDE Lab, Beihang University,
China
wangxu@act.buaa.edu.cn

Hongyu Zhang
The University of Newcastle,
Australia
hongyu.zhang@newcastle.edu.au

Hailong Sun*
SKLSDE Lab, Beihang University,
China
sunhl@act.buaa.edu.cn

Yanjun Pu
SKLSDE Lab, Beihang University,
China
puyanjun@nlsde.buaa.edu.cn

Xudong Liu*
SKLSDE Lab, Beihang University,
China
liuxd@act.buaa.edu.cn

ABSTRACT

Exception handling is an important built-in feature of many modern programming languages such as Java. It allows developers to deal with abnormal or unexpected conditions that may occur at runtime in advance by using try-catch blocks. Missing or improper implementation of exception handling can cause catastrophic consequences such as system crash. However, previous studies reveal that developers are unwilling or feel it hard to adopt exception handling mechanism, and tend to ignore it until a system failure forces them to do so. To help developers with exception handling, existing work produces recommendations such as code examples and exception types, which still requires developers to localize the try blocks and modify the catch block code to fit the context. In this paper, we propose a novel neural approach to automated exception handling, which can predict locations of try blocks and automatically generate the complete catch blocks. We collect a large number of Java methods from GitHub and conduct experiments to evaluate our approach. The evaluation results, including quantitative measurement and human evaluation, show that our approach is highly effective and outperforms all baselines. Our work makes one step further towards automated exception handling.

CCS CONCEPTS

• **Software and its engineering** → **Error handling and recovery**; **Automatic programming**.

KEYWORDS

Exception handling, deep learning,neural network, code generation

ACM Reference Format:
Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Yanjun Pu, and Xudong Liu. 2020. Learning to Handle Exceptions. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416568

1 INTRODUCTION

Exception handling mechanism is essential for modern programming languages such as Java and C# to build robust and reliable software systems [43, 58]. It provides an effective means of dealing with exceptional conditions and recovering from them by using try-catch blocks [22, 25]. Exception handling separates error-handling code from regular source code and improves program comprehension and maintenance [15]. However, missing or improperly using exception handling statements may cause severe problems such as system crash [67] and information leakage [68]. One recent study [19] reveals that there is a significant relationship between exception flow characteristics and post-release defects in Java projects. Therefore, it is crucial for developers to handle exceptions.

Despite the importance of exception handling, the exception handling statements such as try-catch blocks in real-world software are often poorly written and error-prone. For example, prior studies [10, 18, 21] indicate that many industrial systems exhibit poor quality code with respect to exception handling. Besides, similar bad practices have been found in open source software [4, 46]. Among our collected millions of original Java methods in 2,000 projects from GitHub with high numbers of stars and forks, only 14.9% of the methods apply try blocks to capture exceptions, and 31.2% of these methods do not have catch blocks and do nothing when exceptions occur. The reason is twofold. On the one hand, developers tend to ignore exception handling code or even have little knowledge about whether or not exception handling is needed until an error occurs [56, 57]. On the other hand, the exception handling code is often difficult to write for developers, especially when it comes to program evolution [53]. Hence, as suggested in [19], it is essential to propose automatic techniques for assisting developers in writing high-quality exception handling code.

Existing work on automatic techniques for exception handling mainly includes violation detection of exception handling policies [6, 7, 58] and exception handling code recommendation [45, 51]. For example, Thummalapenta et al. [58] mined association rules on the sequence of function calls in try and catch blocks and applied them

[*] Also with Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, Beijing 100191, China.
[†] Corresponding author: Xu Wang, wangxu@act.buaa.edu.cn.

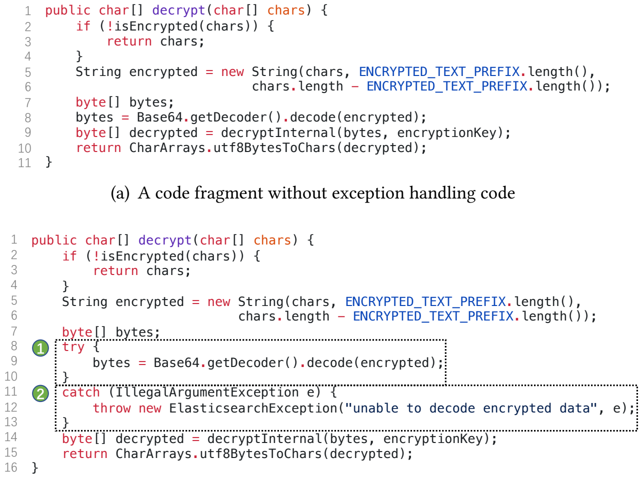
ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416568

to detect violations of exception handling policies. Such approaches are helpful to improve the quality of exception handling code, yet they are limited since they assume that the try-catch blocks have been completely written by developers. By contrast, exception handling code recommendation is more useful to assist developers in writing the try-catch blocks. Given an under-development code fragment without try-catch blocks, Rahman et al. [51] retrieved similar code fragments that include try-catch blocks to recommend exception handling code examples from popular code repositories at GitHub. Nguyen et al. [45] recommended exception types and method calls in the catch block by utilizing the fuzzy set theory [35] and N-gram model [27]. However, there are two main drawbacks of recommendation-based approaches. First, these studies assume that developers understand when to seek for help about writing exception handling code and where the try-catch blocks should be put, while in practice it is not always the case [56, 57]; Second, even though the recommended code such as examples, exception types and method calls can provide useful knowledge to aid developers in writing try-catch blocks, the developers still need to modify the code to fit the context.

In this paper, we propose a novel **Ne**ural approach to automated **ex**ception handling code **gen**eration (namely *Nexgen*), which can predict locations of try blocks and automatically generate the complete catch blocks. We separate exception handling into two tasks: one is to predict the try block locations for identifying source code that needs to handle potential exceptions; the other is to generate complete catch blocks to deal with the exceptions. Consider a typical coding scenario that a developer has written a code fragment as shown in 1(a), which is a Java method from the Elasticsearch[1] project. Here we remove its original exception handling code for illustration purpose. The first task is to identify which statements need to be put in a try block. In this example, the statement at line 8 in Figure 1(a) requires a try block. The second task is to generate the catch block. In this example, according to the code above line 11 in Figure 1(b), the catch block in lines 11-13 is generated to handle the exception. In this way, the potential exceptions can be automatically captured and handled.

Our approach utilizes deep learning to learn regularities/patterns from a large amount of historical exception handling code. More specifically, we tackle the two tasks as follows:

- **Try block localization**. We design a locator that jointly learns to determine whether or not a given code fragment needs to handle exceptions and localize where to put the try block if it does. Considering the sequential naturalness [27] of source code, we transform the localization problem into the sequence tagging problem [29]. To avoid the long-term dependency problem [9], we regard the code fragment as a sequence of statements and hierarchically encode them into vectors so as to predict the label sequence. We first use long short-term memory (LSTM) [28] to encode statements into statement vectors and then use another LSTM to capture the sequential dependency of the statement sequence. Besides, we apply hierarchical attention mechanism [65] on top of the two LSTMs to let the model pay attention to individual tokens and statements.



```
 1 public char[] decrypt(char[] chars) {
 2     if (!isEncrypted(chars)) {
 3         return chars;
 4     }
 5     String encrypted = new String(chars, ENCRYPTED_TEXT_PREFIX.length(),
 6                                   chars.length - ENCRYPTED_TEXT_PREFIX.length());
 7     byte[] bytes;
 8     bytes = Base64.getDecoder().decode(encrypted);
 9     byte[] decrypted = decryptInternal(bytes, encryptionKey);
10     return CharArrays.utf8BytesToChars(decrypted);
11 }
```

(a) A code fragment without exception handling code

```
 1 public char[] decrypt(char[] chars) {
 2     if (!isEncrypted(chars)) {
 3         return chars;
 4     }
 5     String encrypted = new String(chars, ENCRYPTED_TEXT_PREFIX.length(),
 6                                   chars.length - ENCRYPTED_TEXT_PREFIX.length());
 7     byte[] bytes;
 8     try {
 9         bytes = Base64.getDecoder().decode(encrypted);
10     }
11     catch (IllegalArgumentException e) {
12         throw new ElasticsearchException("unable to decode encrypted data", e);
13     }
14     byte[] decrypted = decryptInternal(bytes, encryptionKey);
15     return CharArrays.utf8BytesToChars(decrypted);
16 }
```

