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 NEUREX, 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 NEUREX 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 NEUREX 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 NEUREX 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), NEUREX correctly recommend exception handling in all three tasks. Our extrinsic evaluation also shows that with its recommendations, NEUREX 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 NEUREX, 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 NEUREX 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 NEUREX 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. NEUREX 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 NEUREX three advantages over the state-of-the-art IR approach. First, with the learning-based approach, NEUREX 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, NEUREX has advantages in both predicting and training. During predicting, for a given incomplete code, the context enables NEUREX 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 NEUREX. 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 NEUREX 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), NEUREX correctly recommend exception handling in all three tasks. Our extrinsic evaluation also shows that with its recommendations, NEUREX 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]. NEUREX is the first neural network approach to automated exception handling recommendation in three above tasks. NEUREX 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 NEUREX'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`.

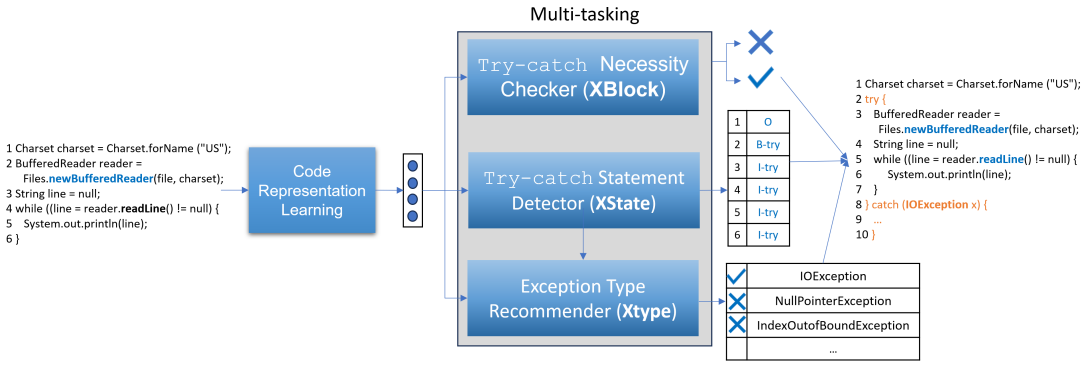

Figure 5: NEUREX: Architecture Overview

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 NEUREX 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 NEUREX 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, NEUREX 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 NEUREX OVERVIEW

Figure 5 displays the overview architecture of NEUREX. 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, NEUREX 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

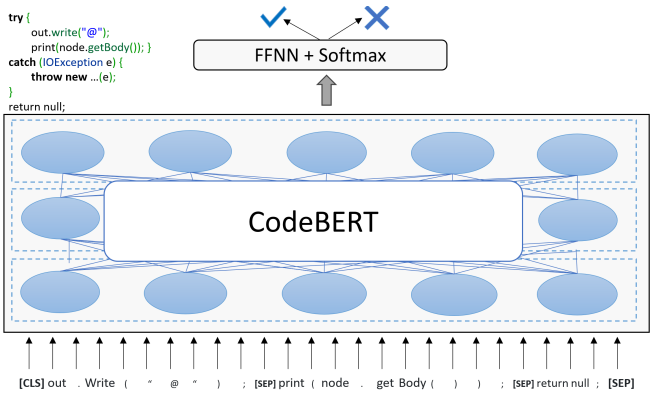

Figure 6: Try-catch Necessity Checker (XBlock)

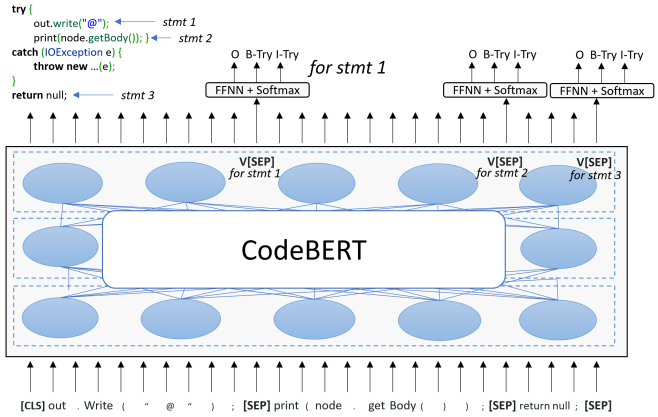

Figure 7: Try-catch Statement Detector (XState)

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, Neurex forms the final output.

