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 shows that NEUREX correctly performs all three exception handling recommendation tasks in 71.5% of the cases with an F1-score of 70.2%. It improves relatively 166% over the baseline. It achieves high F1-score from 98.2%–99.7% in `try-catch` block necessity checking (an relative improvement of up to 55.9% over the baselines). It also correctly decides both the need of `try-catch` block(s) and the statements to be placed in such blocks with the accuracies of 74.7% and 87.1% at the instance and statement levels, an improvement of 128.7% and 44.9% over the baseline, respectively. Our extrinsic evaluation shows that NEUREX relatively improves over the baseline by **9.6%** in F1-score in detecting exception-related bugs in incomplete StackOverflow code snippets.

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 *program analysis 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 due to incomplete code. The second category utilized *exception handling policies*, which are enforced in all cases [? ?]. However, the policies need to be predefined 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 **dependencies** *among statments with APIs* 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 30,764 code snippets (half of them have `try-catch` blocks). Our empirical evaluation shows that Neurex correctly performs all three exception handling recommendation tasks in 71.5% of the cases with an F1-score of 70.2%. It has an relative improvement of 166% over the baselines. It achieves high F1-score from 98.2%–99.7% in `try-catch` block necessity checking (an relative improvement of up to 55.9% over the baselines). It also correctly decides both the need of `try-catch` block(s) and the statements to be placed in such blocks with the accuracies of 74.7% and 87.1% at the instance and statement levels, an improvement of 128.7% and 44.9% over the baseline, respectively. Our extrinsic evaluation shows that Neurex relatively improves over the baseline by **9.6%** in F-score in detecting exception-related bugs in incomplete StackOverflow code snippets.

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.

```
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

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.

3. [Empirical Evaluation]. Our 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. Thus, it is desirable to have an automated tool to recommend proper exception handling for such adaptation.

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 [? ? ? ? ?]. However, the state-of-the-art, IR-based approaches [?], which have been shown to outperform others, still have the limitations. First, it is not trivial to pre-define a threshold for feature

```
1  /** ...
2   * taken from http://stackoverflow.com/questions/15409223/
3   * adding-new-paths-for-native-libraries-at-runtime-in-java
4   */
5  private static void addLibraryPath(String pathToAdd) {
6    try {
7      final Field usrPathsField =
             ClassLoader.class.getDeclaredField("usr_paths");
8      usrPathsField.setAccessible(true);
9
10     // get array of paths
11     final String[] paths = (String[]) usrPathsField.get(null);
12
13     // check if the path to add is already present
14     for (String path : paths) {
15       if (path.equals(pathToAdd)) {
16     return;
17       }
18     }
19
20     // add the new path
21     final String[] newPaths = Arrays.copyOf(paths, paths.length + 1);
22     newPaths[newPaths.length - 1] = pathToAdd;
23     usrPathsField.set(null, newPaths);
24   } catch (NoSuchFieldException | SecurityException |
         IllegalArgumentException | IllegalAccessException e) {
25     throw new RuntimeException(e);
26   }
27 }
```

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

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 their exceptions via only simple names.

```
1  public Object readField(Class<?> clazz, String name, Object instance) {
2    try {
3      Field field = clazz.getDeclaredField(name);
4      if (!field.isAccessible()) {
5         field.setAccessible(true);
6      }
7      return field.get(instance);
8    } catch (NoSuchFieldException | SecurityException |
         IllegalArgumentException | IllegalAccessException e) {
9     throw new RuntimeException("Cannot read field value: " + clazz.getName()
            + "#" + name, e);
10   }
11 }
```

Figure 3: Project `quarkus` with same exception handling

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

```
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

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, a model could learn from the public repositories with complete code to suggest proper exception handling.

Observation 2 (**Regularity of Exception Handling**). *Finding the patterns from complete code in existing code corpora could be a good strategy for a model 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, leading to better prediction of the handled exceptions.*

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.