(b) The code fragment after adding exception handling code

Figure 1: An example of exception handling

- **Catch block generation**. Different from previous work such as [45], we take all the code before the catch blocks as the context instead of simply relying on the try blocks. Thus we consider the dependencies (e.g., how the variable "encrypted" is initialized) for writing the exception handling code. A simple way is to treat all the tokens before line 11 (including line 5) as a single sequence, then apply the encoder-decoder architecture [5] to generate the tokens of the catch block in lines 11-13 one by one. However, such a model will not distinguish whether the tokens are inside or outside the try block. Therefore, we propose to encode them separately by two encoders, and fuse the context vectors using the learned weights. Furthermore, it is obvious that not all the tokens before the try blocks are helpful and may be noisy data, especially for the long methods with complex logic. In order to concentrate more on the dependencies without destroying the naturalness (i.e., deleting those noisy code), we incorporate program slicing technique [61] by taking the statements in the try blocks as the slicing criterion, and backtracking dependencies between statements. We add an additional attention module with masks to attend only to the tokens in these slices. Finally, we fuse the slicing-based context vector with the context mentioned above, and generate code by the LSTM decoder.

We collect a large number of Java methods from popular open source repositories in GitHub to construct datasets for the two tasks. We conduct extensive experiments on the datasets to evaluate our approach and the results demonstrate the effectiveness of our models on both tasks. More specifically, for try block localization, we achieve an accuracy of 74.7% in terms of correctly predicted methods and an F1-score of 77.4% in terms of correctly predicted statements. For the generation of catch block, our model can generate the catch blocks with 22.6% exact matches and a BLEU value of 46.7%. Both models significantly outperform the baselines.

[1] https://github.com/elastic/elasticsearch

Moreover, we perform a human evaluation for the generated code and the results confirm the superiority of our model.

In summary, this paper makes the following contributions:

- We propose a novel approach to automated exception handling, including two neural-network-based models to localize potential exceptions in the source code and generate code to handle the exceptions.
- We provide two datasets with over 700K Java methods for experiments on automated exception handling. We have publicly released the datasets to promote further research on this interesting topic.
- We conduct extensive experiments to evaluate our approach using the collected datasets. We also perform a human evaluation. The results show that the proposed approach is effective and outperforms all baselines.

The remainder of this paper is organized as follows. Section 2 presents the problem formulation of the automated exception handling. Section 3 describes the details of our approach. Section 4 describes datasets, evaluation procedure, and evaluation results. We discuss our approach in Section 5. We describe threats to the validity of this work and related work in Section 6 and Section 7, respectively. Finally, we conclude our paper in Section 8.

2 PROBLEM FORMULATION

As illustrated in Figure 1, the problem of automated exception handling can be decomposed into two successive tasks. The first task is determining which statements may throw exceptions and should be enclosed by a try block. The second task is generating the corresponding catch blocks for handling such exceptions. For simplification, we call the two tasks *try block localization* and *catch block generation*, respectively. We formally define them as follows.

Try Block Localization. This task aims to localize the statements in a code fragment that should be enclosed by try blocks. Given the code fragment $C = \{s_1, s_2, \dots, s_K\}$ where K is the number of statements, the target is to find one sequence $\mathbb{Y} = \{y_1, y_2, \dots, y_K\}$, where y_i = Y or N means whether the statement s_i is in one try block or not. In addition, the adjacent statements with label Y should be put in one same try block. If all statements are labelled by N, it means that the code fragment does not need try blocks. Note that $\mathbb{Y}$ may include multiple disjoint subsequences of Ys for multiple non-nested try blocks, and we do not consider the nested try-catch blocks in this work.

Catch Block Generation. For a try block, this task is designed to generate code tokens to compose the corresponding one or more catch blocks. Suppose a token sequence $C = \{c_1, c_2, \dots, c_l\}$ where c_i $(i \in [1, l])$ means one token in the try block and the source code before it, the target is to generate the token sequence of catch blocks $\mathbb{Y} = \{y_1, y_2, \dots, y_m\}$ so that $\mathbb{Y}$ can handle the exceptions of the try block. Here $\mathbb{Y}$ may include multiple catch blocks for different types of exceptions thrown by the try block.

We can automatically handle exceptions of code fragments based on the try block localization and catch block generation. We achieve this by learning regularities/patterns from a large amount of historical exception handling code, which can be obtained by mining open source repositories such as GitHub. In our work, we utilize deep learning as it can better capture contextual information and understand the semantics of source code. We design two deep neural network based models for the two tasks, and introduce our approach in detail in the next section.

3 APPROACH

In this section, we introduce our neural network based approach to automated exception handling, including the model for try block localization and the model for catch block generation. Note that if developers know where the exceptions are thrown and have written try blocks, the catch block generation can work independently to produce the source code of corresponding catch blocks.

3.1 Try Block Localization

3.1.1 Overview. As defined above, given a code fragment without try-catch blocks, the try block localization task is to find statement subsequences of the code fragment that may throw exceptions and should be enclosed by try blocks. Since all the input tokens of the code fragment will be checked if they are inside or outside a try block, we transform it into a sequence tagging problem [29], where each token of the code fragment will be tagged with label Y or N to represent whether it should be in a try block or not. If the code fragment does not need try blocks, the labels are all Ns. This kind of problem has been well studied in the NLP community such as Part-of-Speech (POS) tagging [50, 52], named entity recognition (NER) [37, 71], and semantic role labeling [59, 64]. Among these studies, recurrent neural network (RNN) [54] based approaches show many advantages over the traditional ones. But they were designed to parse natural language sentences which are usually shorter than code fragments. As a result, they suffer from the long-term dependency problem [9] when applied to tag source code. Since the try blocks are always comprised of one or multiple statements, we propose a two-layered neural model to tag the source code, which encodes tokens of a statement into the statement vector and tags a code fragment at the statement level.

Figure 2 shows the overall architecture of our try block locator. First, we split the code fragment into a sequence of statements. For each statement, we obtain its token sequence and use the LSTM [28] to encode the token sequence for the semantic vector of the statement. Besides, we apply the attention mechanism [65] on the token sequence of the statement to consider different weights of individual tokens. After encoding all the statements of the code fragment to a sequence of statement semantic vectors, we utilize another LSTM to further encode the statement sequence, as well as an attention mechanism for capturing the importance of different statements. Finally, we adopt a binary classifier over the normalized vectors to predict the labels of the corresponding statements.

3.1.2 Try Block Locator. Specifically, given a code fragment without try-catch blocks, we obtain the sequence of statements S in a line-by-line manner excluding blank and comment lines. Let $S = \{s_1, s_2, \dots, s_K\}$, where K is the number of statements; $s_i = \{c_{i_1}, c_{i_2}, \dots, c_{i_L}\}$, where L is the length of one statement s_i in terms of tokens. Given a statement s_i, we first embed each token into a vector via an embedding matrix W_e, that is, $x_{i_t} = W_e c_{i_t}$. We use the LSTM network to encode s_i based on the token embeddings, which reads the token embeddings from x_{i_1} to x_{i_L} to obtain the hidden

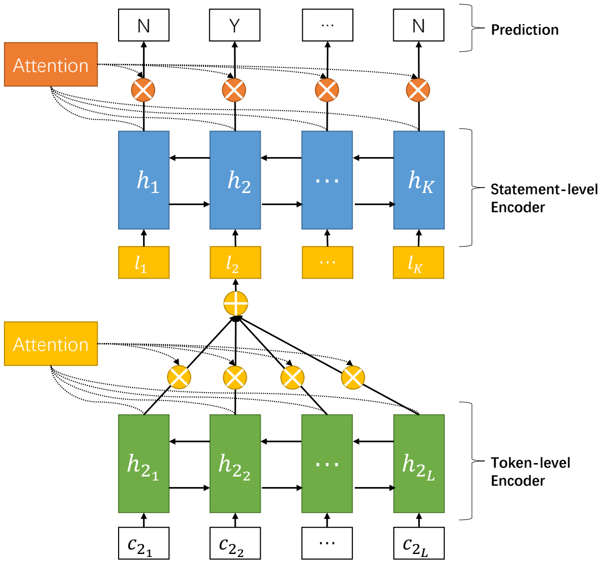


Figure 2: The architecture of our try block locator

states. At time step t, the hidden state $\overrightarrow{h_{i_t}}$ is obtained by:

$$\overrightarrow{h_{i_t}} = \overrightarrow{LSTM}(h_{i_{t-1}}, x_{i_t}), t \in [1, L]. \tag{1}$$

