Programming Assistant for Exception Handling with CodeBERT

Yuchen Cai
University of Texas at Dallas
Texas, USA
yuchen.cai@utdallas.edu

Aashish Yadavally
University of Texas at Dallas
Texas, USA
aashish.yadavally@utdallas.edu

Abhishek Mishra
University of Texas at Dallas
Texas, USA
abhishek.mishra@utdallas.edu

Genesis Montejo
University of Texas at Dallas
Texas, USA
genesis.montejo@utdallas.edu

Tien N. Nguyen
University of Texas at Dallas
Texas, USA
tien.n.nguyen@utdallas.edu

ABSTRACT

With practical code reuse, the code fragments from developers' 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` 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 these three exception handling recommendation tasks. Via the large language model, NEUREX can learn the surrounding context, leading to better learning the dependencies among the API elements, 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 a F1-score of 70.2%, which is a relative improvement of 166% over the baseline. It achieves high F1-score from 98.2%–99.7% in `try-catch` block necessity checking (a relative improvement of up to 55.9% over the baselines). It also correctly decides both the need for `try-catch` block(s) and the statements to be placed in `try` blocks with the F1-scores of 74.7% and 87.1% at the instance and statement levels, an improvement of 129.1% and 44.9% over the baseline, respectively. Our extrinsic evaluation shows that NEUREX relatively improves over the baseline by 56.5% in F1-score for detecting exception-related bugs in incomplete Android code snippets.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Software and its engineering** → **Language features**.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04
https://doi.org/10.1145/3597503.3639188

KEYWORDS

AI4SE, Large Language Models, Automated Exception Handling

ACM Reference Format:
Yuchen Cai, Aashish Yadavally, Abhishek Mishra, Genesis Montejo, and Tien N. Nguyen. 2024. Programming Assistant for Exception Handling with CodeBERT. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639188

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.* [41] 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* [41]. 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.* [25] 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.* [15] found that 19% of the crashes in Android applications could have been caused by insufficient documented exceptions in Android APIs. In fact, it has been reported that the lack of API documentation and specifications has hindered software library utilization [10]. Thus, it is desirable to have automated tools to recommend proper exception handling for the adaptation of online code snippets.

There exist several approaches to automatic recommendation of exception handling [3–5, 25, 32, 42]. They can be classified into four categories. The first category of approaches relies on *program analysis heuristics* on exception types, API calls, and variable types to recommend exception handling code [4]. These heuristic-based approaches do not always work in all the cases due to incomplete code. The second category utilized *exception handling policies*, which are

enforced in all cases [5, 20]. However, the policies need to be pre-defined and encoded within the recommendation 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 snippets [32]. While avoiding hard-coding of the rules, these mining approaches suffer the issue of how much similar for two snippets to be considered as having similar exception handling. For the mining approaches, deterministically setting a right threshold for frequent occurrences is challenging.

To provide flexibility in code matching, several approaches follow the fourth category in *information retrieval* (IR). XRank [25] takes the input source code and recommends a ranked list of API method calls in the code that are potentially involved in the exceptions in a `try` block. XHand [25] recommends the exception handling code in a `catch` block for a given code. Both use a fuzzy set technique to compute the association scores between the API calls (e.g., `new BufferedReader`) and the exceptions (e.g., `IOException`).

While the IR-based approach achieves higher accuracy than the others [25], it has key limitations. First, it is not trivial to *pre-define a threshold* for feature matching for the 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` *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, machine learning-based approach from the previous studies reporting that exception handling for the API elements is frequently repeated across projects [32, 43]. The reason for such repetitions is that the designers of a software library have the intents for users to utilize specific API elements paired with corresponding exception types. This intention is typically conveyed through API documentation, which may be inadequate or even absent [10]. Thus, we design Neurex to learn from the statements in `try` blocks and exception types retrieved from *complete source code* in a large code corpus, and derive suggestions for a given *(in)complete code*.

We leverage and fine-tune CodeBERT [11], a large language model, to capture the surrounding code **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 the statements with API elements* in the given (in)complete code, leading to better learning in XBlock, XState and XType. Inspired by the sequence chunking methods [40] 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 employs a **multi-tasking** mechanism, incorporating those tasks to foster mutual influence among their learning. This enhances the performance across all tasks.