4 XBLOCK: TRY-CATCH NECESSITY CHECKER

Given an input code snippet, we first split it into the statements. Each statement is then tokenized into sub-tokens using the CodeBERT tokenizer. We use a special separator token `[SEP]` to concatenate the tokenized statements, 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 6) 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

The goal of XState (Figure 7) is to decide if a statement in the given code snippet needs to be in a `try-catch` block or not. For code representation learning, we use one single CodeBERT [**?**] model as in XBlock (see Figure 6) to produce the embedding for code (sub)-tokens. For the input code, each `[SEP]` token represents the statement preceding it. Note that semicolon would not be as consistent as our explicit separator, because some statements might not end with a semicolon. A semicolon also could appear inside string literals and for-loop conditions.

The advantage in using CodeBERT on the entire code snippet has two folds. First, as in Key idea 2, the context of the entire code facilitates our model to learn the identities of the API elements, thus, making the connections between the APIs and the presence of `try-catch` blocks. For example, `write` in the statement 1 could be determined to belong to `OutputStream` in JDK. Thus, that leads to better learning to place that statement inside a `try-catch` block. Second, we expect that CodeBERT would learn *the data and control dependencies among the consecutive statements* separated by the special tokens **[SEP]**. Those dependencies would help the model better decide whether some consecutive statements need to be placed together in a `try-catch` block. For example, in Figure 5, the statements at lines 2–5 have data dependencies, thus, they should be in the same `try-catch` block.

We take the embedding produced by CodeBERT for each **[SEP]** token as the statement embedding and feed it to a layer with Feed-forward neural network (FFNN). We then use a softmax function to learn to label each corresponding statement with one of three tags: `O` tag means that the statement is outside of any `try-catch` blocks; `B-Try` tag means that the statement begins a `try-catch` block; and `I-Try` tag means that the statement is inside a `try-catch` block and is not the first line of that block. For example, in Figure 7, the embedding computed by CodeBERT for the first **[SEP]**, which represents the

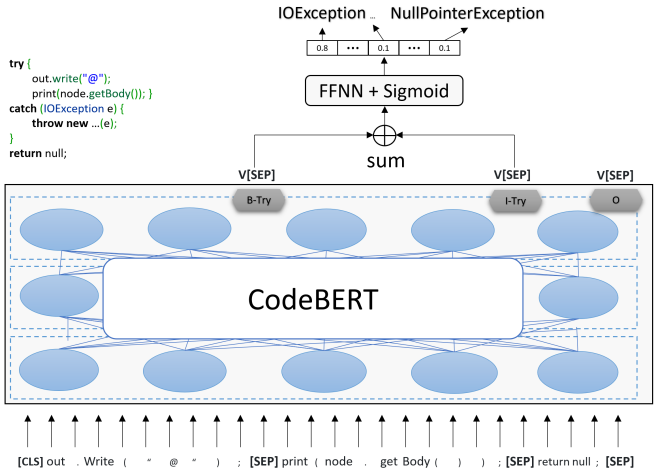

Figure 8: Exception Type Recommendation (XType)

statement `out.write(...)`, is classified by the first `FFNN+Softmax` layer into the `B-Try` category/label because it is the first statement of a `try-catch` block. However, the embedding for the second **[SEP]**, representing `print(node.getBody(...))`, is labeled as `I-Try` because it is the second statement of the `try-catch` block. Finally, the embedding of the third **[SEP]**, is labeled as `O` because the statement `return null;` does not need to be in the `try-catch` block. With this tag encoding, we can support multiple `try-catch` blocks in which the statement with the `B-Try` tag is the start of a `try-catch` block and the statement with the last respective `I-Try` tag is the end of that block. A new block will be formed with another statement having a `B-Try` tag.