Furthermore, we adopt the Bidirectional LSTM(Bi-LSTM) [54] to enhance the capability of capturing the context information within the statement, where the hidden states of both directions are concatenated to form the new states:

$$\begin{aligned} \overleftarrow{h_{i_t}} &= \overleftarrow{LSTM}(h_{i_{t+1}}, x_{i_t}), t \in [L, 1]. \\ h_{i_t} &= [\overrightarrow{h_{i_t}}, \overleftarrow{h_{i_t}}], t \in [1, L]. \end{aligned} \tag{2}$$

Intuitively, not all the tokens in the statement are important for throwing an exception. To incorporate such knowledge in our model, we exploit the attention mechanism [65] over the hidden states to assign important tokens with higher weights and then aggregate them to form the statement vector. The process can be expressed as the following equations:

$$\begin{aligned} u_{i_t} &= tanh(W_\omega h_{i_t} + b_\omega), \\ \alpha_{i_t} &= \frac{exp(u_{i_t}^\top u_\omega)}{\sum_{t=1}^{L} exp(u_{i_t}^\top u_\omega)}, \\ s_i &= \sum_{t=1}^{L} \alpha_{i_t} h_{i_t}. \end{aligned} \tag{3}$$

As listed in Equation 3, we first transform the hidden states into a new high-dimension space with a one-layer MLP to prepare for scoring the importance. Then we compute the scores of these hidden states by measuring how they match the fixed vector u_ω and get the normalized weights α_{i_t} through the softmax function. Here u_ω acts as the query of which token is more likely to throw an exception. Finally, a vector representation s_i of the statement is obtained by computing the weighted sum of the hidden states.

After obtaining a sequence of statement vectors, we model the sequential dependencies of the statements by feeding them into another Bi-LSTM. Similarly, we get the hidden states of the statements $h_i = BiLSTM(s_i)$. Next, we apply the attention mechanism by considering different weights of statements so that:

$$\begin{aligned} u_i &= tanh(W_s h_i + b_s), \\ \alpha_i &= \frac{exp(u_i^\top u_s)}{\sum_{t=1}^{L} exp(u_i^\top u_s)}, \\ h_i &= \alpha_i h_i. \end{aligned} \tag{4}$$

The probability of label Y for statement s_i is calculated by

$$\hat{y}_i = sigmoid(W_p h_i + b_p) \in [0, 1], \tag{5}$$

where W_p is the weight matrix and b_p is a bias term. To train the model, we employ the binary cross-entropy loss that is defined as

$$\mathcal{L}(\Theta, \hat{y}, y) = \sum_{n=1}^{N} \sum_{i=1}^{L} (-(y_{ni} \cdot log(\hat{y}_{ni}) + (1 - y_{ni}) \cdot log(1 - \hat{y}_{ni}))), \tag{6}$$

where y is the ground truth label for each statement, Θ represents the parameters to be learned and N is the total number of instances in the training set. During prediction, we make the label as Y if $\hat{y}_i \geq \delta$ otherwise N, where δ is the threshold.

Based on the predictions, all consecutive statements with Y labels will be enclosed by one try block. In this way, we obtain the locations of try blocks. Once one try block is determined in the code fragment, we need to generate the corresponding catch blocks as well, which will be described in the next subsection.

3.2 Catch Block Generation

3.2.1 Overview. For handling exceptions of one try block, inspired by the success of Neural Machine Translation (NMT), we adopt the popular encoder-decoder architecture [5] to encode the source code before catch blocks and output a sequence of tokens in the catch blocks. An important question here is what context should be taken as the input for generating the catch blocks. One straightforward solution is to only consider the source code in try blocks, as recent studies [45, 51] did, because the source code before try blocks contains much irrelevant and noisy data. In this paper, to facilitate explanation, we call the source code before try blocks as the *leading code*. For example, lines 1-7 in Figure 1(b) is the leading code. We find that the code in a catch block may depend on some statements (such as variable initialization) in the leading code. Therefore, as depicted in Figure 3, our catch block generator includes two encoders to separately encode the leading code (i.e. the blue ones) and the try block (i.e. the red ones), and a slicing-based attention module (i.e. the curves) to filter out the noisy data in the leading code.

3.2.2 Catch Block Generator. We describe the three main components of the catch block generator in detail below.

1) First, we only use the source code in the try blocks as the input. Let $W = \{w_1, w_2, \ldots, w_n\}$ denote the token sequence of a try block, we first encode the tokens by a Bi-LSTM with the embedding layer to obtain the hidden states:

$$h_t = BiLSTM(w_t, h_{t-1}). \tag{7}$$

Then we use an LSTM as the decoder to decode the vector representation of one try block and generate the tokens of its catch blocks

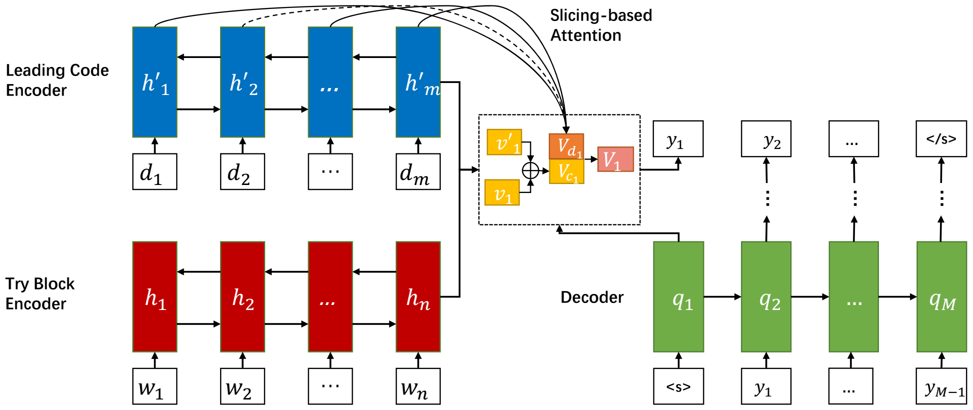


Figure 3: The architecture of our catch block generator.

one by one. Specifically, when generating the i-th token at time step i, we update the hidden state q_i of the decoder by

$$q_i = LSTM(q_{i-1}, y_{i-1}), \tag{8}$$

where y_{i-1} is the previous token. To enhance the alignment ability of the decoder, we adopt the attention mechanism over the hidden states $H = \{h_1, h_2, \cdots, h_n\}$ of the encoder to compute the context vector v_i. For simplification, we represent v_i by the following equation:

$$v_i = Attention(q_i, H). \tag{9}$$

Different from u_ω in Equation 3, here q_i can be considered as a query vector to learn to align the target token with the source ones.

2) Second, as mentioned before, considering only the context of the try blocks may miss important dependency information. A simple solution is to concatenate the leading code and the try block into a single sequence and then train a standard encoder-decoder model, but such a model cannot distinguish the tokens of the leading code from those in the try blocks. While in practice, the scope of the try blocks does matter since the exceptions are produced within it. To overcome the drawback, we train an additional encoder for storing the information of the leading code. Let $D = \{d_1, d_2, \ldots, d_m\}$ represents the token sequence of the leading code, according to Equation 7, we encode it by another Bi-LSTM and get the hidden states $H' = \{h'_1, h'_2, \ldots, h'_m\}$. Then we get the context vector of the leading code by Equation 9, that is, $v'_i = Attention(q_i, H')$. We fuse the two context vectors of the leading code and the try block by weighted sum with one MLP layer to form the context vector V_{c_i}.