Our above idea gives Neurex advantages over the state-of-the-art IR approaches. First, with the learning-based approach, Neurex does not rely on a pre-defined threshold for explicit feature matching to retrieve API elements or exceptions. Second, instead of computing the association score between an API element and an exception type, Neurex relies on *dependencies* and *contexts* in both prediction and training. During training, the complete code provides the context for Neurex to learn the dependencies among the API elements and the exception types. During prediction, for a given (in)complete code, Neurex can implicitly match the current context with such knowledge to avoid the name ambiguity and to derive the exception handling suggestions. For example, encountering the program dependencies among the API elements and exceptions in JDK, e.g., `getDeclaredField` in `java.lang.Class`, `get` in `java.lang.Class.Field`, and `NoSuchFieldException` in the training data, will help Neurex learn to match the context in a given code and then to suggest `NoSuchFieldException`.

We conducted several experiments to evaluate Neurex with a large dataset of 5,726 projects from GitHub having 246,118 code snippets for training, 30,764 for validation, and 30,764 for testing. Our empirical evaluation shows that Neurex correctly performs all three exception handling recommendation tasks in 71.5% of the cases with a F1-score of 70.2%. It has a relative improvement of 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 `try` blocks with the F1-scores of 74.7% and 87.1% at the instance and statement levels, an improvement of 127.3% and 44.9% over the baseline, respectively. Our extrinsic evaluation shows that Neurex relatively improves over the baseline by 56.5% in F1-score for detecting exception-related bugs in incomplete Android 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 tasks. It works for either complete or incomplete code.
(2) **Multi-tasking among three Tasks of Exception Handling Recommendation**. We formulate the problem as sequence chunking with a multi-tasking scheme for three tasks.
(3) **Empirical Evaluation**. Our evaluation shows Neurex's high accuracy in exception handling suggestion and exception-related bug detection. Data and code is available [22].

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*

```
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

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 [41], this code snippet was adopted by developers into a GitHub project named `armint` (Figure 2). `armint`'s developers handled 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 [41]. Such manual adaptation for a code snippet could lead to exception-related bugs, which could cause serious issues including crashes or unstable states [25]. 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 [3–5, 25, 32]. However, the state-of-the-art, IR-based approaches [25], which have been shown to outperform others, still have limitations. First, it is not trivial to pre-define a *threshold* for feature matching for a retrieval, e.g., the threshold to determine the association scores between an API call (e.g., `getDeclaredField`) and an exception type (e.g., `NoSuchFieldException`). Thus, the pre-defined threshold affects much the effectiveness. Second, relying on the lexical values of API elements' names, they suffer the issue of *ambiguous names* of the APIs or exceptions (e.g., the API method `get` at line 6 of Figure 1 occurs in multiple libraries) in an incomplete code snippet, which might not be parseable for fully-qualified name resolution. Thus, this also reduces effectiveness. Finally, they consider only the *association pair between an API method and an exception type*, and *discard the surrounding context*. For example, they examine the associations between the names of the API call (e.g., `getDeclaredField`) and the exceptions to be handled (e.g., `NoSuchFieldException`, `SecurityException`,

```
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 S/O Post in Figure 1

```
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

etc.). Without the context, it is challenging to decide the identities of the APIs and their exceptions via only simple names.

Let us 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` 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 complete code in public code repositories 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.*

In addition, the relations between the API `java.lang.Class.getDeclaredField` and the exceptions `NoSuchFieldException`, `SecurityException`, `IllegalArgumentException`, and `IllegalAccessException` exhibit in the code corpora. Thus, a model can learn to recommend those

