Neural Exception Handling Recommender for Code Snippets

Anonymous Author(s)

ABSTRACT

With practical code reuse, the code fragments from developer forums often migrate to applications. Owing to the incomplete nature of such fragments, they often lack the details on exception handling. The adaptation for exception handling to the codebase is not trivial as developers must learn and memorize what API methods could cause exceptions and what exceptions need to be handled. We propose DeepEx, an exception handling recommender that learns from complete code, and accepts a given Java code snippet and recommends 1) if a `try-catch` block is needed, 2) what statements need to be placed in a `try-catch` block, and 3) what exception types need to be caught in the `catch` clause. Inspired by the sequence chunking techniques in natural language processing, we design DeepEx via a multi-tasking model with the fine-tuning of the large language model CodeBERT for the three above exception handling recommending tasks. Via the large language model, we enable DeepEx to learn the surrounding context, leading to better learning the identities of the APIs, and the relations between the statements and the corresponding exception types needed to be handled.

Our empirical evaluation on a real-world dataset shows that DeepEx achieves a high accuracy of **98.3%** and improves relatively by **56%** over the state-of-the-art approach in `try-catch` block necessity checking. Moreover, it can correctly decide both the need of `try-catch` block(s) and the statements to be placed in such blocks in **79.4%** of the cases, an improvement of **62X** over the baseline. Importantly, with an accuracy of **71.5%** (an improvement of **146X** over the baseline), DeepEx correctly recommend exception handling in all three tasks. Our extrinsic evaluation also shows that with its recommendations, DeepEx improves by **YY.Y%** in F-score over the existing approach in detecting exception-related bugs.

1 INTRODUCTION

The online question and answering (Q&A) forums, e.g., StackOverflow (S/O) provide important resources for developers to learn how to use software libraries and frameworks. While the code snippets in an S/O answer are good starting points, they are often incomplete with several missing details, even with ambiguous references, etc. Zhang *et al.* [**?**] have conducted a large-scale empirical study on the nature and extent of manual adaptations of the S/O code snippets by developers into their Github repositories. They reported that the adaptations from S/O code examples to their Github counterpart projects are prevalent. They qualitatively inspected all the adaptation cases and classified them into 24 different adaptation types. They highlighted several adaptation types including *type conversion*, *handling potential exceptions*, and *adding if checks* [**?**]. Among them, adding a `try-catch` block to wrap the code snippet and listing the handled exceptions in the `catch` clause are frequently performed, yet not automated by existing tools.

The adaptation process for exception handling is not trivial as Nguyen *et al.* [**?**] have reported that it is challenging for developers to learn and memorize what API methods could cause exceptions and what exceptions need to be handled. Kechagia *et al.* [**?**] found that 19% of the crashes in Android applications could have been caused by insufficient documented exceptions in Android APIs. Thus, it is desirable to have an automated tool to recommend proper exception handling for the adaptation of online code snippets.

There exist several approaches to automatic recommendation of exception handling [**? ? ? ? ? ?**]. They can be classified into four categories. The first category of approaches relies on a few *heuristics* on exception types, API calls, and variable types to recommend exception handling code [**?**]. These heuristic-based approaches do not always work in all cases. The second category of approaches utilized *exception handling policies*, which are enforced in all cases [**? ?**]. However, the policies need to be pre-defined and encoded within the recommending tools. This is not an ideal solution considering the fast evolution of software libraries. To enable more flexibility than policy enforcement, the third category leverages *mining algorithms* that derive similar exception handling for two similar code fragments [**?**]. While avoiding hard-coding of the rules, these mining approaches suffer the issue of how much similar for two fragments to be considered as having similar exception handling. For the mining approaches, deterministically setting a threshold for frequent occurrences is also challenging.

To provide more flexibility in code matching, the fourth category follows *information retrieval* (IR) [**?**]. XRank [**?**] takes as input source code and recommends a ranked list of API method calls in the code that are potentially involved in the exceptions in a `catch-try` block. XHand [**?**] recommends the exception handling code in a `catch` block for a given code. Both use a fuzzy set technique to compute the associations between the API calls (e.g., `newBufferedReader`) and the exceptions (e.g., `IOException`).

While the IR-based approach achieves higher accuracy than the others [**?**], it has key limitations. First, it is not trivial to **pre-define a threshold** for feature matching for a retrieval of an exception type or an API element. The effectiveness of those IR techniques depends much on the correct value of such pre-defined threshold. Second, the IR-based techniques rely on the lexical values of the code tokens and API elements, whose names can be *ambiguous* in an incomplete code snippet. For example, the `Document` class in `org.w3c.dom` of the W3C library has the same simple name as the `Document` class in `com.google.gwt.dom.client.Document` of Google Web Toolkit library (GWT). An API method to open/write/read a `Document` in the W3C library might need to catch a different set of exceptions than the one in GWT. Those IR-based techniques are not sufficiently flexible to handle such **ambiguous names**. Third, the IR techniques *do not consider the* **context** *of surrounding code*, thus, cannot leverage the *dependencies* among API elements to resolve the ambiguity of the names of the APIs and exceptions in an incomplete snippet.

In this paper, we propose DeepEx, a learning-based exception handling recommender, which accepts a given Java code snippet and recommends 1) *whether a try-catch block is needed for the snippet* (XBlock), 2) *what statements need to be placed in a* `try-catch` *block* (XState) and 3) *what exception types need to be caught in the* `catch` *clause* (XType). We find a motivation for such a data-driven,

learning-based approach from the previous studies reporting that exception handling for the API elements is frequently repeated across different projects [**?** **?**]. The rationale is that the designers of a software library have the intents for users to use certain API elements with corresponding exception types. Thus, we design DeepEx to learn from the statements in `try-catch` blocks and the exception types retrieved from *complete source code* in a large code corpus, and derive the above exception handling suggestions for the *(partial) code snippet* under study.

We leverage and fine-tune the large language model CodeBert [**?**] to capture the surrounding context with the dependencies among the API elements. Capturing such contextual information and the dependencies enable DeepEx to realize the idea *"Tell Me Your Friends, I'll Tell You Who You Are"* to learn the identities of the API elements in a given (in)complete code, leading to better learning in XBlock, XState and XType. Inspired by sequence chunking in natural language processing (NLP), we formulate our problem as detecting one or multiple chunks of consecutive statements that need `try-catch` blocks. DeepEx also has the three tasks in a *multi-tasking* mechanism to enable the mutual impact among the learning, leading to better performance in all three tasks.