3) Third, there are also many noisy tokens in the leading code that may influence the performance of catch block generation. For the long methods, the leading code may include many unrelated tokens such as checking parameters or processing some data while the try blocks do not depend on them. Therefore, we incorporate the program slicing technique [61] to filter out the noisy tokens of the leading code in our model. The slicing results are used to label the tokens of leading code by 1 or 0. When applying the attention on them, we mask the noisy tokens with 0s, which means that the attention weights of them will be 0. In this way, this attention module will only attend to the dependent tokens whose labels are 1s. In short, we initially take the statements in the try block as the slicing criterion SC, and recursively backtrack statements in the leading code that may influence the statements of SC by data dependency to update SC. Then we get the token sequence D' of all statements in the leading code from SC, which has filtered noisy tokens that have no influence on the try block. We use this sequence D' to label any token d_i in D and get the labels $L = \{l_1, l_2, \ldots, l_m\}$, where:

$$l_i = \begin{cases} 1, & if\ d_i \in D', \\ 0, & otherwise. \end{cases} \tag{10}$$

We use the labels as masks and combine them with the attention mechanism to enhance our model by considering the program dependencies and ignoring the noisy data. Thus:

$$V_{d_i} = Attention(q_i, L \cdot H'). \tag{11}$$

We concatenate the slicing-based vector V_{d_i} with V_{c_i} and add another one-layer MLP to get the final context vector V_i.

Based on the context vector, we calculate the probability of generating the i-th token of the corresponding catch blocks by:

$$p(y_i | y_1, \ldots, y_{i-1}, C) = softmax(W_g V_i + b_g), \tag{12}$$

where W_g and b_g are the weight matrix and bias term, respectively, C represents the two input sequences of the leading code and the try block. Training such an attentional encoder-decoder model is to minimize the loss function:

$$\mathcal{L}(\Theta) = -\sum_{i=1}^{N} \sum_{t=1}^{M} \log P(y_t^i | y_{<t}^i, C), \tag{13}$$

where Θ is the trainable parameters, N is the total number of training instances and M is the length of each target sequence.

In practice, the trained model can predict the next token one by one with top-k candidates by the beam search algorithm [63], and finally generate the whole catch blocks.

4 EVALUATION

In this section, we first introduce our datasets collected from GitHub[2]. Then we conduct experiments to evaluate the effectiveness of our models, including automatic metrics and human evaluation.

[2] https://github.com

4.1 Data Collection

To prepare exception handling code fragments, we first crawled the repositories written in Java at GitHub and selected the top 2,000 popular ones based on their total number of stars and forks. These popular repositories usually have high-quality source code, since most of them carry out the code review process [42] to guarantee the software quality. We extracted Java methods by parsing all Java files in the repositories using ANTLR4 [49], and obtained 7,840,688 methods in total.

Obviously, not all the Java methods are useful for training our models. There are many small and simple methods that do not need exception handling such as get/set methods. Thus we filtered out the methods whose source lines of code is less than 7. We also discarded very long methods (i.e., more than 100 lines) to reduce the possibility of overfitting [14]. Then we obtained 3,269,127 methods for constructing our datasets.

For the try block localization task, we searched for the methods including at least one try block and obtained 486,435 results. We found that there were about 88,686 cases where the try block missed the corresponding catch blocks, thus we removed these cases. We also eliminated the methods that have nested try or catch blocks and finally got 377,923 of them as positive samples. In addition, we randomly selected an equal number of methods that do not have any try block as the negative samples. In this way, we built our try block localization dataset (namely TBLD) for evaluation.

Based on the positive samples described above, we built another dataset for the catch block generation task. If one positive sample has more than one non-nested try-catch pair (i.e. one try block and the corresponding catch blocks), we split it into multiple samples which include only one try-catch pair. This produced 432,679 samples in total. However, we found that some of them exposed an obvious bad practice, that is, catching an exception but doing nothing. We removed them to improve the quality of these code fragments. Finally, we obtained 351,420 samples as our catch block generation dataset (namely CBGD).

4.2 Experimental Setup

We conduct experiments on the TBLD and CBGD datasets to evaluate our try block localization model and catch block generation model, respectively. The two datasets are split into training sets, validation sets, and testing sets with fractions of 80%, 10%, and 10%, respectively. We utilize ANTLR4 to tokenize all the code fragments. The statistics of these two datasets are described in Table 1, where MaxT, AvgT and UniqT are the maximum number of tokens, the average number of tokens, and the total number of unique tokens, respectively. In the left column, MaxS and AvgS denote the maximum and average number of statements in the Java methods respectively. The Source and Target in the right column represent the source code before catch blocks and the catch blocks respectively. Note that as defined in Section 2, there can be multiple try blocks in one Java method, and each try block may have multiple catch blocks, thus we denote the number of try blocks in one method in the positive samples of TBLD by TryNum, and the number of catch blocks in one try-catch pair of CBGD by CatchNum.

The configurations of our models are as follows. For the try block localization task, we set the vocabulary size to 50k by selecting the most frequent words and replacing out-of-vocabulary words with *UNK*, because too large vocabulary may lead to worse performance. The embedding size and the dimensions of hidden states in LSTMs are 128. The batch size is set to 32 and the maximum epochs is 20. We use the best trained parameters of the 20 saved checkpoints for the later prediction according to their performance on the validation set. We adopt the widely-used Adam [33] as the optimizer with learning rate 0.001 for training our model. The threshold δ for predicting labels is 0.5 by default. For the catch block generation task, we implement our model based on OpenNMT[3]. We keep the same settings as above including the vocabulary size, embedding size and dimension of hidden states in LSTMs of the encoder and decoder. We set the length limits (in terms of #tokens) of the source and target of one try-catch pair to 400 and 100, because such settings can cover most of their original lengths. The batch size is set to 32 and the maximum iterations is 100k. When testing, we leave the beam size k as the default 5 since it yields good results.

Table 1: The statistics of the datasets used in our study

TBLD		CBGD	
#Java methods	755,846	**#Try-catch pairs**	351,420
#TryNum=1	341,040	**#CatchNum=1**	324,084
#TryNum≥2	36,883	**#CatchNum≥2**	27,336
MaxT	6403	**MaxT of Source**	3,313
AvgT	115.9	**AvgT of Source**	113.1
MaxS	99	**MaxT of Target**	365
AvgS	14.7	**AvgT of Target**	26.5
UniqT	566,378	**UniqT**	214,799

All the experiments are conducted on a Ubuntu 16.04 server with 16 cores of 2.4GHz CPU, 128GB RAM and a Tesla V100 GPU with 32GB memory.

4.3 Evaluation Metrics

We evaluate the performance of different approaches on the two tasks. For the task of try block localization, similar to the metrics in sequence labeling [29], we use Precision, Recall and F1-score to assess how well one approach predicts the locations of try blocks for code fragments in the testing set of TBLD. Their values are calculated as follows:

$$Precision = \frac{\#correct_Y}{\#predict_Y}, Recall = \frac{\#correct_Y}{\#gold_Y},$$

$$F1 - score = \frac{2 * Precision * Recall}{Precision + Recall}.$$

Here Y represents that a statement should be enclosed by one try block in a code fragment, and we denote it as $correct_Y$ if it is correctly predicted. Similarly, $predict_Y$ and $gold_Y$ are the predicted and ground truth labels of Y in the corpus respectively. Besides, we also want to know the overall ratio of methods where all statements are correctly predicted, denoted by Accuracy.

For the task of catch block generation, we adopt the automatic metric BLEU[48] in evaluating the quality of catch blocks. This metric has been widely used in many similar tasks such as machine translation [5], source code summarization [30] and API sequence generation [24]. Given the generated code Y' and the ground-truth

[3] https://github.com/OpenNMT/OpenNMT-py

Y, BLEU measures the n-gram precision between Y' and Y by computing the overlap ratios of n-grams and applying brevity penalty on short translation hypotheses. The value is computed by:

$$BLEU = BP \cdot \exp \sum_{n=1}^{N} \omega_n \log p_n,$$

where p_n is the precision score of the n-gram matches between candidate and reference sentences. The default BLEU calculates a score for up to 4-grams using uniform weights ω_n. BP is the brevity penalty and defined as:

$$BP = \begin{cases} 1, & if\ Y' > Y, \\ e^{(1-Y'/Y)}, & if\ Y' \le Y. \end{cases}$$

Likewise, we also calculate the ratio of generated catch blocks that are exactly same with the ground-truth ones, referred to Accuracy.