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`. 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

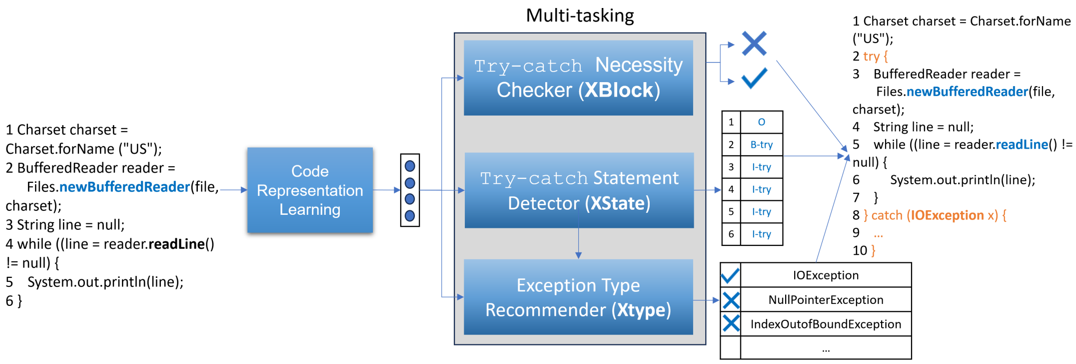

Figure 5: NEUREX: Architecture Overview

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 three functionalities 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 the exception types to be in the catch clause (XTYPE).

From Observations, we design NEUREX with the following 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 follow a learning-based approach 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 model can help the adaptation tasks.

2.2.2 [Key Idea 2] Leveraging Language Model to learn 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 implicitly (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 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 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.

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 in complete code are used as the positive samples and the ones without them as the negative ones. For a positive sample, the statements inside the try-catch 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 (in)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 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 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.

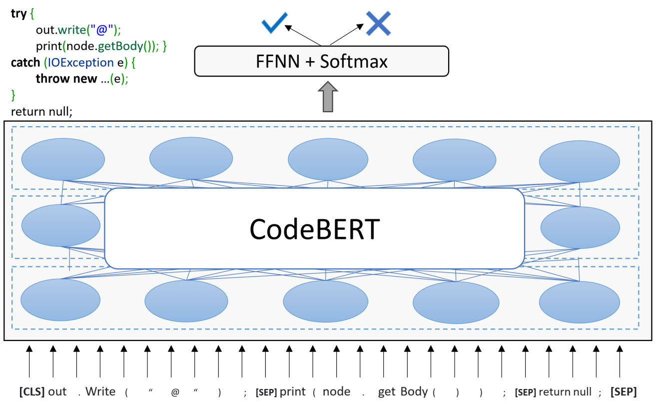

Figure 6: `Try-catch` Necessity Checker (XBlock)

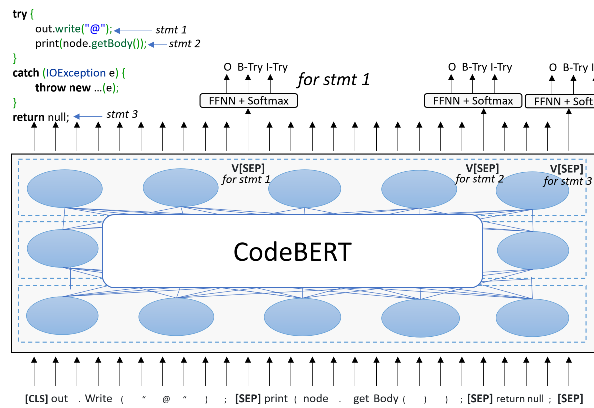

Figure 7: `Try-catch` Statement Detector (XState)

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. A *Yes* outcome indicates the need to catch a specific exception type of interest in the set of libraries under consideration. A *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 BLOCK 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. We expect it to be able to learn the dependencies among statements with API elements 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 dependencies among the consecutive statements* separated by the special tokens **[SEP]** (see Section 9.4 for our experiment on this). 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 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.

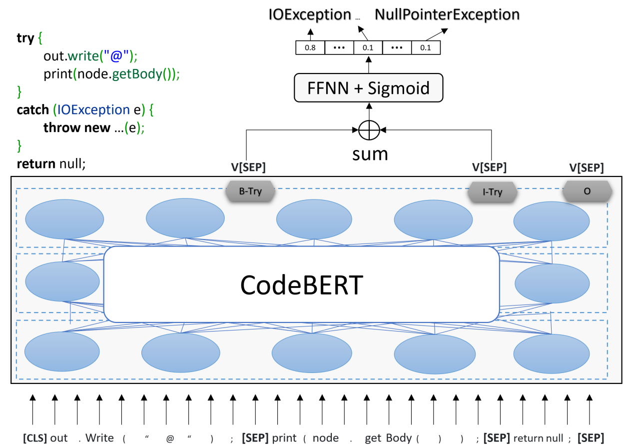

Figure 8: Exception Type Recommendation (XType)