Our aforementioned idea gives DeepEx three advantages over the state-of-the-art IR approach. First, with the learning-based approach, DeepEx does not rely on a pre-defined threshold for explicit feature matching for the retrieval of the API elements or exception types. Second, instead of learning only the associations between the API elements and corresponding exception types, DeepEx has advantages in both predicting and training. During predicting, for a given incomplete code, the context enables DeepEx to *learn the identities of the API elements via the dependencies/relations* among them in the context, thus, avoiding the name ambiguity. Let us call it *dependency context*. During training, the complete code enables the identifications (i.e., the fully-qualified names) of the API elements. Third, the context of surrounding code also helps the model implicitly learn the important features to connect between the API elements and the corresponding exception types.

We have conducted several experiments to evaluate DeepEx. We have collected a large dataset of 5,726 projects from Github, with 19,379 code snippets that contain `try-catch` blocks. The result shows that DeepEx achieves a high accuracy of **98.3%** and improves relatively by **56%** over the state-of-the-art approach in `try-catch` block necessity checking. Moreover, it can correctly decide both the need of `try-catch` block(s) and the statements to be placed in such blocks in **79.4%** of the cases, an improvement of **62X** over the baseline. Importantly, with an accuracy of **71.5%** (an improvement of **146X** over the baseline), DeepEx correctly recommend exception handling in all three tasks. Our extrinsic evaluation also shows that with its recommendations, DeepEx improves by **YY.Y%** in F-score over the existing approach in detecting exception-related bugs.

In brief, this paper makes the following major contributions:

1. [Neural Network-based Automated Exception Handling Recommendation]. DeepEx is the first neural network approach to automated exception handling recommendation in three above tasks. DeepEx works for either complete or incomplete code.

2. [Multi-tasking among three Exception Handling Recommendations] We formulate the problem as sequence chunking with a multi-tasking mechanism to learn for three above tasks.

```
1  public static void addLibraryPath(String pathToAdd) throws Exception {
2    final Field usrPathsField =
           ClassLoader.class.getDeclaredField("usr_paths");
3    usrPathsField.setAccessible(true);
4
5    //get array of paths
6    final String[] paths = (String[])usrPathsField.get(null);
7
8    //check if the path to add is already present
9    for(String path : paths) {
10     if(path.equals(pathToAdd)) {
11       return;
12     }
13   }
14
15   //add the new path
16   final String[] newPaths = Arrays.copyOf(paths, paths.length + 1);
17   newPaths[newPaths.length-1] = pathToAdd;
18   usrPathsField.set(null, newPaths);
19 }
```

Figure 1: StackOverflow post #15409223 on adding new paths for native libraries at runtime in Java

3. [Empirical Evaluation]. Our extensive evaluation shows DeepEx's high accuracy in exception handling recommendation as well as in exception-related bug detection. Data and code is available at [**?**].

2 MOTIVATION

2.1 Motivating Examples

Let us use a few real-world examples to explain the problem and motivate our approach. Figure 1 displays a code snippet in an answer to the StackOverflow (S/O) question 15409223 on how to *"add new paths for native libraries at runtime in Java"*. The code snippet serves as an illustration in the S/O post, thus, does not contain all the details on what exceptions that need to be handled. It contains only a throw of a generic `Exception` in the method header (`addLibraryPath`). From Zhang *et al.*'s study [**?**], this code snippet was adopted by developers into their Github project named `armint` (Figure 2). `armint`'s developers handle in a `try-catch` block several exceptions caused by `java.lang.Class.getDeclaredField(...)` (line 7) according to JDK's documentation, e.g., `NoSuchFieldException`, `SecurityException`, `IllegalArgumentException`, and `IllegalAccessException` (line 24, Figure 2).

The manual adaptation on exception handling by inserting a `try-catch` block is quite popular, yet not automated by any tools [**?**]. Such manual adaptation for a code snippet could lead to exception-related bugs, which could cause serious issues including crashes or unstable states. It is not trivial for developers to memorize what API methods could cause exceptions and what exceptions need to be handled [**?**]. Thus, it is desirable to have an automated tool to recommend proper exception handling in order to adapt the incomplete code snippets. Such a tool could recommend if a `try-catch` block is needed for the snippet, what lines need to be included in that block, and what exception types need to be handled.

Observation 1 (**Exception Handling Recommendation**). *Automated recommendation to handle exceptions is desirable to assist developers in adapting incomplete code snippets into their codebases.*

As explained in Section 1, four categories of automated approaches have been proposed to recommend exception handling [**?** **?** **?** **?** **?**]. While early approaches were not effective in all cases due to their *heuristics* [**?**], the *exception policies* are too strict in

```java
/** ...
 * taken from http://stackoverflow.com/questions/15409223/
 * adding-new-paths-for-native-libraries-at-runtime-in-java
 */
private static void addLibraryPath(String pathToAdd) {
  try {
    final Field usrPathsField =
          ClassLoader.class.getDeclaredField("usr_paths");
    usrPathsField.setAccessible(true);

    // get array of paths
    final String[] paths = (String[]) usrPathsField.get(null);

    // check if the path to add is already present
    for (String path : paths) {
      if (path.equals(pathToAdd)) {
    return;
      }
    }

    // add the new path
    final String[] newPaths = Arrays.copyOf(paths, paths.length + 1);
    newPaths[newPaths.length - 1] = pathToAdd;
    usrPathsField.set(null, newPaths);
  } catch (NoSuchFieldException | SecurityException |
        IllegalArgumentException | IllegalAccessException e) {
    throw new RuntimeException(e);
  }
}
```

Figure 2: GitHub project `armint` adapts SO post in Figure 1

enforcing them, yet requires the rules to be encoded in the tools [**? ?**]. The state-of-the-art *information retrieval*-based approaches (e.g., XRank/Xhand [**?**]) have been shown to outperform the existing approaches including the *mining approaches* [**?**] (which suffers the issue of setting a threshold for frequent occurrences).

However, the state-of-the-art, IR-based approaches [**?**] have the limitations. First, it is not trivial to pre-define a threshold for feature matching for a retrieval, e.g., the threshold to determine the associations between an API call (e.g., `getDeclaredField`) and an exception type (e.g., `NoSuchFieldException`). Thus, the pre-defined threshold affects much their effectiveness. Second, relying on the lexical values of API elements' names, they suffer the issue of ambiguous names of the APIs or exceptions in an incomplete code snippet (e.g., the API method `get` at line 6 of Figure 1 occurs in multiple libraries), which might not be parseable for fully-qualified name resolution. Thus, this reduces effectiveness. Finally, they consider only the associations between an API method and an exception type, and discard the surrounding context. For example, they compute the association between the names of the API call (e.g., `getDeclaredField`) and the exceptions to be handled (e.g., `NoSuchFieldException`, `SecurityException`, etc.). Without the context, it is challenging to decide the identities of the APIs and exceptions via only simple names.

```java
public Object readField(Class<?> clazz, String name, Object instance) {
  try {
    Field field = clazz.getDeclaredField(name);
    if (!field.isAccessible()) {
      field.setAccessible(true);
    }
    return field.get(instance);
  } catch (NoSuchFieldException | SecurityException |
        IllegalArgumentException | IllegalAccessException e) {
   throw new RuntimeException("Cannot read field value: " + clazz.getName()
         + "#" + name, e);
  }
}
```