During training, the labels for all the statements are known from the code. During prediction, our model will assign the labels to the statements, from which we can derive the `try-catch` blocks and what statements belonging to each block.

6 XTYPE: EXCEPTION TYPE RECOMMENDER

The goal of XType (Figure 8) is to predict what exception types need to be placed in the `catch` clause of each of the predicted `try-catch` block(s) for the given input code snippet. We use one single CodeBERT [**?**] model as in XBlock to produce the embedding for code (sub)-tokens. We expect CodeBERT to learn the connection between the statements in a `try-catch` block and the corresponding exception types that need to be caught. The rationale is from the Key idea 2 in which we expect that via context, CodeBERT could learn the identities of the API elements, thus, leading to better learning of the exception types to be caught.

During training, we know all the labels of the statements in a code snippet. For each `try-catch` block, we identify the statements at the beginning (with the `B-Try` label) and at the end of a `try-catch` block (with the last respective `I-Try` label). We consider the embeddings produced by CodeBERT for the **[SEP]** tokens corresponding to the statements from the beginning to the end of the `try-catch` block. We add them together to get the embedding for the entire `try-catch` block, and then feed it into a linear layer. We use a sigmoid function to perform binary classifications for the exception types in the libraries of interest.

During prediction, we use the predicted tags for the given statements in the code snippet. From the predicted tags, we obtain the the statements in a predicted `try-catch` block. From there, the embeddings computed by CodeBERT are retrieved and used in the same way as during training. For example, in Figure 8, the model predicts the `try-catch` block from the statement `out.write` to the statement `print(node.getBody(...))`. The corresponding embeddings of all the statements in the block are used and the output of `IOException` is predicted with the highest probability.

7 MULTI-TASK LEARNING

The learning on the three tasks, XBlock, XState, and XType can benefit to one another. For example, if a model decides the need of a `try-catch` block, there must be some statements in the code snippet that will be placed in such a block. If a model learns the statements to be placed in a `try-catch` block, it can decide that the code snippet needs such a block and make the connections to what exception types to be caught. The knowledge on the exception types can help a model decide better what important statements need to be in a `try-catch` block. Therefore, we put the three tasks in a multi-task learning fashion.

We calculate the training loss by combining the losses from the three modules. $Loss_{\text{XBlock}}$ is the Binary Cross Entropy loss for the decision as to whether try-catch blocks exist in the input. To calculate $Loss_{\text{XState}}$, we first get the classification loss for each statement ($loss_{Stmt}$) in the input, and sum them together. $loss_{Stmt}$ is the Cross Entropy loss calculated from the distribution for the three tags – `O`, `B-Try`, `I-Try` – and the ground-truth tag. Finally, in XType, several `try-catch` blocks might be present, so $Loss_{\text{XType}}$ comes from the summation of the exception prediction losses from all the `try-catch` blocks. For each `try-catch` block, the $loss_{try-block}$ is calculated by adding the Binary Cross Entropy loss for the prediction of each exception that we considered.

The overall training loss is calculated as follows. If the input does not contain any `try-catch` block, the loss will be only the $Loss_{\text{XBlock}}$. If the input contains a `try-catch` block, the overall loss will be the summation of the losses from all three tasks (1).

$$Loss_{overall} = \begin{cases} Loss_{\text{XBlock}}, & \text{no try-catch} \\ Loss_{\text{XBlock}} + Loss_{\text{XState}} + Loss_{\text{XType}}, & \text{otherwise.} \end{cases} \quad (1)$$

8 EMPIRICAL EVALUATION

8.1 Research Questions

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

RQ$_1$. [Effectiveness on Try-Catch Necessity Checking] *How accurate is Neurex 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 Neurex 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 Neurex 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* NEUREX *affect its performance?*

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

8.2 Empirical Methodology

8.2.1 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, which was not part of any fix in a later version. In total, we have 153,823 code snippets containing `try-catch` blocks 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 750 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.

8.2.2 RQ1. Effectiveness on Try-Catch Necessity Checking . Baselines. We compared XBLOCK against the pre-trained CodeBERT without any fine-tuning steps. In the pre-trained CodeBERT, We add a randomly-initialized linear layer on top of the output vector of the `[CLS]` token, and use a softmax function to learn the decision. We also 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. Meanwhile, we make sure that each partition contains the equal amount of positive samples and negative samples; and the training and tuning partitions do not contain any duplicates.