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. From Key idea 2, we expect that via context, CodeBERT could learn the identities of the API elements, 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 the block (with the last respective `I-Try` label). We consider the embeddings from CodeBERT for the **[SEP]** tokens corresponding to the statements from the beginning to the end of the block. We add them together to get the embedding for the entire `try-catch` block, and 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 used in the same way as in 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 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

Learning on the three tasks, XBlock, XState, and XType can benefit to one another. 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. Thus, 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 add 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 $Loss_{\text{XBlock}}$. If the input contains a `try-catch` block, the overall loss will be the summation of the losses from all three tasks:

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

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$_4$. [Dependency Probing]. *How well Neurex learn the dependencies among statements for grouping them into a* `try-catch` *block?*

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

RQ$_6$. [Ablation Study]. *How does fine-tuning improve Neurex?*

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 15,382 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. In total, we have 30,764 instanaces.

For extrinsic evaluation, we used FuzzyCatch dataset, provided by the authors of XRank/XHand [**?**], which contains 750 StackOverflow code snippets with exception-related bugs (missing `try-catch` blocks or missing catching some excxeptions). We also randomly selected from the projects in FuzzyCatch dataset the same amount of snippets with no exception-related bugs as the negative samples.

8.2.2 RQ1. Effectiveness on Try-Catch Necessity Checking.

Baselines. We compared XBlock with GPT-3.5 in ChatGPT [**?**]. Due to cost of using ChatGPT-3.5 on OpenAI, we performed sampling on the Github dataset of 30,764 snippets. To obtain the confidence level of 95%, we randomly selected 380 code snippets in which 190 are negative samples (no `try-catch` block), and 190 are positive samples (at least one `try-catch` block). We trained on Github dataset and compared with GPT-3.5 on this sampled test set.

We also compared XBlock with XRank [**?**] (XRank is part of FuzzyCatch) on the Github dataset. XRank computed the exception risk score for each API call. If a score of a call in the snippet is higher than a threshold, we consider it as needing a `try-catch` block.

Procedure. We randomly split both the positive and negative sets in a dataset 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.

To request responses from ChatGPT, we construct prompt with the format "question + code", where the question we pose is "Does the code below need to catch any exceptions?\n\n". Considering that the answers from ChatGPT for the same prompt may vary, for each code snippet, we send the prompt three times through the Chat Completions API. Labeling the responses has three steps. First, we check whether the first word in each response is Yes or No. If it is a Yes, we assign a positive label; If it is a No, we assign a negative label. Second, for the responses that do not start with Yes or No, we read each response and manually assign labels to them. However, there are some cases that we cannot, based on the response, easily determine whether or not the code needs any try-catch blocks. The common reasons are (1) the response is not informative enough in that it only relays some general advice about exception handling, (2) In the text, it states that it is uncertain about the case, or (3) the decision depends on background knowledge about either the project structure or certain method specifications. Thus, in evaluating the performance of ChatGPT, for each metric, we calculate the lower and upper bound. In the upper bound, all such uncertain cases are considered as correct, in the lower bound, all such cases are considered as incorrect. Lastly, we assign the final label for each instance from the majority vote from three tries.

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 follows. $Recall = \frac{TP}{TP+FN}$, $Precision = \frac{TP}{TP+FP}$, F-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 GPT-3.5 as in RQ1.

Tuning. We used the same tuning as in RQ1.

Procedure and Metrics. We evaluated the models at both the instance level and the statement level. At the instance level, a prediction for a code snippet is considered as correct if all the statements inside or outside of all the predicted (zero or multiple) `try-catch` blocks must match with the grouping of the corresponding statements in the corresponding (zero or multiple) `try-catch` blocks in the oracle. To do so, we match the encoded vector [O, B-Try, I-Try] of the prediction against the vector for a snippet in the oracle. Because we have both positive/negative instances, we used the same metrics Precision, Recall, and F1-score as in RQ1. At the statement level, we evaluated if a model predicts correctly whether a statement needs to be inside a `try-catch` block, regardless of the blocks themselves. Thus, we use *Accuracy* for the statement-level evaluation, which is defined as the ratio between the number of correct taggings of statements over the total number of statements. Importantly, we also evaluated Neurex in two ways. First, we evaluated XState in connection with XBlock. That is, we consider a case as correct if both XBlock and XState give correct predictions (correct on the need of a `try-catch` block and correct on statement tagging). Second, we also evaluated XState as individual. We assume that XBlock predicted correctly on the positive instances. We evaluated XState individually at both instance and statement levels as explained.