4.4 Baselines

4.4.1 Try Block Localization. While many previous studies use program analysis technique to detect and fix exception-related errors such as string handling [20] or API misuse detection [3], they usually relies on compilable and complete source code. We found no existing learning-based approaches to directly predicting the locations of try blocks for incomplete code fragments, thus we borrow several popular models originally proposed for sequence labeling as the baselines, including Conditional Random Field (CRF) [36], Bidirectional LSTM (BiLSTM) [34], BiLSTM-CRF [29] and Sequence to Sequence (Seq2Seq) [16] . We briefly introduce them below.

- **CRF**. We use the linear CRF model, which is a sequence modeling framework for processing text and has been used for predicting program properties [2]. It takes the raw token sequence as input and maximizes the joint probability of the entire sequence of labels given the observation sequence. We train a CRFTagger of NLTK toolkit [40] to tag the source code.
- **BiLSTM**. This model has shown its effectiveness for modeling sequential data [34]. It can capture the semantics of the words through word embedding and the sequential dependency between the words through LSTM. Besides, a BiLSTM model can access both past and future input features for a given position and thus improves the performance. We set the dimensions as same as our model for fair comparison.
- **BiLSTM-CRF**. This model is a combination of the LSTM network and the CRF module by feeding the output vectors of BiLSTM into a CRF layer. Since its superiority in learning the dependency and sentence level tag information, many studies [29, 41] use it for labeling sequences.
- **Seq2Seq**. As previously described, Seq2Seq models have witnessed great success in various applications. Recently it was applied to named entity recognition and achieved better results than BiLSTM [16]. Therefore, we also consider it as one baseline in our study. We use the standard attentional encoder-decoder model to generate label sequences. The parameter settings are the same as ours.

Table 2: The effectiveness of try block localization

Models	Precision	Recall	F1-score	Accuracy
CRF	64.7	26.9	38.0	52.4
BiLSTM	73.7	70.6	72.1	67.5
BiLSTM-CRF	51.6	59.2	55.2	54.4
Seq2Seq	77.1	60.0	67.5	48.7
Nexgen	**80.9**	**74.3**	**77.4**	**74.7**

4.4.2 Catch Block Generation. We select several commonly used models related to code completion and generation in SE community as comparative methods. For code completion, Language Models (LMs) [8] such as N-gram and LSTM are widely used to model and predict tokens of source code [26, 27, 39]. In addition, the machine translation model Seq2Seq is also used to generate pseudo code from source code [47] or APIs from natural language queries [24]. Thus we consider the following baselines.

- **N-gram**. Hindle et al. [27] first explored the naturalness of source code with the N-gram model and applied it to predict next tokens. A recent study [26] found that carefully adapting N-gram models for source code can yield performance that surpasses deep-learning models. Thus we borrow the best configuration from their work, which is a 6-gram model with Jelinek-Mercer interpolation.
- **LSTM**. As a neural language model, LSTM was also used for code completion and achieved good results [26, 38]. We compare our approach with LSTM-based language model and the hyperparameters are set to be the same as those in our approach.
- **Seq2Seq**. There are many Seq2Seq-based applications in software engineering. For instance, Oda et al. [47] used a statistical Seq2Seq model to generate pseudo code given the source code. Recently, the attentional Seq2Seq model [5] has proven to be more effective, so we use this model for comparison. The model configurations are the same as those used in our model.

4.5 Results

We investigate the following research questions to present the experimental results and analysis.

RQ1: How does our approach perform compared with baselines in try block localization?

To keep the results of our approach *Nexgen* comparable with those of baselines, we transform the predicted token-level labels of baselines into statement-level ones by making one statement label as Y if any token in it is predicted Y, otherwise N. Table 2 provides the experimental results of all the compared models. The best results are shown in bold.

It is clear that the CRF model performs worst among them. Although it has a good precision, the recall is quite low and so is F1-score. This means that there are few predicted try block locations in its results, which is confirmed by our manual inspection. As a result, the accuracy is only 52.4% since most of the predictions are all Ns, that is, it is prone to omit the exceptions.

Table 3: The effectiveness of catch block generation

Groups	Models	BLEU	Accuracy
Partial Context	N-gram	6.7	0.0
	LSTM	27.5	8.2
	Seq2Seq	38.0	18.9
Full Context	N-gram	5.0	0.0
	LSTM	29.4	8.1
	Seq2Seq	42.6	20.7
	Nexgen	**46.7**	**22.6**

By contrast, neural network based models achieve much better results. For example, the F1-score of BiLSTM is 72.1%, which outperforms CRF by more than 30%. This may be largely attributed to its ability to understanding semantics through word embedding. Because in CRF, the tokens are represented as numbers, which makes the features very sparse. In addition, it captures the dependencies between tokens by combining past and future information, which helps determine whether to predict the Y label when encountering a specific token (e.g., a predefined variable). Nevertheless, the performance drops more than 10% in terms of F1-score or Accuracy when combining them together (i.e., BiLSTM-CRF). Intuitively, the worse performance of the CRF module impairs the whole model. A deeper reason is that adding the CRF layer forces it to optimize the log-likelihood on the sequence level, whereas it is demonstrated to be not effective if using CRF independently. With regard to Seq2Seq, a surprising phenomenon is that it performs as well as BiLSTM such as the precision of 77.1%. But the Accuracy is only 48.7%, which is even lower than that of CRF. We inspected its predictions and found that it tends to generate shorter label sequences than the ground truth, indicating that it cannot tag the token sequence in a one-for-one manner due to the limitation of encoder-decoder architecture on this task.

At last, our approach achieves the best results among all the models. For example, it improves BiLSTM by 5.3% (F1-score) and 7.2% (Accuracy). The reason is that our model use a two-layered neural architecture to tag the source code at the statement level, which can significantly shorten the length of prediction time steps and get rid of the long-term dependency problem. As depicted in Table 1, the average length of Java methods in terms of tokens is 115.9 (AvgT), while it is only 14.7 (AvgS) in terms of statements. In addition, we employ attention mechanism at both token and statement levels, and thus can learn the distinction between important and unimportant elements.

RQ2: How does our approach perform in comparison with the baselines in generating catch block code ?
In this research question, we want to know how our approach and the baselines perform in generating catch block code and whether different contexts affect the performance. As shown in Table 3, we consider two different contexts: only the try blocks (Partial Context), the leading code and the try blocks (Full Context). For evaluation, we limit the length of the generated code to 100 because the LM-based models tend to generate very long sequences.

First, we can see that as a statistical language model, N-gram performs badly in this task. In particular, the BLEU score is only 6.7%, which is far from satisfaction. It is of no avail to use the full context for training such a model, and the performance even decreases a bit. The reason is that N-gram model only learns the probability of the next token within a fixed window (i.e., 6 tokens), no matter what the input is. It fails to capture enough context information. We also found that most of the generated code does not adhere to Java syntax, which may explain why its accuracy is 0. In contrast, LSTM achieves better results, since LSTM can capture more dependency information. When given the full context, LSTM learns to generate more correct tokens than the partial context and improves the BLEU value. However, the full context may lead to more noisy data from the leading code and make the accuracy slightly lower.

The Seq2Seq model shows a significant improvement over N-gram and LSTM. Because Seq2Seq separates the precondition (i.e., the leading code and try blocks) and the learning object (i.e., the catch blocks) into two different sequences, and can learn the knowledge of try blocks and catch blocks without interference, whereas the LM-based models predict tokens in a whole sequence. Also, the attention mechanism in Seq2Seq helps capture different weights of tokens in the encoder. Compared with the partial context, the full context also leads to an improvement, such as 4.6% higher in terms of BLEU, which shows that the leading code is actually helpful in Seq2Seq.