Tuning. We trained NEUREX for 15 epochs with the following key hyper-parameters: (1) Batch size is set to 32; (2) Learning rate is set to 0.000006; (3) Weight decay is set to 0.01. We select the model with the lowest overall loss.

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.

8.2.3 RQ2. Effectiveness on Try-Catch Statement Detection . Baselines. We compared XSTATE against the pre-trained CodeBERT model without any fine-tuning step as in RQ1.

Procedure. We used the same procedure as in RQ1.

Tuning. We used the same tuning as in RQ1.

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

	Precision	Recall	F1-score
CodeBERT w/o fine-tuning	0.4969	**0.9719**	0.6576
XRank			
NEUREX	**0.9805**	0.9842	**0.9824**

Metrics. We used the same metrics as in RQ1. However, we computed Recall, Precision, and F-score in two ways. First, we evaluated XSTATE in connection with XBLOCK. That is, we consider a correct case if XBLOCK gives a correct prediction on whether `try-catch` blocks are needed, *and* XSTATE produces correct tags for all the statements when there exists a `try-catch` block. Second, we evaluated XSTATE as individual. That is, we assumed that XBLOCK predicts correctly whether a code snippet needs `try-catch` blocks or not. Thus, we put the code snippets that need such blocks in the oracle and used XSTATE to predict the groupings of statements for the blocks in those snippets.

8.2.4 RQ3. Effectiveness on Exception Type Recommendation . Baselines. We compared XSTATE against the pre-trained CodeBERT model without any fine-tuning step as in RQ1.

Procedure. We used the same procedure as in RQ1.

Tuning. We used the same tuning as in RQ1.

Metrics. We used the same metrics as in RQ1. However, we computed Recall, Precision, and F-score in two ways. First, we evaluate XTYPE in connection with XBLOCK and XSTATE. That is, we consider a correct case if XBLOCK gives a correct prediction on whether `try-catch` blocks are needed, XSTATE produces correct tags for all the statements, and XTYPE predicted correctly on the exceptions that need to be caught for all the `try-catch` blocks. Second, we evaluate XTYPE as individual. That is, we assumed that both XBLOCK and XSTATE have given 100% correct prediction results for their own tasks. Thus, in this second way, we put the code snippets that need `try-catch` blocks with statements correctly labelled in the oracle and used XTYPE to predict the exception types for all such blocks.

8.2.5 RQ4.

8.2.6 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 NEUREX on the Github dataset and detected the exception-related bugs in FuzzyCatch bug dataset via XBLOCK.

Metrics. We compared the result against the oracle in FuzzyCatch dataset. If XBLOCK correctly detects a 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.

9 EMPIRICAL RESULTS

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

As seen in Table 1, NEUREX achieves very high Precision, Recall and F-score on the Github Dataset—all above 98%. In comparison, the CodeBERT baseline model has a much lower Precision, around 50%. However, it achieves a slightly higher Recall. After examining the result, we find that the model overwhelmingly predicts that the

```
protected Class << ? > loadClass(String name, boolean resolve) throws ClassNotFoundException {
  if (name.startsWith(Test.class.getName())) {  0.01
    Class << ? > c = findLoadedClass(name);  0.04
    if (c != null) {  - 0.008
      return c;  - 0.052
    }  - 0.022
    try {
      COUNTER++;  0.012
      InputStream in = getSystemResourceAsStream(name.replace('.', File.separatorChar) + ".class");  0.329
      byte[] buf = in .readAllBytes();  0.076
      return defineClass(name, buf, 0, buf.length);  6.94e-05
    } catch (IOException e) {
      throw new ClassNotFoundException(name);
    }
  }  0.029
  return super.loadClass(name, resolve);  - 0.011
}
```

Figure 9: XBlock Case Study

Table 2: Try-Catch Necessity Checking Evaluated On Test Partitions by The Number Of Try-Catch Blocks (RQ1)

	Number of Try-Catch Blocks (Github Dataset)					
	Zero	One	Two	Three	Four	Five
Precision	0.0	1.0	1.0	1.0	1.0	1.0
Recall	0.0	0.983	0.9988	1.0	0.9861	1.0
F1	0.0	0.9914	0.9994	1.0	0.993	1.0

input code snippet contains a `try-catch` block: In our balanced test dataset that contains 30,764 samples, only 236 samples receives the negative label (i.e., no `try-catch`) from CodeBert.