8.2.4 RQ3. Effectiveness on Exception Type Recommendation.

Baselines. We compared XType against GPT-3.5 as in RQ1.

Tuning. We used the same tuning as in RQ1.

Procedure and Metrics. For a predicted set of exception types, we used 1) Precision (defined as the ratio between the size of the overlapping set between the predicted and oracle sets over the size of the predicted one), 2) Recall (defined as the ratio between the size of the overlapping set and the size of the oracle set), and 3) F-score (harmonious mean of Precision and Recall). We evaluated Neurex in two ways. First, we evaluate XType in connection with XBlock and XState. We consider the cases where all three parts are correct. We also computed those metrics in the cases where XBlock and XState are correct at the instance level. Second, we evaluate XType as individual: we evaluated XType with those metrics in the cases in which XBlock and XState are correct at the instance level.

8.2.5 RQ4. Dependency Probing.

In this research question, we examine whether Neurex puts more attention weights for statements inside try blocks. The evaluation is done on the correctly predicted portion of the Github dataset. For each input code snippet, we feed it through the Transformer body of Neurex, and extract the attention weight matrix from the last layer. We sum up the attention weights between statements inside try blocks, and sum up the attention weights from statements inside to statements outside try blocks (recall that statements are represented by [SEP] tokens). Therefore, for all instances, we have a list of attention scores for inside statements (see Formula 2), and a list of attention scores for inside to outside statements (see

Table 1: Try-Catch Block Comparison with XRank (RQ1)

Github dataset	Precision	Recall	F1-score
XRank	0.810	0.530	0.630
NEUREX	**0.981**	**0.984**	**0.982**

Table 2: Try-Catch Block Comparison with GPT-3.5 (RQ1)

Small dataset	Precision	Recall	F1-score
GPT-3.5	0.666–0.804	0.726–0.778	0.695–0.791
NEUREX	**0.994**	**1.0**	**0.997**

Formula 3). Through two sample t-test, we analyze whether means of the two lists ($\overline{X}_{in}$ and $\overline{Y}_{in \to out}$) are different and report the p-value. If *p-value* < 0.05, we reject the Null Hypothesis—there is no difference between the means.

$$X_{in} = [S^1_{attn,in}, S^2_{attn,in}, ..., S^n_{attn,in}] \quad (2)$$

$$Y_{in \to out} = [S^1_{attn,in \to out}, S^2_{attn,in \to out}, ..., S^n_{attn,in \to out}] \quad (3)$$

8.2.6 RQ5. Extrinsic Evaluation on Exception-related Bug Detection. *Baselines.* We compared with FuzzyCatch [?] in exception-related bug detection, i.e., missing `try-catch` blocks and/or exceptions.

Procedure. We trained NEUREX on the Github dataset and detected the exception-related bugs in FuzzyCatch dataset of StackOverflow code snippets. If the exception handling in a snippet matches with the one suggested by NEUREX, we consider it as a correct detection.

Metrics. We use **Recall**, **Precision**, and **F-score** as in RQ1.

8.2.7 RQ6. Ablation Study.
We aim to evaluate the contribution of fine-tuning in NEUREX. We compared NEUREX against CodeBERT without fine-tuning.

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, NEUREX relatively improves over XRank **21%, 85.7%, and 55.9%** in Precision, Recall, and F1-score, respectively.

Examining the result, we reported the following on XRank. First, its precision is marginally better than a coin toss (0.53) in our balanced dataset. In XRank, if the association score of *only one API method* in the snippet and *one exception* is higher than a threshold, it decides that a `try-catch` block is needed. 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, leading to mistakenly considering them the same. Unlike XRank, which considers only the API calls in a `try-catch` block, NEUREX considers the code in the block as the context to learn better the identities of APIs and dependencies with exceptions, thus, better deciding the need of `try-catch` blocks.