Figure 3: Project `quarkus` with same exception handling

```java
Charset charset = Charset.forName("US-ASCII");
try {
    BufferedReader reader = Files.newBufferedReader(file, charset);
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

Figure 4: Using `newBufferedReader` to read from a file

Now, consider the complete code example in Figure 3 from the Github project named `quarkus`. While there are differences between the complete code in Figure 3 and the adapted code in Figure 2, the lists of the handled exceptions are the same (line 8 in Figure 3 and line 24 in Figure 2) due to the presence of the API call to `getDeclaredField` in both code. This is expected because the designers of the JDK library have the intent for developers to use the API method `getDeclaredField` within a `try-catch` block and to handle the list of exceptions as in line 8 of Figure 3. Thus, to adapt the incomplete code snippet in Figure 1, one could learn from the public code repositories to properly handle the exceptions.

Observation 2 (**Regularity of Exception Handling**). *Finding the patterns from complete code in existing code corpora could be a good strategy to* **learn to properly handle the exceptions** *in adapting an (incomplete) code snippet into a codebase.*

Observation 3 (**Relations between API methods and Exceptions**). *The presence of certain API elements helps decide the exceptions that need to be handled.*

For example, the relation between `java.lang.Class.getDeclaredField` and the exceptions `NoSuchFieldException`, `SecurityException`, `IllegalArgumentException`, and `IllegalAccessException` can be learned from the code corpora. Thus, a model can learn to recommend those exceptions for an incomplete code snippet involving `getDeclaredField`.

Observation 4 (**Surrounding Context help resolve name ambiguity**). *The surrounding code context can help resolve the ambiguity of the names of those elements in incomplete code snippets.*

For an incomplete code snippet, as explained earlier, the simple names of the API elements (methods, fields, classes) could be ambiguous. However, if a model can learn from the complete code the fully-qualified names of the API elements, the surrounding context consisting of those API elements and their program dependencies can help a model decide the correct identities of the API elements, leading to correct prediction of the handled exceptions.

In Figure 1, to derive the identities of `Field` (line 2), `getDeclaredField` (line 2), `setAccessible` (line 3), `get` (line 6), etc., a model could rely on the dependencies among them in the surrounding context. For example, the return type of `getDeclaredField` is `Field` (thanks to line 2), which has an API method named `setAccessible` (thanks to line 3) and another API method named `get` (thanks to line 6). Considering all those dependencies among the API elements in the context and with the knowledge learned from the complete code, a model could decide that in the code snippet, the identity of `Field` is `java.lang.Class.Field`, that of `setAccessible` is `java.lang.Class.Field.setAccessible`, and that of `get` at line 6 is `java.lang.Class.Field.get`.

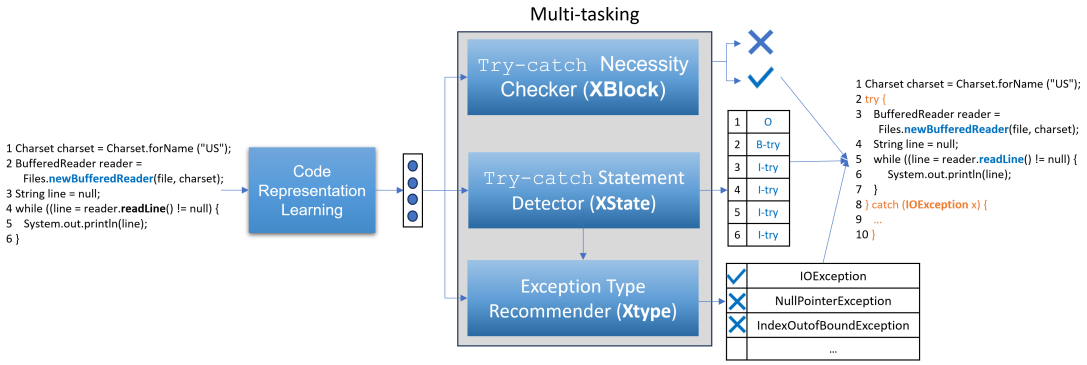

Figure 5: DeepEx: Architecture Overview

The rationale is that a model could see such dependencies among those API elements before in a complete code in training.

For the list of statements in an incomplete code, not all of them needs to be wrapped around in a `try-catch` block. For example, considering the example of using `newBufferedReader` in Figure 4. While the API call `java.nio.file.newBugfferedReader` needs to be within a `try-catch` block, the statement at line 1 to retrieve the character set does not. Moreover, the statement at line 5 with the API call to `readLine` needs to be wrapped in a `try-catch` block as well.

Observation 5 (**Learn to decide what statements to be in a Try-Catch block**). *A model can learn from the code corpora what statements need to be placed within a* `try-catch` *block or not.*

2.2 Key Ideas

We introduce DeepEx with the following three functionality for exception handling recommendation: given a Java code snippet, it will 1) predict if a `try-catch` block is required (XBlock), 2) point out which statements in the code snippet need to be placed in a `try-catch` block (XState), and 3) suggest what exceptions need to be caught in the `catch` clause (XType). Following the above Observations, we design DeepEx with the following key ideas:

2.2.1 **[Key Idea 1] Neural Network-based approach to Exception Handling Recommendation by Learning from Complete Code**. Instead of deterministically deriving the exceptions to be handled for a given (incomplete) code snippet, following Observation 2, we design a deep learning model (DL) to learn to properly handle the exceptions in the three above tasks. By learning from the `try-catch` blocks of the complete code in the open-source projects in the training process, our DL model can help the adaptation tasks.

2.2.2 **[Key Idea 2] Leveraging Context to avoid name ambiguity and Learning the Relations between API elements and Exception types**. Instead of learning only the associations between an API element and exception types as in IR-based approaches, we leverage as the context the complete code in the training corpus, which are parsable and provide the identities (i.e., FQNs) of the API elements. In predicting for a code snippet, DeepEx will also leverage the context and dependencies among the API elements to learn their identities (see Observation 4). Importantly, that leads to the learning of the relations between the key API elements in the context and the handled exception types (Observation 3).

2.2.3 **[Key Idea 3] Leveraging Sequence Chunking with a Large Language Model and Multi-tasking**. Inspired by the sequence chunking techniques [**?**] in NLP, we formulate our problem as identifying one or multiple chunks of consecutive statements that need to be placed within `try-catch` blocks. We leverage and fine-tune the large language model CodeBERT [**?**] to learn to the relations among statements with the API elements. We also leverage the multi-tasking framework for all three tasks XBlock, XState, and XType because the learning for one task can benefit for another task and vice versa.

3 DEEPEX OVERVIEW

Figure 5 displays the overview architecture of DeepEx. Generally, it has three main components dedicated to the three tasks: for a given (in)complete code snippet, 1) XBlock aims to check the necessity of `try-catch` blocks, 2)XState aims to detect which statements need to be in a `try-catch` block, and 3) XType aims to detect the exception types need to be caught in the `catch` clauses. We support the detection of one or multiple `try-catch` blocks if any.