Attribution Scores. To illustrate how XBlock makes the prediction, in Figure 9, we shows a code snippet that catches an `IOException` thrown by the `readAllBytes` API call on an `InputStream` object. CodeBERT produces as a by-product an *attribution score* for each code sub-token in the input. The higher the score of a token the higher attention that the model pays to that token, contributing to the prediction result. In Figure 9, for each statement, we show the statement attribution score, which is calculated by averaging the attribution scores of all the sub-tokens in the statement. A positive attribution score means that the statement contributes positively to the model's predicted class, while a negative score means the statement contributes negatively to the predicted class. As seen, the two statements that receive the highest scores are the statement that defines the `InputStream` variable and the statement that invokes the `readAllBytes` method call on the `InputStream` object. This example illustrates that the model is able to put the attention on the right (sub)tokens of those statements including `InputStream`, `in`, `get`, `System`, `name`, `replace`, etc., leading to its correct prediction.

In addition, we partitioned the test dataset according to the number of `try-catch` blocks, and evaluated XBlock on each partition. As seen in Table 2, XBlock gives 100% correct prediction on the partitions with zero, three and five `try-catch` blocks. Moreover, it achieves 100% precision and above 0.99 F1-score across all the partitions, showing that XBlock's prediction ability remains strong regardless of the number of `try-catch` blocks in a code snippet.

9.2 Try-Catch Statement Detection (RQ2)

As Table 3 shows, when evaluating XState in connection with XBlock, XState achieves high precision score and is able to recover statements in try-catch blocks for about 60% of the positive code samples, while the codebert baseline model fails the task completely.

Table 3: Try-Catch Statement Detecting Comparison (Evaluate XState in Connection with XBlock — Instance Level) (RQ2)

	Precision	Recall	F1-score
CodeBERT w/o fine-tuning	0.0	0.0	0.0
XState	**0.9688**	**0.6072**	**0.7465**

Table 4: Try-Catch Statement Detecting Comparison (Evaluate XState as Individual — Instance Level) (RQ2)

	Precision	Recall	F1-score
CodeBERT w/o fine-tuning	0.0	0.0	0.0
XState	**1.0**	**0.6198**	**0.7652**

Table 5: Try-Catch Statement Detecting Comparison (Evaluate XState As Individual — Block Level) (RQ2)

	Precision	Recall	F1-score
CodeBERT w/o fine-tuning	0.0	0.0	0.0
XState	**0.4**	**0.6369**	**0.4914**

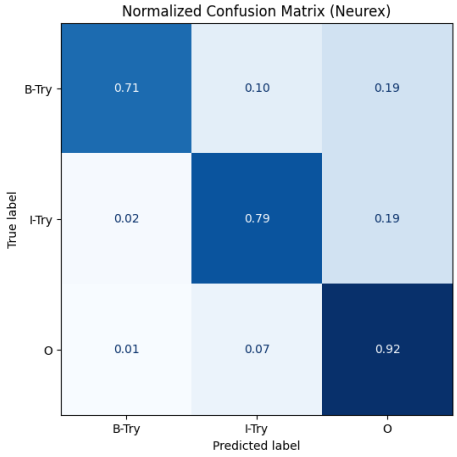

Figure 10: Normalized Confusion Matrix — Neurex (XState Evaluated As Individual) (RQ2)

Furthermore, we present results of evaluating XState as individual in Table 4. As can be seen, the codebert baseline still cannot give any correct predictions. In comparison, XState manages to achieve a 100% precision, showing that XState is capable of giving correct predictions for those false negatives from XBlock.