Our approach *Nexgen* outperforms all the baselines. Specifically, we improve the attentional Seq2Seq model by 4.1% and 1.9% in terms of BLEU and Accuracy, respectively. The reason is that *Nexgen* encodes the leading code and try blocks separately, and uses slicing-based attention to eliminate the noisy data from the leading code.

RQ3: To what extent does the components of our proposed models contribute to the effectiveness of both tasks?
This research question aims at analyzing the contributions of different components of our model to the overall effectiveness. We conduct an ablation study to answer this RQ. The results are shown in Table 4.

We start with the try block localization task (Task1). We evaluate the influence of removing two main attention modules from our original model (*Nexgen*). When replacing the token-level attention by the last time step of token encoder, we find that the recall becomes a bit higher from 74.3% to 74.8%, yet the F1-score and Accuracy decline by 0.5% and 0.6%. This indicates that the token-level attention indeed captures some more token-level semantics. Similarly, the performance also gets worse and the F1-score and Accuracy decrease by 0.8% and 0.9% when removing the statement-level attention. We can see that the statement attention contributes more to the overall performance than the token attention. If we remove both of the attention modules, the F1-score and Accuracy drops by 1.3% and 1.7%, respectively.

For the task of catch block generation (Task2), we obtained similar results. We first remove the standard attention over the leading code (Leading attention). The drops of BLEU and Accuracy show that different tokens in the leading code actually have different weights for capturing the code semantics. However, compared with removing slicing-based attention, the BLEU score of removing leading attention is 1.2% higher. This means that it is more effective to capture the dependencies in the leading code of the try blocks than the standard attention. Next, we remove slicing attention and

Table 4: The effect of model components on two tasks. The minus '–' symbol means removing one component from *Nexgen*

Task1					Task2		
Description	**Precision**	**Recall**	**F1-score**	**Accuracy**	**Description**	**BLEU**	**Accuracy**
– Token Attention	79.3	74.8	76.9	74.1	– Leading Attention	45.4	22.1
– Statement Attention	79.5	**74.9**	76.6	73.8	– Slicing Attention	44.2	22.0
– Both Attention	78.9	74.3	76.1	73.0	– Both Attention	30.2	14.5
Nexgen	**80.9**	74.3	**77.4**	**74.7**	*Nexgen*	**46.7**	**22.6**

Table 5: The score distribution of the generated catch blocks

Score	1	2	3	4	5	6	7	Avg	≥6	≤3
N-gram	70	21	8	0	1	0	0	1.41	0	99
LSTM	15	20	15	17	19	5	9	3.56	14	50
Seq2Seq	8	8	15	20	15	9	25	4.53	34	31
Nexgen	9	4	12	17	13	12	33	4.89	45	25

replace leading attention over the leading code with just the last time step of the leading code encoder. It can be seen that the performance degrades significantly by 16.5% (BLEU) and 8.1% (Accuracy), indicating that the leading code indeed introduces noisy data and influences the performance.

In summary, the token-level attention and statement-level attention components of the try block locator have a relatively small influence on the performance, while the leading attention and slicing-based attention mechanisms of the catch block generator are more effective and contribute significantly to the overall performance.

4.6 Human Evaluation

For the task of catch block generation, we use the quantitative metric BLEU to compare the code generated by different methods. BLEU calculates the textual similarity between the reference and the generated code, rather than the semantic similarity. Thus we perform human evaluation to complement the quantitative evaluation.

We invite 12 evaluators to assess the quality of catch block code generated by our approach *Nexgen* and the three baselines. They are undergraduate, master and Ph.D. students in CS with 1-6 years of experience in Java programming. The three baselines N-gram, LSTM and Seq2Seq are selected from the better ones in the partial or full context. We randomly choose 100 try-catch pairs from the testing set of CBGD and their catch block code produced by the three baselines and our approach, and evenly divide them into four groups. Each group is assigned to three different evaluators since such redundancy can help obtain more consistent results. For each try-catch pair, we show its leading code and try block, the reference catch blocks, and four results of catch blocks generated by the three baselines and our approach. The four generated results of catch blocks are randomly ordered, thus the evaluators have no idea which catch block code is produced by which approach. The evaluators can select a score between 1 to 7 to measure the semantic similarity between the generated catch blocks and the reference, where 1 means "Not Similar At All" and 7 means "Highly Similar/Identical". The higher scores mean that the corresponding generated catch blocks are more semantically similar to the reference. For each generated result of catch blocks, we get three scores from evaluators and choose the median value as the final score.

Table 5 shows the score distribution of the generated catch blocks. We can see that our approach achieves the best scores and improves the average (Avg) score from 1.41 (N-gram), 3.56 (LSTM) and 4.53 (Seq2Seq) to 4.89. Specifically, among the randomly selected 100 try-catch pairs, our approach can generate 33 highly similar or even identical catch blocks with the reference ones (score = 7), 45 good catch blocks (score ≥ 6). Our approach also receives the smallest number of negative results (score ≤ 3). Based on the 100 final scores for each approach of N-gram, LSTM, Seq2Seq and *Nexgen*, we conduct Wilcoxon signed-rank tests [62]. Comparing our approach with N-gram, LSTM, and Seq2Seq, the p-values of Wilcoxon signed-rank tests at 95% confidence level are 2.2e-16, 4.5e-09 and 0.0076, respectively, showing that the improvements achieved by our approach are statistically significant. In summary, the results of human evaluation confirm the effectiveness of the proposed approach.

5 DISCUSSION

5.1 Case study

Now we discuss the superiority and limitation of our model for catch block generation by analyzing two examples, which are shown in Table 6. Example 1 shows one incomplete Java method "add" including the leading code and try block. The developer wants to catch "IllegalStateException" thrown by "map.put" in the try block when multiple objects write data into the map and incur conflicts. Then the catch block handles the exception by converting and throwing the exception to notify upper-layer methods. In the full context, N-gram generates very long code with wrong Java syntax (we omit 94 more tokens here). Since there are many noisy statements in the leading code, LSTM and Seq2Seq fail to report the proper exception type. For example, Seq2Seq generates ClassCastException. By contrast, *Nexgen* correctly predicts the exception type and the whole catch block, since *Nexgen* treats the leading code and the try block separately, and uses the slicing-based attention to remove irrelevant and noisy statements.

Although *Nexgen* outperforms the baselines, we do not claim that it has already matured and is ready for use in practice. Indeed, our approach only makes one step further towards automated exception handling and it still has some limitations. In Example 2, although our approach successfully predicts the exception type FileNotFoundException, it fails to correctly handle it in the catch block. Specifically, it logs the event maybe because logging information for FileNotFoundException is a very common behavior in the training set. But the correct exception handling code is to set "mapping" as null, which possibly relies on the project-specific features

Table 6: Two examples of catch block generation

	Example 1	Example 2

Code	public void add(Session session, Row row){
	TransactionMap<Value, Value> map =
	getMap(session);
	ValueArray array = convertToKey(row, null);
	boolean checkRequired =
	indexType.isUnique()&&
	!mayHaveNullDuplicates(row);
	if(checkRequired){
	checkUnique(map, array, Long.MIN_VALUE);
	}
	try{
	map.put(array, ValueNull.INSTANCE);
	}
		private boolean revalidate(boolean flag){
		if(mapping==null || flag){
		File catalog = findFile(dtdSetFolder,
		CATALOG_FILE_NAME);
		if(catalog == null){ mapping = null;}
		else
		try{
		mapping = parseCatalog(new
		InputStreamReader(
		new FileInputStream(catalog)));
		}

Reference	catch(IllegalStateException e){throw mvTable.convertException(e);}	catch(FileNotFoundException exc){mapping = null;}
N-gram	catch(final IOException x);...(94 more tokens)	catch(IOException ioe){throw new KafkaCruiseControlException(e);...(89 more tokens)
LSTM	catch(Exception e){throw new RuntimeException(e); }	catch(IOException e){throw new RuntimeException(e);}
Seq2Seq	catch(ClassCastException e){throw new UnsupportedOperationException();}	catch(IOException exc){return false;}

Nexgen	catch(IllegalStateException e){throw mvTable.convertException(e);}	catch(FileNotFoundException exc){
		Logger.getLogger(<unk>.class.getName()).
		log(Level.INFO, null, exc);}

and personal preferences of developers. Our approach also encounters the problem of out of vocabulary and replaces the class name with "<unk>". In our future work, we will improve our approach by incorporating the knowledge of the exception specifications from projects and designing mechanisms that can overcome the large vocabulary problem.