During training, the code snippets with `try-catch` blocks are used as the positive samples and the ones without them as the negative samples. For a positive sample, the statements inside the `try` blocks and the exception types in the `catch` clauses are used as the labels for training. The negative samples are labeled as not needing a `try-catch` block. In prediction, DeepEx accepts as input any incomplete or complete code snippet without a `try-catch` block, and predicts the results for those tasks. The predicted results from XState and XType are considered only when XBlock predicts *Yes*, i.e., a need of a `try-catch` block for the given code snippet.

The input code is used as the input of a large language model to act as the code representation learning model to produce the vector representations for the (sub)tokens and statements in the source code. We use CodeBERT [**?**] as it is capable of producing embeddings that capture both the syntactic and semantic information.

The vectors are used as the inputs for three components. The prediction is made on the vectors that are attained by composing the embeddings of individual (sub)tokens into the ones for a statement and for a block of statements. First, XBlock is modeled as a binary classifier on deciding whether the input code needs at least one `try-catch` block. Second, inspired by the sequence chunking techniques [**?**] in NLP, we model the second task, XState, as

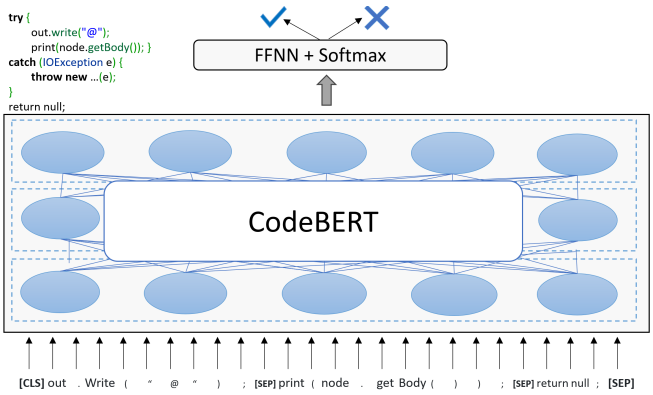

Figure 6: Try-catch Necessity Checker (XBlock)

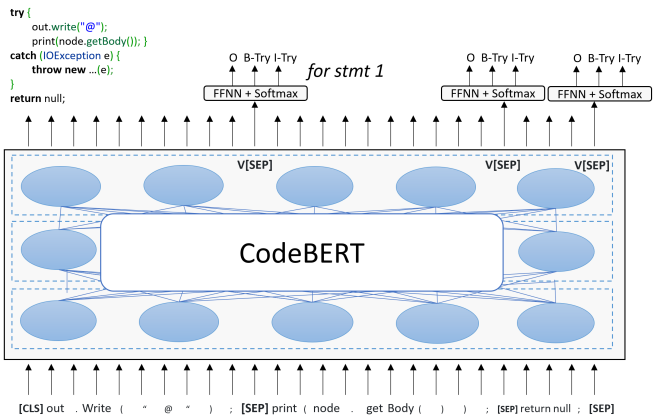

Figure 7: Try-catch Statement Detector (XState)

learning to tag/label each statement in the code snippet with either `O` (i.e., the statement is outside of a `try-catch` block), `B-try` (i.e., it is the beginning of a `try-catch` block), or `I-try` (i.e., it is inside of such a block). During training, the statements within or outside of the `try-catch` blocks enable us to build the tags/labels for them.

The last task, XType, is modeled as a set of binary classifiers, each is responsible for deciding whether an exception type of interest needs to be caught in the `catch` clause. An *Yes* outcome indicates the need to catch a specific exception type of interest in the set of libraries under consideration. An *No* outcome indicates otherwise. During training, the exception types for each `try-catch` block in the positive samples are used as labels. During prediction, for each predicted block from XState (starting from a statement with `B-try` to the last respective `I-try`), XType uses the embeddings for those statements to predict the corresponding exception types. Finally, from the results in all three tasks, DeepEx forms the final output.

4 XBLOCK: TRY-CATCH NECESSITY CHECKER

Given an input code snippet, we first split it into the lines of code. Each line is then tokenized into sub-tokens using the CodeBERT tokenizer. We use a special separator token `[SEP]` to concatenate the tokenized lines, and add a `[CLS]` token at the beginning. As in CodeBERT, we take the `[CLS]` token to be the representation of the entire code snippet.

We fine-tune a CodeBERT(MLM) for this problem. Given the good performance of CodeBERT on many code-related downstream tasks (https://github.com/microsoft/CodeXGLUE), we expect it to be able to learn the correlation between important code tokens that would signal the need of exception handling. Importantly, by providing the code snippet, we expect to leverage the code context in which the API elements are used with regard to one another. For example, in Figure 5, CodeBERT is expected to learn that the APIs `newBufferedReader` of the class `Files` and `readLine` of the class `BufferedReader` are used often together in API usages and they require `IOException`. For the input incomplete code snippet, CodeBERT is expected to learn such relations/connections to avoid name ambiguity and to connect them with the exception types.

During training, as we use exactly one CodeBERT, all three modules (Sections 5 and **??**) contribute to the signal for updating the CodeBERT parameters. In XBlock, we feed the vector representation of the `[CLS]` token to a linear layer (Feed-forward neural network - FFNN) and use a softmax function to learn the decision as to whether the input code needs to handle any exceptions.

5 XSTATE: TRY-CATCH STATEMENT DETECTOR

XState also uses CodeBERT, the same one as in XBlock. However, we create a separate figure for each module so that it is easier to follow.

Each `[SEP]` token represents the line of code preceding it (Note that semicolon would not be as consistent a line separator as the `[SEP]` token, since a line of code may not end with a semicolon. And a semicolon often appears inside string literals and inside for-loop conditions.). We expect that the relationship between lines of code can be learnt through these `[SEP]` tokens during training, so that at prediction time, the model can determine for each line as to whether it is inside or outside a try-catch block. We devise three tags in the IOB2 format to accomplish this task: `O` tag means that the statement is outside any try-blocks; `B-Try` tag means the statement begins a try-block; and `I-Try` tag means the statement is inside a try-block and is not the first line in that try-block.

During training, tags of all the lines are known from the code. During prediction, the model will be able to assign tags to all the lines in the code snippet. From these tags, we get to know how many try-blocks should be added, and which lines of code belong to which try-block.

6 XTYPE: EXCEPTION TYPE RECOMMENDER

XType uses the same CodeBERT that is shared among other modules. We expect CodeBERT to learn the relation between statements in the same try-block, as well as the relation between the try-block and the corresponding exceptions that need to be caught.

For each try block, we first take the vector outputs of `[SEP]` tokens, from CodeBERT, that correspond to statements in the try block. Then, we add them together to get the vector representation of the try block, before feeding it into a linear layer and use a sigmoid function to do binary classification for the most common exceptions that are in the library. If we included more libraries for

the study, the classification head will likely need to change to a different one.