```
1  Charset charset = Charset.forName("US-ASCII");
2  try {
3      BufferedReader reader = Files.newBufferedReader(file, charset);
4      String line = null;
5      while ((line = reader.readLine()) != null) {
6          System.out.println(line);
7      }
8  } catch (IOException x) {
9      System.err.format("IOException: %s%n", x);
10 }
```

Figure 4: An Example Using `newBufferedReader` to Read from File

exceptions for any complete or incomplete code snippet involving the API `getDeclaredField`.

Observation 3 (**Relations between API Elements and Exceptions**). *The presence of the relations between API elements and exceptions helps a model learn to suggest exception handling.*

For an incomplete code snippet, the simple names of the API elements (methods, fields, classes) can be ambiguous. However, from the training data, if a model could learn the *dependencies among the API elements* and the *relations between the APIs and the exceptions* in the code context, it can match the context of the given code snippet to the learned relations to suggest exception handling.

In Figure 3, the dependencies among `Field` (line 3), `getDeclaredField` (line 3), `setAccessible` (line 5), `get` (line 7), etc. are as follows. The return type of the API call to `getDeclaredField` is `Field` (thanks to line 3), which has an API method named `setAccessible` (thanks to line 5) and another API method named `get` (thanks to line 7). If a model can be trained to learn from such dependencies among the API elements during training, it can match the given incomplete code in Figure 1 with the similar dependencies among the API elements `Field` (line 2), `getDeclaredField` (line 2), `setAccessible` (line 3), and `get` (line 6). Thus, the model can learn to suggest the exception handling similarly to the complete code in training (Figure 3).

Observation 4 (**Surrounding Code Context with Dependencies among API Elements**). *The code context with dependencies among the API elements can help resolve the ambiguity in incomplete code snippets, leading to better prediction of the handled exceptions.*

For the list of statements in an incomplete code, not all of them needs to be wrapped around in a `try` block. For example, let us consider the example of using `newBufferedReader` in Figure 4. The API call `java.nio.file.newBufferedReader(.)` needs to be within a `try` block, while 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` block as well.

Observation 5 (**Learn to Decide What Statements to Be Placed within a `try` Block**). *A model can learn from the code corpora what statements need to be placed within a `try` block or not.*

2.2 Key Ideas

We introduce Neurex with three functions for exception handling recommendation: given an (in)complete 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 within a `try` 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. Instead of deterministically deriving the exceptions to be handled for a given (incomplete) code snippet, from Observation 2, we use a machine learning-based approach to suggest to properly handle the exceptions via the above tasks. By learning from the `try-catch` blocks of complete code in the open-source projects in training, Neurex can suggest exception handling.

2.2.2 [Key Idea 2] Leveraging a Code-aware Language Model to learn Context with Dependencies among API Elements and Relations with Exceptions. Instead of computing the association scores for the pairs between API elements and exception types as in IR-based approaches, we leverage as the context the complete code in the training corpus, to fine-tune a code-aware language model (LM) to learn the dependencies among the API elements, and their relations with the corresponding exception types. In predicting for a snippet C, Neurex uses the LM to take the context with the dependencies among the API elements in C and suggest exception handling in three tasks (Observation 4). The relations between API elements and the exceptions also help the model suggest the exception types.