5.2 Why is *Nexgen* better?

As stated previously, learning to handle exceptions includes two main successive tasks: finding potential exceptions and writing code to handle them. The common challenge of them is how to understand the semantics of given code snippets. Compared with traditional approaches, *Nexgen* learns the semantics of code from the following two aspects.

First, traditional approaches like static analysis or N-gram model have been applied in finding string-handling errors [20] or recommending exception-handling APIs [45], but they ignore the semantics of tokens (e.g., treating them as numerical IDs). In contrast, *Nexgen* adopts the word embedding technique to automatically learn the semantics of tokens by mapping them into a high-dimensional vector space. This is a fundamental component of many deep learning based approaches to code analysis [1, 24] because tokens with similar embeddings tend to be used in similar contexts, which helps determine whether a same/similar exception would occur and how it can be handled. For example, even the Java tokens "AudioInputStream" and "StringBufferInputStream" have different names, they are used in similar contexts and thus possibly throw the "IOException". Therefore, ignoring the semantics of tokens may lead to a wrong result.

Second, it is also necessary to know the deep semantics by understanding dependencies in source code. Statistical models (e.g., N-gram) consider only the probability distribution of code tokens and may have a poor performance. Program analysis can analyze the data-flow and control-flow dependencies, which has been shown effective for compilable and complete source code. Unlike them, *Nexgen* learns the internal dependencies within methods for incomplete code fragments. On the one hand, similar to existing studies [23, 69], it utilizes the LSTM to capture the sequential dependencies [70] of tokens and statements, so that exception-prone statements can be identified based on the context. As shown in Example 1 of Table 6, *Nexgen* identified the statement "map.put" instead of "convertToKey" as the exceptional one because the statement "checkUnique" is already used to check the variables, hence it only needs to check the latter statement. On the other hand, when generating catch blocks, *Nexgen* can learn the explicit data dependencies for eliminating noisy data from the leading code by the standard attention and our slicing-based attention mechanisms. In this way, *Nexgen* can understand the deep code semantics by capturing the sequential dependencies and the data dependencies in code fragments.

6 THREATS TO VALIDITY

There are three main threats to validity of our evaluation.

- In this work, we only collected Java source code to construct the datasets for out tasks. It remains unknown whether our approach will perform well on other programming languages such as C# and Python. However, we believe Java is representative because of its popularity in real-world software applications. In our future work we will collect data from programs in different programming languages to further evaluate the proposed approach.

- The quality of the Java methods may affect the effectiveness of our approach. To mitigate it, we selected top 2,000 projects at GitHub as corpus by the numbers of their stars and forks, which indicate their impact and popularity. We filter out simple methods (e.g., get/set methods) that have less than 7 lines of code. We also discard the exception handling code with bad practices, for example, swallowing exceptions. Still, we cannot guarantee that all data we collected is of high quality. We will explore effective techniques to improve data quality in the future.
- There may exist some score bias in the human evaluation, since developers can make different judgements on the same generated code according to their different experiences. Hence, each generated code is assigned to three redundant evaluators and we take the median value as the final score to reduce the bias.

7 RELATED WORK

7.1 Exception handling

Many studies have been conducted to understand exception handling practices [12, 13, 19, 55]. For example, Sena et al. [55] investigated potential impact of the exception handling strategies on the client applications by exception flow analysis and manual inspections. They found that 20.71% of the bug reports are related to the anti-patterns of exception handling. Padua et al. [19] performed an empirical study of the relationship between exception handling practices and software quality (measured by the probability of having post-release defects). The case study on open-source Java and C# projects showed that exception flow characteristics in Java projects have a significant relationship with post-release defects. Even so, previous studies revealed that developers tend to avoid it or misuse exception handling because they consider it hard to learn and to use [4, 10, 32, 57]. For example, Cabral et al. [10] studied exception handling practices from 32 projects in both Java and .Net and unveiled suboptimal practice of exception handling by developers. Shah et al. [57] interviewed Java developers in order to contrast the viewpoints of experienced and inexperienced developers regarding exception handling. They recognized that developers tend to ignore the proper implementation of exception handling until defects are found, although developers should do it in the early releases of a system. All these studies indicate that there is a need for automatic techniques to facilitate the writing of exception handling code for developers.

Meanwhile, prior research also explored different kinds of approaches to exception handling [6, 11, 45, 51, 58]. Cabral et al. [11] proposed a prototype transactional model that aims at automatically recovering runtime environments at the platform level, instead of writing exception handling code. Although this proposal is promising, its overhead is high and its precision is low due to the complexity of exceptions. Subsequent research focuses on recommending source code related information to aid programming practice. For example, Barbosa et al. [6] proposed a heuristic strategy that is aware of the global context of exceptions and produces recommendations on the violations of exception handling. Nguyen et al. [45] proposed a fuzzy and N-gram based approach to recommend exception types and repairing method calls for exception handling code. Such approaches can provide useful knowledge to assist developers in writing exception handling code, but developers still need to design the whole logic of exception handling and modify the code to fit the context. Compared with previous work, we provide a deep learning based approach to automated exception handling, which can predict the locations of try blocks and generate the complete catch blocks.

7.2 Code completion and generation

There exists a rich literature of research work on code completion in SE, most of which relies on language models (LMs). For example, Hindle et al. [27] used the N-gram model on lexical tokens to predict the next token. Hellendoorn et al. [26] performed an extensive comparison between N-gram and LSTM-based LMs for source code, and concluded that N-gram models can outperform LSTM models if carefully adapted. Apart from token-level code completion, Nguyen et al. [44] combined program analysis and statistical LM in the process of statement completion. Recently, some research intends to generate code in various scenarios [17, 24, 47, 60]. For example, Oda et al. [47] used a statistical machine translation model to generate pseudo code given the source code. Tufano et al. [60] investigated the NMT model (also called Seq2Seq) on pairs of code components before and after the implementation of the changes provided in the pull requests. Their results showed that NMT can automatically replicate the changes implemented by developers during pull requests. Similarly, Chen et al. [17] proposed a neural model based on Seq2Seq, which learns from pairs of the original version and fixed version of buggy programming lines. There are also some recent studies in NLP community, which propose neural network based models to generate code from natural language descriptions [31, 66]. Different from the above work, our work generates exception handling code instead of general code.

8 CONCLUSION

In this paper, we propose a novel deep learning based approach to automated exception handling. We decompose the problem into two coherent tasks, namely try block localization and catch block generation, whose targets are to recognize potential exceptions within a Java method and generate code to handle them, respectively. For both tasks, we design neural models with attention mechanisms to capture deep semantics. We conduct extensive experiments to evaluate our approach. All results demonstrate the superiority of our models on both tasks. Although our results are significantly better that those of the baselines, they can still be improved before being applied in practice. Our work can be considered as one of the first steps towards automated exception handling and we hope it can inspire follow-up research work.

Our source code and experimental data are available at https://github.com/zhangj111/nexgen.

ACKNOWLEDGMENTS

This work was supported partly by National Key Research and Development Program of China (No.2018YFB1004805), partly by National Natural Science Foundation of China (No.61702024, 61932007, 61972013 and 61421003) and ARC DP200102940.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 38–49.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 404–419.

[3] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2018. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering* 45, 12 (2018), 1170–1188.

[4] Muhammad Asaduzzaman, Muhammad Ahasanuzzaman, Chanchal K Roy, and Kevin A Schneider. 2016. How developers use exception handling in Java?. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 516–519.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015*.

[6] Eiji Adachi Barbosa and Alessandro Garcia. 2017. Global-aware recommendations for repairing violations in exception handling. *IEEE Transactions on Software Engineering* 44, 9 (2017), 855–873.

[7] Eiji Adachi Barbosa, Alessandro Garcia, Martin P Robillard, and Benjamin Jakobus. 2015. Enforcing exception handling policies with a domain-specific language. *IEEE Transactions on Software Engineering* 42, 6 (2015), 559–584.