During training, we know all the tags associated with every line in the input code snippet, so we know how to connect them, while during prediction, the model uses tags predicted in XState in order to make the connection.

7 EMPIRICAL EVALUATION

7.1 Research Questions

We conducted several experiments to evaluate DeepEx. We seek to answer the following questions:

RQ$_1$. [Effectiveness on Try-Catch Necessity Checking] *How accurate is DeepEx in predicting whether a given code snippet needs to have a* `try-catch` *block?*

RQ$_2$. [Effectiveness on Try-Catch Statement Detection]. *How accurate is DeepEx in predicting which statements in a given code snippet needs to be placed in a* `try-catch` *block?*

RQ$_3$. [Effectiveness on Exception Type Recommendation]. *How accurate is DeepEx in recommending what exception types need to be handled in the* `catch` *clause of a* `try-catch` *block?*

RQ$_4$. [Ablation Study]. *How do various components in DeepEx affect its performance?*

RQ$_5$. [Extrinsic Evaluation on Exception-related Bug Detection]. *How well does DeepEx detect exception-related bugs?*

7.2 Empirical Methodology

7.2.1 Datasets. We conducted experiments on two datasets: 1) *Github dataset* for intrinsic evaluation on exception handling recommendation tasks (XBlock, XState, XType), and 2) *FuzzyCatch* dataset [**?**] for extrinsic evaluation on exception-related bug detection. We collected the Github dataset as follows. We first chose in Github 5,726 Java projects with the highest ratings that use the following libraries: jodatime, JDK, Android, xtream, GWT, and Hibernate. These are the well-established libraries that have been used in several prior research on the topics related to APIs [**? ?**]. We then selected the methods with at least one `try-catch` block. If there is one `try-catch` block, we take the body of the method as the code snippet. If there are multiple `try-catch` blocks in a method, we split the method into multiple code snippets by checking the above and below statements, one by one, starting from the closest one for each `try-catch` block, to verify if the statement is included in the other block. If not, we put it in the current code snippet. If yes, we finish the current one. In total, we have 19,379 code snippets containing one `try-catch` block as positive samples. We also randomly selected from the same Github projects the same amount of code snippets that do not have any `try-catch` block as the negative samples.

For extrinsic evaluation, we used FuzzyCatch dataset, provided by the authors of XRank/XHand [**?**], which contains 608 samples of methods with exception-related bugs (missing `try-catch` blocks or missing catching some exceptions). We also randomly selected from the projects in FuzzyCatch dataset the same amount of code snippets with no exception-related bugs as the negative samples.

7.2.2 RQ1. Effectiveness on Try-Catch Necessity Checking
. *Baselines.* We compared XBlock against XRank [**?**] (XRank is part of FuzzyCatch tool). XRank computed the exception risk score for each API call. If one score of a call in the snippet is higher than a threshold, it is considered as needing a `try-catch` block.

Procedure. We used the Github dataset and randomly split both the positive and negative sets into 80%, 10%, and 10% of the code snippets for training, tuning, and testing.

Tuning. We tuned DeepEx with autoML [**?**] for the following key hyper-parameters to have the best performance: (1) Epoch size (50, 100, 150); (2) Batch size (32, 64, 128); (3) Learning rate (0.001, 0.003, 0.005); (4) Vector length of feature embeddings and its output (64, 128, 256); (5) Number of R-GCN layers (4, 6, 8).

Metrics. We use **Recall, Precision**, and **F-score** to evaluate the performance of the approaches. They are calculated as $Recall = \frac{TP}{TP+FN}$, $Precision = \frac{TP}{TP+FP}$, $F\text{-score} = \frac{2*Recall*Precision}{Recall+Precision}$. TP: true positive, FN: false negative, and FP: false positive.

7.2.3 RQ2. Effectiveness on Try-Catch Statement Detection
. *Baselines.* None. FuzzyCatch (XRank/XHand) does not have this.

Procedure. We processed the Github dataset for this experiment as follows. For the snippets that need `try-catch`, but were predicted as not, we consider them as incorrect because the resulting statements predicted from XState do not make sense for incorrect XBlock's detection. The snippets that do not need `try-catch`, but were predicted as yes, we also consider them as incorrect for the same reason. Thus, for the evaluation of XState, we used the set of code snippets that XBlock predicted correctly as needing a `try-catch` block. We used the same data splitting scheme with 80%, 10%, and 10% for training, tuning, and testing as in RQ1. Each code snippet in the testing set and the trained R-GCN model in XBlock are the input of the GNNExplainer model in XState in this experiment.

Tuning. We used the same parameters as in GNNExplainer [**?**]. It also has a parameter on the limit of the number N of the nodes in the explanation sub-graph. We varied N from 1 to 10. The average number of statements in a `try-catch` block in Github dataset is 5.9.

Metrics. If XState predicts for a statement correctly if it is in a `try-catch` block or not, we count it as a correct case. Otherwise, it is an incorrect one. **Accuracy** is defined as the ratio between the number of correct statements over the total number of statements.

7.2.4 RQ3. Effectiveness on Exception Type Recommendation
. *Baselines.* None. XRank/Xhand do not support this feature.

Procedure. We processed the Github dataset for this experiment in the same manner as in RQ2 because the result of XType (i.e., what exception types need to be in the `catch` clause) makes sense only when one knows that the given code snippet needs to have a `try-catch` block. Thus, for the evaluation of XType, we used the set of code snippets that XBlock predicted correctly as needing a try-catch block. We used the same splitting as RQ2. The exception types in the `catch` clauses are used as the labels.

Tuning. We tuned DeepEx with autoML [**?**] for the key hyper-parameters to have the best performance as in RQ1 including epoch size, batch size, learning rate, vector length, and R-GCN layers.

Metrics. Let us use E and P to denote the set of the exception types in the oracle and the set of predicted ones for one code snippet in the testing set. We aim to measure 1) how precise DeepEx predicts the exception types (**Precision**), i.e., how accurate the predicted types in P, and 2) how much DeepEx can cover in its prediction

Table 1: Try-Catch Necessity Checking Comparison (RQ1)

	Github Dataset	
	DeepEx	XRank [?]
Recall	0.79	**0.81**