2.2.3 [Key Idea 3] Leveraging Sequence Chunking with a Language Model and Multi-tasking. Inspired by the sequence chunking techniques [40] in NLP, we formulate our problem as identifying one or multiple chunks of consecutive statements that need to be placed within `try` blocks. We leverage and fine-tune the language model CodeBERT [11] to learn the relations among the statements with the API elements. We also utilize a multi-tasking framework for three tasks: XBlock, XState, and XType, as the knowledge learned from one task can enhance the performance of the others.

3 NEUREX OVERVIEW

Figure 5 illustrates the overview of Neurex. Generally, Neurex 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 placed within a `try` block, and 3) XType aims to detect the exception types need to be caught in the `catch` clause(s). 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` block(s) and the exception type(s) in the `catch` clause(s) 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 snippet is fed into a large language model. We use CodeBERT [11] as the code representation learning model to produce vector representations for the (sub)tokens and statements in the source code, as CodeBERT is capable of producing embeddings that capture both the syntactic and semantic information.

The vectors are used as the inputs for three components. The vectors for the `[SEP]` tokens are used to represent the respective

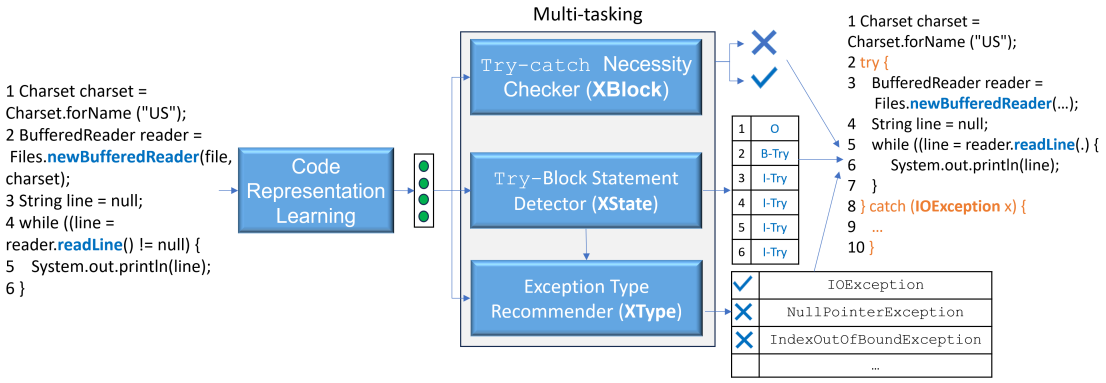


Figure 5: NEUREX: Neural Network-Based Exception Handling Recommender

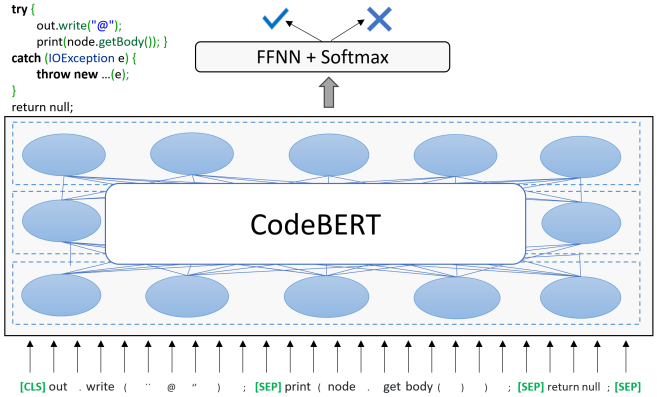


Figure 6: XBLOCK: `Try-catch` Necessity Checker

statements, while the summation of the vectors for the `[SEP]` tokens at the output layer is used for multiple 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 [40] 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` block), `B-Try` (i.e., it is the beginning of a `try` block), or `I-Try` (i.e., it is inside of a `try` block). During training, the statements within or outside of the `try` blocks enable us to build their tags/labels.

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. A *No* outcome indicates otherwise. During training, the exception types for each `try-catch` block in a positive sample 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 code.

4 XBLOCK: TRY-CATCH NECESSITY CHECKER

Figure 6 illustrates XBLOCK's architecture. Given an input code snippet, XBLOCK first splits it into the statements (Figure 6). Each statement is then tokenized into sub-tokens using CodeBERT's tokenizer. We use a special separator token `[SEP]` to concatenate the tokenized statements, and add a `[CLS]` token at the beginning of the code snippet. As in CodeBERT, we take that `[CLS]` token to be the representation of the entire code snippet.

We fine-tune a CodeBERT (MLM) [9] for this task. 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 often used together in API usages and they require `IOException`. For the input incomplete code, 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 one CodeBERT, all three modules (Sections 5 and 6) contribute to the signal for updating the model's parameters (multi-tasking). 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 be placed in a `try-catch` block.

5 XSTATE: TRY-BLOCK STATEMENT DETECTOR

Figure 7 illustrates the architecture of XSTATE, the `try`-block statement detection component. The goal of XSTATE is to decide if a statement in the given code snippet needs to be placed within a `try` block. For code representation learning, we use one single CodeBERT [11] model as in XBLOCK (see Figure 6) to produce the embedding for the code (sub)-tokens. For the input code, a `[SEP]` token represents the statement preceding it.