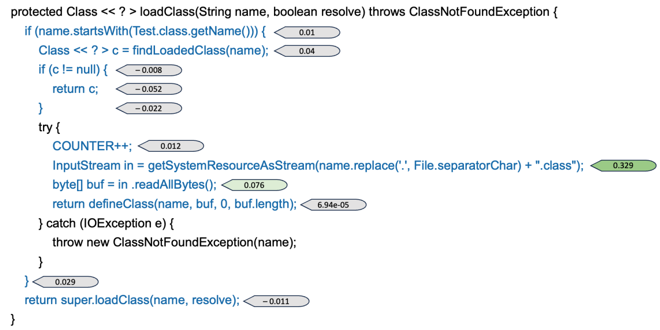

Figure 9: XBLOCK Puts Attention on the Right Tokens

Table 3: 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-score	0.0	0.9914	0.9994	1.0	0.993	1.0

As seen in Table 2, NEUREX relatively improves over GPT-3.5 from **23.6%–49.3%**, **28.5%–37.7%**, and **26%–43.5%**, in Precision, Recall, and F1-score, respectively. Examining GPT-3.5's results, we found that it detected well only the popular APIs and corresponding exception types because it was not trained specifically for the exception handling task. Moreover, for the un-popular API names, GPT-3.5 often resorted to another API with similar name, and predicted that the given code snippet needs a `try-catch` block because that API requires such a block. For example, in an instance containing a method call to `interval.parseWithOffset`, which is specific to a project, GPT-3.5 incorrectly considered it as to have a `try-catch` block. GPT-3.5 explained that it is similar to `parse` in a compiler, which needs to handle `InvalidInputException`. Thus, it incorrectly considers `parseVals` needs to handle that exception.

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 *NEUREX 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 3, XBLOCK gives 100% correct prediction on the

Table 4: Try-Catch Statement Detecting Comparison (XBlock+XState, Instance Level) (RQ2)

Small dataset	Precision	Recall	F1-score
GPT-3.5	0.550	0.232	0.326
XBlock + XState	**0.969**	**0.607**	**0.747**

Table 5: Try-Catch Statement Detecting Result (XState as Individual, Instance Level) (RQ2)

Github dataset	Precision	Recall	F1-score
XState	**1.0**	**0.6198**	**0.7652**

Table 6: Try-Catch Statement Detecting Comparison (XBlock+XState, Statement Level, Small Dataset) (RQ2)

Statement Level	Accuracy
GPT-3.5	0.601
XBlock + XState	**0.871**

Table 7: Try-Catch Statement Detecting Result (XState as Individual, Statement Level, Github Dataset) (RQ2)

Statement Level	Accuracy
XState	**0.874**

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)

Table 4 shows the result when we evaluated XState in connection with XBlock. That is, both individual results from XBlock and XState must be correct for the instance to be considered correct. Neurex achieves a very high precision (96.9%) and predicts correctly *all the statements in all* `try-catch` *blocks* (could have multiple blocks) for 61% of the positive code snippets. It improves relatively over GPT-3.5 **76.1%**, **161.7%**, and **128.7%** in Precision, Recall, and F1-score. Examining the results, we found that GPT-3.5 did not work well for the code snippets that have more than one `try-catch` blocks. It also does not recognize well multiple statements with dependencies that need to be placed in the same block. Neurex recognizes/captures well the dependencies among statements (see RQ4), thus, better grouping them into a `try-catch` block.

Table 5 displays the result when we evaluated XState individually (i.e., assuming XBlock correctly predicts the presence of `try-catch` blocks). As seen, the numbers are slightly higher than those for XBlock+XState because there is no impact from XBlock's result. In other words, XState manages to achieve a 100% precision, showing that XState is capable of giving correct predictions for those false negatives from XBlock.

Table 6 displays the result at the statement level (i.e., whether a statement needs to be inside a `try-catch` block or not). As seen, XBlock+XState improves relatively over GPT-3.5 **44.9%** in accuracy at the statement level. As seen in Table 7, Neurex also achieves high numbers in accuracy at the statement level as individual. As expected, it has a slightly higher accuracy than XBlock +XState.