Precision	**0.68**	0.53
F-score	**0.73**	0.63

with respect to the oracle (**Recall**), i.e., how much of E that the predicted set P can cover. Toward measuring Precision and Recall, we define **Hit-n** as the number of the cases in which the predicted set P contains *at least n* correct exception types, i.e., P and E overlaps *at least n* exception types regardless of the sizes of both sets.

In Github dataset, more than 98.1% of the code snippets have 1–3 exception types in a `catch` clause. Thus, we compute Hit-n, Precision, and Recall for the size of E ($|E|$) from 1–3 and 3+, and for the size of P ($|P|$) from 1–3 and 3+. When computing Recall, we use the set E as the basis. Recall at a size N_E of E is computed as the ratio between Hit-n at that size and the total number of cases with that size N_E. We compute Recall for all N_E = 1–3, and 3+, and n=1–N_E. We also define **`Hit-All`**$_{Rec}$ as `Hit-`n when the number n of overlapping exception types is equal to $|E|$, i.e., all exception types in the oracle set for a code snippet are covered. When computing Precision, we use the set P as the basis. Precision at a size N_P of P is computed as the ratio between Hit-n at that size and the total number of cases with that size N_P. We compute Precision for all N_P = 1–3, and n=1–N_P. We also define **`Hit-All`**$_{Prec}$ as `Hit-`n when the number n of overlapping exception types is equal to $|P|$, i.e., all predicted types in the predicted set P for a snippet are correct.

7.2.5 RQ4. Ablation Study. In this experiment, we aim to evaluate the impact on the performance of the key features: sequences of code tokens and Abstract Syntax Tree. We removed one key feature at a time and compared the performance with the original model to evaluate its impact. We used the same evaluation metrics as in the previous experiments. Because our model is designed with R-GCN, we cannot remove it to evaluate the impact of dependencies.

7.2.6 RQ5. Extrinsic Evaluation on Exception-related Bug Detection . Baselines. FuzzyCatch [**?**] leverages XRank to detect the exception-related bugs, which are the code snippets that were supposed to handle exceptions, but missed `try-catch` blocks and/or exceptions.

Procedure. We trained DeepEx on the Github dataset and detected the exception-related bugs in FuzzyCatch bug dataset.

Metrics. We compared the result against the oracle in FuzzyCatch dataset. If a model correctly detects a (non)buggy snippet (missing a `try-catch` block or exceptions), we consider it as correct. Otherwise, it is a miss. We use **Recall**, **Precision**, and **F-score** as in RQ1.

8 EMPIRICAL RESULTS

8.1 Comparison on Try-Catch Necessity Checking Effectiveness (RQ1)

As seen in Table 1, DeepEx achieves high performance across two datasets. With a Precision of 68%, it can decide correctly 2 out of 3 cases if a code snippet needs a `try-catch` block or not. With a Recall of 79%, DeepEx covers 4 out of 5 cases that needs to be placed in a `try-catch` block. Users just need to find 1 out of 5 cases. As a result, it achieves a high F-score of 0.73. In comparison, DeepEx improves relatively over XRank 28.3% in Precision and 12.3% in F-score.

Table 2: Try-Catch Statement Detection Effectiveness (RQ2)

	Accuracy									
	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10
DeepEx	0.42	0.47	0.59	0.66	0.71	0.74	0.76	0.77	0.78	0.79

Nx is number of nodes in the explanation sub-graph (`try-catch` block)

Examining the result, we reported the following. First, if the association score of *only one API method* in the snippet and *one exception* is higher than a threshold, XRank decides that a `try-catch` block is needed. Thus, it often tends to output "Yes". *Its recall is slightly better, but precision is just marginally better than a coin toss (0.53) in our balanced dataset. That leads to lower F-score than DeepEx.* Second, the decisions on the necessity of a `try-catch` block or the exception types depend on the pre-defined thresholds in XRank on those association scores. Thus, those pre-defined thresholds might not be suitable across the libraries. Third, for the incomplete code snippets in which the names of the API methods in different packages or libraries are the same (e.g., `toString` or `getText` in various JDK packages), XRank cannot distinguish them and use one entry in the dictionary for them due to its IR approach. In contrast, unlike XRank which considers only the API method calls in a `try-catch` block, DeepEx considers the code in the block as the context to learn the program dependencies/relations among the names of those API elements. That is, it leverages the relations among the names of API elements to learn their identities, thus, deciding better the need of `try-catch` blocks and the corresponding exception types.

Take as an example a code snippet (not shown) in our dataset with the presence of `getText`. This name is popular with a very large number of API method candidates. However, considering the relation between `css` and `getText` in the code '`...css().getText()`', the number of candidates for `getText` is only 4. Finally, considering the return value of `getText` as an argument of `setInnerText(...)` in the code '`setInnerText(...css().getText())`', only one candidate is remained: `com.google.gwt.resources.client.CssResource.getText()`. Thus, those relations actually help identify the API elements, leading to better decision in DeepEx on the `try-catch` block and exception types. Because it has not seen any `try-catch` block involving `com.....getText()` and those related ones, DeepEx decides that the code snippet does not need a `try-catch` block. In contrast, XRank considers only the *pairwise* associate scores between an *individual API method call* and the exception types in a `catch` clause. It disregards those above relations/dependencies among the API names. Thus, it might misunderstand that `getText` needs a `try-catch` due to the co-occurrences of other API elements that need one. That is, without the dependencies, XRank might make incorrect identification of the API elements via their names, leading to incorrect exception recommendation.

8.2 Try-Catch Statement Detection (RQ2)

Table 2 displays the result on detecting the statements that need to be placed in a `try-catch` block. N_x is a parameter in GNNExplainer that defines the number of nodes in the explanation graph $\mathcal{G}_C$, i.e., the number of statements to be placed in the `try-catch` block. As the number of nodes (statements) in $\mathcal{G}_C$ increases, the number of correct statements covered also increases, thus, accuracy increases.