[8] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A neural probabilistic language model. *Journal of machine learning research* 3, Feb (2003), 1137–1155.

[9] Yoshua Bengio, Paolo Frasconi, and Patrice Simard. 1993. The problem of learning long-term dependencies in recurrent networks. In *IEEE international conference on neural networks*. IEEE, 1183–1188.

[10] Bruno Cabral and Paulo Marques. 2007. Exception handling: A field study in java and. net. In *European Conference on Object-Oriented Programming*. Springer, 151–175.

[11] Bruno Cabral and Paulo Marques. 2008. A case for automatic exception handling. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 403–406.

[12] Nélio Cacho, Eiji Adachi Barbosa, Juliana Araujo, Frederico Pranto, Alessandro Garcia, Thiago Cesar, Eliezio Soares, Arthur Cassio, Thomas Filipe, and Israel Garcia. 2014. How does exception handling behavior evolve? an exploratory study in java and c# applications. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 31–40.

[13] Nélio Cacho, Thiago César, Thomas Filipe, Eliezio Soares, Arthur Cassio, Rafael Souza, Israel Garcia, Eiji Adachi Barbosa, and Alessandro Garcia. 2014. Trading robustness for maintainability: an empirical study of evolving c# programs. In *Proceedings of the 36th International Conference on Software Engineering*. 584–595.

[14] Rich Caruana, Steve Lawrence, and C Lee Giles. 2001. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *Advances in neural information processing systems*. 402–408.

[15] Chien-Tsun Chen, Yu Chin Cheng, Chin-Yun Hsieh, and I-Lang Wu. 2009. Exception handling refactorings: Directed by goals and driven by bug fixing. *Journal of Systems and Software* 82, 2 (2009), 333–345.

[16] Lingzhen Chen and Alessandro Moschitti. 2018. Learning to Progressively Recognize New Named Entities with Sequence to Sequence Models. In *Proceedings of the 27th International Conference on Computational Linguistics*. 2181–2191.

[17] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* (2019).

[18] Roberta Coelho, Awais Rashid, Alessandro Garcia, Fabiano Ferrari, Nélio Cacho, Uirá Kulesza, Arndt von Staa, and Carlos Lucena. 2008. Assessing the impact of aspects on exception flows: An exploratory study. In *European Conference on Object-Oriented Programming*. Springer, 207–234.

[19] Guilherme B de Pádua and Weiyi Shang. 2018. Studying the relationship between exception handling practices and post-release defects. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 564–575.

[20] Aritra Dhar, Rahul Purandare, Mohan Dhawan, and Suresh Rangaswamy. 2015. CLOTHO: saving programs from malformed strings and incorrect string-handling. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 555–566.

[21] Alessandro F Garcia, Cecília MF Rubira, Alexander Romanovsky, and Jie Xu. 2001. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of systems and software* 59, 2 (2001), 197–222.

[22] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. 2000. *The Java language specification*. Addison-Wesley Professional.

[23] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.

[24] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 631–642.

[25] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. 2003. *C# language specification*. Addison-Wesley Longman Publishing Co., Inc.

[26] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 763–773.

[27] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 837–847.

[28] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[29] Zhiheng Huang, Wei Xu, and Kai Yu. 2015. Bidirectional LSTM-CRF models for sequence tagging. *arXiv preprint arXiv:1508.01991* (2015).

[30] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. 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)*, Vol. 1. 2073–2083.

[31] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping Language to Code in Programmatic Context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 1643–1652.

[32] Mary Beth Kery, Claire Le Goues, and Brad A Myers. 2016. Examining programmer practices for locally handling exceptions. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 484–487.

[33] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[34] Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional LSTM feature representations. *Transactions of the Association for Computational Linguistics* 4 (2016), 313–327.

[35] George J Klir and Bo Yuan. 1996. Fuzzy sets and fuzzy logic: theory and applications. *Possibility Theory versus Probab. Theory* 32, 2 (1996), 207–208.

[36] John D Lafferty, Andrew McCallum, and Fernando CN Pereira. 2001. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of the Eighteenth International Conference on Machine Learning*. 282–289.

[37] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. 2016. Neural Architectures for Named Entity Recognition. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 260–270.

[38] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. 4159–4165.

[39] Chang Liu, Xin Wang, Richard Shin, Joseph E Gonzalez, and Dawn Song. 2016. Neural code completion. (2016).

[40] Edward Loper and Steven Bird. 2002. NLTK: The Natural Language Toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*.

[41] Xuezhe Ma and Eduard Hovy. 2016. End-to-end Sequence Labeling via Bidirectional LSTM-CNNs-CRF. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1064–1074.

[42] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering* 21, 5 (2016), 2146–2189.

[43] Robert Miller and Anand Tripathi. 1997. Issues with exception handling in object-oriented systems. In *European Conference on Object-Oriented Programming*. Springer, 85–103.

[44] Son Nguyen, Tien Nguyen, Yi Li, and Shaohua Wang. 2019. Combining Program Analysis and Statistical Language Model for Code Statement Completion. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 710–721.

[45] Tam Nguyen, Phong Vu, and Tung Nguyen. 2019. Recommending exception handling code. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 390–393.

[46] Ana Filipa Nogueira, José CB Ribeiro, and Mário A Zenha-Rela. 2017. Trends on empty exception handlers for Java open source libraries. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 412–416.

[47] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 574–584.

[48] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 311–318.

[49] Terence Parr. 2013. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.

[50] Juan Antonio Perez-Ortiz and Mikel L Forcada. 2001. Part-of-speech tagging with recurrent neural networks. In *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No. 01CH37222)*, Vol. 3. IEEE, 1588–1592.

[51] Mohammad Masudur Rahman and Chanchal K Roy. 2014. On the use of context in recommending exception handling code examples. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 285–294.

[52] Adwait Ratnaparkhi. 1996. A maximum entropy model for part-of-speech tagging. In *Conference on Empirical Methods in Natural Language Processing*.

[53] Martin P Robillard and Gail C Murphy. 2003. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 12, 2 (2003), 191–221.

[54] Mike Schuster and Kuldip K Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing* 45, 11 (1997), 2673–2681.

[55] Demóstenes Sena, Roberta Coelho, Uirá Kulesza, and Rodrigo Bonifácio. 2016. Understanding the exception handling strategies of Java libraries: An empirical study. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 212–222.

[56] Hina Shah, Carsten Görg, and Mary Jean Harrold. 2008. Why do developers neglect exception handling?. In *Proceedings of the 4th international workshop on Exception handling*. 62–68.

[57] Hina Shah, Carsten Gorg, and Mary Jean Harrold. 2010. Understanding exception handling: Viewpoints of novices and experts. *IEEE Transactions on Software Engineering* 36, 2 (2010), 150–161.

[58] Suresh Thummalapenta and Tao Xie. 2009. Mining exception-handling rules as sequence association rules. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 496–506.

[59] Kristina Toutanova, Aria Haghighi, and Christopher D Manning. 2005. Joint learning improves semantic role labeling. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 589–596.

[60] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 25–36.

[61] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.

[62] Frank Wilcoxon, SK Katti, and Roberta A Wilcox. 1970. Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test. *Selected tables in mathematical statistics* 1 (1970), 171–259.

[63] Sam Wiseman and Alexander M Rush. 2016. Sequence-to-Sequence Learning as Beam-Search Optimization. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. 1296–1306.

[64] Nianwen Xue and Martha Palmer. 2004. Calibrating features for semantic role labeling. In *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*. 88–94.

[65] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. 2016. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*. 1480–1489.

[66] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 440–450.

[67] Ding Yuan, You Luo, Xin Zhuang, Guilherme Renna Rodrigues, and Xu Zhao. 2014. Simple Testing Can Prevent Most Critical Failures. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*.

[68] Benwen Zhang and James Clause. 2014. Lightweight automated detection of unsafe information leakage via exceptions. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 327–338.

[69] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *Proceedings of the 42nd International Conference on Software Engineering. IEEE.*

[70] 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 *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[71] GuoDong Zhou and Jian Su. 2002. Named entity recognition using an HMM-based chunk tagger. In *proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 473–480.