Table 8: Exception Type Recommendation (XType as Individual, Github Dataset) (RQ3)

XType (Github)	Number of Exceptions						
	One	Two	Three	Four	Five	Six	All
Precision	0.973	0.977	0.978	0.938	0.975	1.0	0.974
Recall	0.739	0.484	0.468	0.469	0.75	0.5	0.569
F1-score	0.840	0.648	0.633	0.625	0.848	0.666	0.718

Table 9: Exception Type Recommendation Result (XType as Individual) (RQ3)

Github dataset	Accuracy
XType	**0.733**

Table 10: Exception Type Recommendation Result (XBlock +XState +XType) (RQ3)

Github dataset	Precision	Recall	F1-score
XBlock + XState + XType	**0.959**	**0.451**	**0.613**

Table 11: Exception Type Recommendation Comparison (XBlock +XState +XType) (RQ3)

Small dataset	Precision	Recall	F1-score
GPT-3.5	0.492	0.181	0.264
XBlock + XState + XType	**0.724**	**0.682**	**0.702**

9.3 Exception Type Recommendation (RQ3)

As seen in Table 8, as individual, XType achieves 97.4%, 56.9%, and 71.8% in Precision, Recall, and F1-score, respectively on the Github dataset. While it performs better for the case of single exception type, the results for the other cases with two or more exceptions to be caught are also consistent.

Table 9 shows that regardless of the blocks, the accuracy for all exception types is 73.3%. That is, almost 3 out 4 predicted exception types, XType is correct.

As seen in Table 10, **Neurex as evaluated as three components, achieves 95.9%, 45.1%, and 61.3%** in Precision, Recall, and F1-score, respectively. In almost 96% of the predictions, Neurex correctly decides the need of the `try-catch` block, the number of blocks and corresponding statements, and the exception types in the `catch` clauses. Neurex covers 45.1% of the exception types, resulting a F1-score of 61.3%. Comparing with Table 8, the numbers are lower because it has impacts of the results from XBlock and XState. In total, Neurex predicts **correctly in all three tasks for 22,010 out of 30,764 total instances (71.5%)** in Github dataset. Among 15,328 positive samples, **Neurex predicts correctly 6,928 positive instances (45%) in all three tasks**.

Finally, in comparison with GPT-3.5, as seen in Table 11, Neurex as evaluated as three components, achieves relatively higher in Precision, Recall, and F1-score with **47.1%, 278%, and 166%, respectively**. In 190 positive instances, Neurex predicted correctly all three tasks in 30 instances. In 380 all instances, it predicted correctly all three tasks in 219 instances (57.6%).

Examining the result from GPT-3.5, we reported that for popular APIs, it works well, e.g., `ClassNotFoundException`, `IllegalArgumentException`, `IOException`, `IndexOutOfBoundException`, etc. For unpopular API calls, it resorted to using the general exception `Exception` (694 in total, 58.6%)

Table 12: 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

```
public void onCreate(Bundle state) {
  requestWindowFeature(Window.FEATURE_NO_TITLE);
  + try {
    final WindowManager.LayoutParams attrs = getWindow().getAttributes();
    final Class<?> cls = attrs.getClass();
    final Field fld = cls.getField("buttonBrightness");
    if (fld != null && "float".equals(fld.getType().toString())) {
    fld.setFloat(attrs, 0);
  + }
  + } catch (NoSuchFieldException e) {
  + } catch (IllegalAccessException e) {
  + }...
}
```

Figure 10: Exception-related Bug #106 in FuzzyCatch dataset (missing `try-catch`) (detected by NEUREX)

or `APIException` as an answer. For the cases of multiple `try-catch` blocks, the common errors are the use of `Exception` for all blocks.

9.4 Dependency Probing (RQ4)

9.5 Exception-Related Bug Detection (RQ5)

As seen in Table 12, 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 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 10 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.

9.6 Impact of Fine-Tuning (RQ6)

Table 13: Impact of Fine-Tuning in NEUREX (RQ6)

Github dataset (XBlock)	Precision	Recall	F1-score
CodeBERT w/o fine-tuning	0.497	**0.972**	0.657
NEUREX	**0.981**	**0.984**	**0.982**

As seen in Table 13, fine-tuning contributes much to NEUREX in much improving Precision (almost twice) and slightly improving in Recall (1%), and much improving in F1-score (relatively 49.5%). Without fine-tuning, the model overwhelmingly predicts that the 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.

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`. Fourth, 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 projects with well-known libraries. FuzzyCatch [?] does not suggest statements and exception types. Thus, we compared only with XRank.

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.