```
1  int ret = -1;
2  try {
3     FileInputStream fin = new FileInputStream(path);
4     int length = fin.available();
5     byte[] buf = new byte[length];
6     fin.read(buf);
7     ret = loadFromBuffer(buf);
8     fin.close();
9  } catch (FileNotFoundException e) {
10       Log.e(TAG, "error:" + e);
11       e.printStackTrace();
12 } catch (IOException e) {
13       Log.e(TAG, "error:" + e);
14       e.printStackTrace();
15 }
16 return ret;
```

Highlighted statements with highest scores contain three crucial API method calls relevant to the exception types at lines 9 and 12.

Figure 8: Correct Exception Handling Suggestion by DeepEx

However, as the number of statements increases higher than 5, accuracy increases more slowly. In our dataset, the average size of a `try-catch` block is 5.9 statements. As seen, the accuracy as N_x=6 is 74%. That is, by pointing out 6 statements on average, DeepEx can correctly suggest 74% of the total number of statements in the dataset that need to be placed in `try-catch` blocks. That is, it points out correctly 4.5 out of 6 statements to be in a `try-catch` block. For the statements that do not need to be placed in a `try-catch` block, DeepEx predicts correctly with 63% accuracy (not shown).

Example. Figure 8 displays an example that DeepEx made correct suggestions. **First**, it made a correct suggestion on the need of a `try-catch` block for the code at lines 1–8, 16. **Second**, GNNExplainer pointed out that XBlock used all the statements at lines 3–8 for such correct prediction. As a consequence, DeepEx correctly suggests to place lines 3–8 into a `try-catch` block. Note that, it also correctly pointed out that lines 1 and 16 do not need to be inside the `try-catch` block. **Third**, GNNExplainer gives three statements at lines 3, 6, and 8 highest scores. We can see that those lines contain three crucial API method calls: 1) `FileInputStream`, 2) `read`, and 3) `close`. **Fourth**, those three lines have data and control dependencies, which could help the model learn the identities of the API elements via their names `FileInputStream`, `read`, and `close`, despite that the code snippet does not have the fully-qualified names for those API elements. This confirms the need of integrating *program dependencies* in our solution. **Finally**, DeepEx was also able to learn from the training corpus that those names refer to those API elements, which often correspond to the following exception types: 1) `FileInputStream` with `FileNotFoundException`, and 2) `FileInputStream.read` and `FileInputStream.close` with `IOException`.

8.3 Exception Type Recommendation (RQ3)

8.3.1 **Recall on Exception Type Recommendation**. Table 3 displays the *Recall* result on how well *DeepEx covers on the actual exception types in the* `catch` *clauses in the oracle.* The result is with respect to the instances (code snippets) in the dataset with different numbers K of exception types in the `catch` clauses: K = the number of exception types in a `catch` clause in the oracle = 1, 2, 3, 3+. For example, in the oracle, there are 98 instances with 2 exception types in a `catch` clause. DeepEx's predicted set correctly contains all 2 exception types for 41 instances (42%) ($\texttt{Hit-All}_{Rec}$). The predicted set contains *at least 1 out of 2* exception types for 75 instances (77%).

Table 3: Detailed Results on Different Numbers of Exception Types in a `catch` clause in Oracle Set (RQ3) (Recall)

# Ex. Types in `catch` clause in Oracle Set	Metrics	Accuracy
1 (1,385 instances)	Hit-1	918 (66%)
2 (98 instances)	Hit-1	75 (77%)
	Hit-2	41 (42%)
3 (18 instances)	Hit-1	12 (67%)
	Hit-2	5 (28%)
	Hit-3	3 (17%)
3+ (30 instances)	Hit-1	18 (60%)
	Hit-2	10 (33%)
	Hit-3	4 (13%)
	Hit-All_{Rec}	962 (63%)

The sum of the percentages in each row section is not 100% because Hit-1 contains Hit-2, which contains Hit-3, and so on. Hit-n: number of overlap between the predicted set and the oracle set is at least n exception types.

Table 4: Detailed Results on Different Numbers of Exception Types in a `catch` clause in Predicted Set (RQ3) (Precision)

# Ex. Types in `catch` clause in Predicted Set	Metrics	Accuracy
1 (1,154 instances)	Hit-1	442 (38%)
2 (294 instances)	Hit-1	90 (31%)
	Hit-2	61 (21%)
3 (83 instances)	Hit-1	27 (33%)
	Hit-2	23 (28%)
	Hit-3	6 (7%)
	Hit-All_{Prec}	509 (33%)

The sum of the percentages in each row section is not 100% because Hit-1 contains Hit-2, which contains Hit-3, and so on. Hit-n: number of overlap between the predicted set and the oracle set is at least n exception types.

There are 1,385 instances (90.5%) over 1,531 total instances with a single exception type in the `catch` clause in the oracle in the testing dataset. DeepEx covers correctly the exception type in 918 (66%) of the instances. There are 98 instances with two exception types in the oracle and it correctly suggests both types in 41 cases (42%).

Note that $\texttt{Hit-All}_{Rec}$ = Hit-n when n (the number of overlaps between the predicted and oracle sets) = K (the number of exception types). For K=1..3, which is a total of 1,501 instances (98.1%), **DeepEx covers all the exception types ($\texttt{Hit-All}_{Rec}$) in 962 instances (63%) in the testing dataset**. That is, developers do not have to search for other exception types in 63% of the cases. Thus, it achieves high $\texttt{Hit-All}_{Rec}$ in 98.1% of the entire dataset.

8.3.2 **Precision on Exception Type Recommendation**. Table 4 shows the *Precision* result, which is shown with respect to different numbers K of *predicted exception types* in the `catch` clauses. For example, as seen in Table 4, there are a total of 294 instances (code snippets) in which DeepEx predicted two exception types in a `try-catch` block. Among them, with `Hit-1`=31%, there are 90 instances in which at least one predicted exception type is correct. With `Hit-2`=21%, there are 61 instances in which both the predicted exception types are correct (some types might be missing). Producing the exact-matched sets (`Hit-All`) for all exception types when $K \geq 3$ is still challenging, however, those cases are only 1.9% in the entire dataset. We do not have the row for 3+ because in our

Table 5: Impact of Different Features on XBlock (RQ4)

	DeepEx w/o SOT	DeepEx w/o AST	DeepEx
Recall	0.69	0.71	**0.79**
Precision	0.64	0.57	**0.68**
F-score	0.66	0.63	**0.73**

SOT: Sequence of tokens; AST: Abstract Syntax Tree

Table 6: Impact of Different Features on XState (RQ4)

	DeepEx w/o SOT	DeepEx w/o AST	DeepEx
Accuracy	0.75	0.73	**0.74**

SOT: Sequence of tokens; AST: Abstract Syntax Tree

dataset, 98.1% of the cases have ≤3 exception types, thus, we set 3 as the limit of the number of exception types for XType.

Importantly, Hit-All$_{Prec}$ = Hit-n when n (the number of overlaps between the predicted and oracle sets) = k (the number of exception types predicted by DeepEx). For k=1..3, which is a total of 1,531 instances (98.1%), **DeepEx predicts correctly all the exception types (Hit-All$_{Prec}$) in 509 instances (33%)**. That is, in 509 instances (i.e., 33%), all the predicted exception types are actually the correct ones. Developers do not have to delete any exception types in a predicted set, thus, DeepEx can save their efforts.

8.4 Ablation Study (RQ4)

8.4.1 Impact of Different Features on Try-Catch Necessity Checking (XBlock). Table 5 displays the results when we removed the two key features in DeepEx and measured XBlock's performance. As seen, without the sequences of code (lexical tokens) for each statement, Recall, Precision, and F-score decrease 12.7%, 5.9%, and 9.6%, respectively. Without considering the AST structure, Recall, Precision, and F-score also decrease even further with 10.1%, 16.2%, and 13.7%, respectively. In other words, *the code structure has a higher contribution than the lexical values of code tokens.*

8.4.2 Impact of Different Features on Try-Catch Statement Detection (XState). We used six as the limit on the number of nodes in the explanation sub-graph in which DeepEx achieves 0.74 of Accuracy. Note: the average number of the statements in a `try-catch` block in our dataset is 5.9. As seen, the result in Table 6 is consistent with Table 5, as AST contributes slightly more than code sequences.

8.4.3 Impact of Different Features on Exception Type Recommendation (XType). Tables 7 and 8 display the results. As seen, the impact result is consistent with those in Tables 5 and 6. Thus, code structure has slightly higher impact than code sequence.

8.5 Exception-Related Bug Detection (RQ5)

Has seen in Table 9, DeepEx can be used to detect well real-world exception-related bugs in which a code snippet needs but did not have a `try-catch` block or miss some exceptions. In comparison, DeepEx improves relatively over FuzzyCatch [**?**] 14.8% in Precision and 9.8% in F-score. While the recall values between two models are almost the same, FuzzyCatch has lower precision. It tends to predict "Yes" (buggy) for all code snippets. That is because if there is

Table 7: Impact of Different Features on XType (Recall)

# ET in Try-Catch Block in Oracle	Metrics	DeepEx w/o SOT	DeepEx w/o AST	DeepEx
1 (1,385 instances)	Hit-1	833 (60%)	791 (57%)	918 (66%)
2 (98 instances)	Hit-1	69 (70%)	66 (67%)	75 (77%)
	Hit-2	39 (40%)	37 (38%)	41 (42%)
3 (18 instances)	Hit-1	10 (56%)	11 (61%)	12 (67%)
	Hit-2	4 (22%)	4 (22%)	5 (28%)
	Hit-3	2 (11%)	3 (17%)	3 (17%)
3+ (30 instances)	Hit-1	16 (53%)	15 (50%)	18 (60%)
	Hit-2	9 (30%)	8 (27%)	10 (33%)
	Hit-3	3 (10%)	2 (7%)	4 (13%)

ET:Exception types; SOT: Sequence of tokens; AST: Abstract Syntax Tree

Table 8: Impact of Different Features on XType (Precision)

# ET in Predicted Try-Catch Block	Metrics	DeepEx w/o SOT	DeepEx w/o AST	DeepEx
1 (1,154 instances)	Hit-1	357 (31%)	309 (27%)	442 (38%)
2 (294 instances)	Hit-1	83 (28%)	78 (27%)	90 (31%)
	Hit-2	57 (19%)	51 (17%)	61 (21%)
3 (83 instances)	Hit-1	24 (29%)	21 (25%)	27 (33%)
	Hit-2	19 (23%)	18 (22%)	23 (28%)
	Hit-3	6 (7%)	4 (5%)	6 (7%)

ET: Exception types; SOT: Sequence of tokens; AST: Abstract Syntax Tree

Table 9: Exception-Related Bug Detection (RQ5)

	FuzzyCatch Dataset	
	DeepEx	FuzzyCatch [**?**]
Recall	0.75	**0.76**
Precision	**0.62**	0.54
F-score	**0.68**	0.62