We also evaluate XState as individual at the try-catch block level (see Figure 5). The Codebert baseline still produce no correct prediction, while XState achieves 40% of precision and about 63% of recall.

To further understand their decision making, we show the confusion matrices for both XState and the Codebert baseline model. In Figure 10, we see XState has high confidence in predicting statements that are outside try blocks; however, it more often conflates try-block statements with non-try-block statements. Figure 11 explains why the Codebert baseline model fails the task: It assign the `I-Try` label to all the statements. And without the `B-Try` label, `I-Try` can never be interpreted as a valid try block.

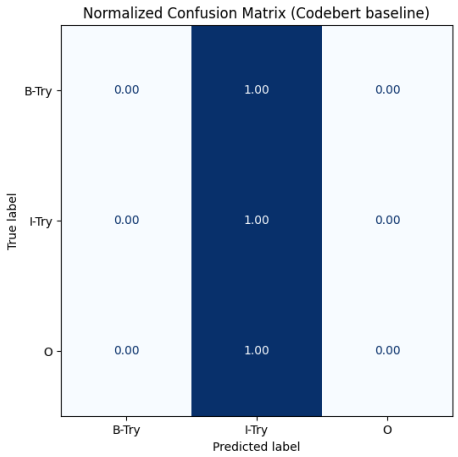

Figure 11: Normalized Confusion Matrix — CodeBERT baseline (XState Evaluated As Individual) (RQ2)

9.3 Exception Type Recommendation (RQ3)

9.3.1 **Recall on Exception Type Recommendation**. Table **??** displays the *Recall* result on how well *NEUREX 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. NEUREX'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%).

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. NEUREX 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%), **NEUREX 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.

9.3.2 **Precision on Exception Type Recommendation**. Table **??** 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 **??**, there are a total of 294 instances (code snippets) in which NEUREX 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 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, $\texttt{Hit-All}_{Prec}$ = `Hit-`n when n (the number of overlaps between the predicted and oracle sets) = k (the number of exception types predicted by NEUREX). For k=1..3, which is a total of 1,531 instances (98.1%), **NEUREX predicts correctly all the exception types ($\texttt{Hit-All}_{Prec}$) 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, NEUREX can save their efforts.

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

	NEUREX w/o SOT	NEUREX w/o AST	NEUREX
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 7: Impact of Different Features on XSTATE (RQ4)

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

SOT: Sequence of tokens; AST: Abstract Syntax Tree

9.4 Ablation Study (RQ4)

9.4.1 Impact of Different Features on Try-Catch Necessity Checking (XBLOCK). Table 6 displays the results when we removed the two key features in NEUREX 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.*

9.4.2 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 NEUREX 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 7 is consistent with Table 6, as AST contributes slightly more than code sequences.

9.4.3 Impact of Different Features on Exception Type Recommendation (XTYPE). Tables 8 and 9 display the results. As seen, the impact result is consistent with those in Tables 6 and 7. Thus, code structure has slightly higher impact than code sequence.

9.5 Exception-Related Bug Detection (RQ5)

Has seen in Table 10, NEUREX 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, NEUREX 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 8: Impact of Different Features on XType (Recall)

# ET in Try-Catch Block in Oracle	Metrics	Neurex w/o SOT	Neurex w/o AST	Neurex
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 9: Impact of Different Features on XType (Precision)

# ET in Predicted Try-Catch Block	Metrics	Neurex w/o SOT	Neurex w/o AST	Neurex
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 10: Exception-Related Bug Detection (RQ5)

	FuzzyCatch Dataset	
	Neurex	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 12: Exception-related Bug #106 in FuzzyCatch dataset (missing try-catch) (detected by Neurex)

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 12 shows a bug detected by Neurex, 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, Neurex 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, Neurex 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.

10 RELATED WORK

The automated approaches to recommend exception handling can be classified into four categories as presented in Section 1. The closest work to Neurex 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.

11 CONCLUSION

Neurex 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 Neurex learn the identities of API elements to avoid name ambiguity and to learn their relations with the exception types. Our evaluation shows that Neurex improves over the state-of-the-art approaches in both intrinsic task and extrinsic one in exception-related bug detection.