```
1  public void onCreate(Bundle state) {
2    requestWindowFeature(Window.FEATURE_NO_TITLE);
3    + try {
4      final WindowManager.LayoutParams attrs = getWindow().getAttributes();
5      final Class<?> cls = attrs.getClass();
6      final Field fld = cls.getField("buttonBrightness");
7      if (fld != null && "float".equals(fld.getType().toString())) {
8      fld.setFloat(attrs, 0);
9    + }
10   + } catch (NoSuchFieldException e) {
11   + } catch (IllegalAccessException e) {
12   + }...
13 }
```

Figure 9: Exception-related Bug #106 in FuzzyCatch dataset (missing `try-catch`) (detected by DeepEx)

an association score between *only* one API method call in the code snippet and one exception type higher than the threshold, it will decide that the snippet is buggy. Figure 9 shows a bug detected by DeepEx, and its fix (adding a `try-catch` block). All buggy code and fixes are available in FuzzyCatch's repository: ebrand.ly/ExDataset.

Limitations and Threats to Validity. First, DeepEx can not handle the code with multiple `try-catch` blocks. Second, it cannot generate new exception types that were not in the training corpus. Third, it does not support the generation of exception handling

code inside the body of `catch`. Finally, DeepEx needs training data, thus, does not work for a new library without any API usage yet.

Our solution is specifically for Java. Our collected data might not be representative. However, we use well-established open-source projects with well-known libraries.

9 RELATED WORK

The automated approaches to recommend exception handling can be classified into four categories as presented in Section 1. The closest work to DeepEx is the state-of-the-art *information retrieval* (IR) approaches [**?**], which provides more flexibility than the others. XRank [**?**] recommends a ranked list of API calls that might need exception handling and XHand [**?**] recommends exception handling code. Both leverages fuzzy set theory to compute the associations between API method calls and the exception types. This direction has three key limitations. First, one needs to pre-define a threshold for feature matching for the retrieval of API elements or exception types. Second, the IR techniques are not flexible as the ML approaches because they use the lexical values of API simple names. Thus, they suffer the ambiguity in the names of API elements in incomplete code snippets. Lastly, XRank/XHand considers only pairwise associations between the API method calls and exceptions. It disregards the surrounding code context and the dependencies/relations. XRank/XHand simply uses Groum [**?**], a dependency graph among API elements, to collect the API calls, but did not use dependencies in computing the association scores.

In addition to exception handling recommendation research, ThEx [**?**] predict which exception(s) shall be thrown under a given programming context. ThEx learns a classification model from existing thrown exceptions in different contexts.

10 CONCLUSION

DeepEx is the first neural-network model to automated exception handling recommendation in three tasks for (in)complete code. It is designed to capture the basic insights to overcome key limitations of the state-of-the-art IR approaches. With the learning-based approach, it does not rely on a pre-defined threshold for explicit feature matching. The dependencies and context help DeepEx learn the identities of API elements to avoid name ambiguity and to learn their relations with the exception types. Our evaluation shows that DeepEx improves over the state-of-the-art approaches in both intrinsic task and extrinsic one in exception-related bug detection.