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 NEUREX to learn the *relations of an API and its exception types*, thus, making the connection between the API and the presence of `try-catch` block. For example, in Figure 7, `write` in the statement s_1 could be determined as having a relation with `IOException`, leading to better learning to place that statement inside a `try` 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). Such dependencies would

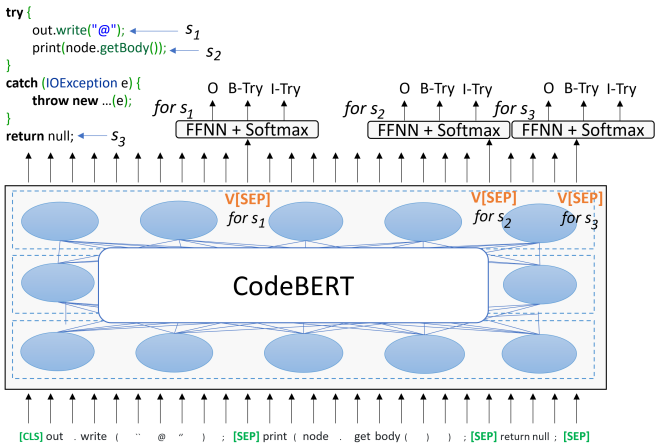


Figure 7: XSTATE: Try-Block Statement Detector

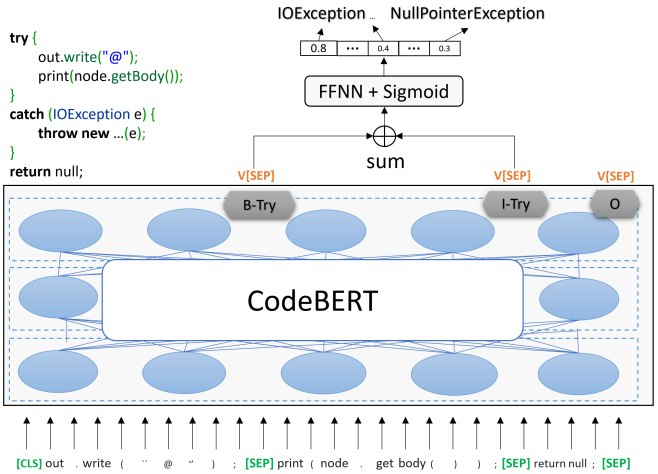


Figure 8: XTYPE: Exception Type Recommender

help the model better decide whether some consecutive statements need to be placed together in a `try` block. For example, in Figure 5, the statements at lines 2–5 have data dependencies, thus, they might be in the same `try` 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` *blocks;* `B-Try` *tag means that the statement begins a* `try` *block; and* `I-Try` *tag means that the statement is inside a* `try` *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` label because it is the first statement of a `try` 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` 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` block. With this IOB2 encoding [13], we can support multiple `try-catch` blocks in which the statement with `B-Try` is the start of a `try` block and the statement with the respective `I-Try` tag is the end of that block. A new block is formed as another statement having a `B-Try` tag.

In training, the labels for all statements are known from the code. In prediction, NEUREX will assign IOB2 labels to the statements.

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 and XSTATE to build the embeddings for code (sub)-tokens. We expect CodeBERT to learn the connection between the statements in a `try` block and the corresponding exception types to be caught. From Key Idea 2, we expect that via context, CodeBERT can implicitly learn the dependencies among API elements, leading to better learning of the exception types.

During training, we know all the exceptions to be caught in a code snippet. For each `try` block, we identify the statement that begins it (with the `B-Try` label) and ends it (with the respective `I-Try` label). Then, we calculate the sum of the embeddings from CodeBERT, for the `[SEP]` tokens that correspond to the statements inside the `try` block and feed this `try` block's representation vector into a linear layer. We use a sigmoid function to perform binary classifications for the exception types 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` 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` 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 since its score is higher than 0.5.

7 MULTI-TASK LEARNING

Mutual enhancement can occur through learning on the three tasks: XBLOCK, XSTATE, and XTYPE. When a model discerns the necessity of a `try-catch` block, it implies the existence of specific statements within the code snippet that warrant such a block. By training a model to recognize the statements to be placed in a `try` block, it gains the ability to determine the need for a `try-catch` block and establish associations with the respective exception types. Moreover, the model's knowledge of exception types aids in making informed decisions about which crucial statements should be in a `try` block. Thus, we put these three tasks in a multi-task learning scheme.

We calculate the training loss by combining the losses from the three tasks. $Loss_{\text{XBLOCK}}$ is the Binary Cross Entropy loss for the decision as to if `try-catch` block(s) is needed. To calculate $Loss_{\text{XSTATE}}$, we add the classification losses for all statements in the input, where a statement loss ($Loss_{Stmt}$) is the Cross Entropy loss calculated from the distribution of the three tags (`O`, `B-Try`, `I-Try`) and the ground-truth tag. Finally, in XTYPE, since several `try-catch` blocks might be present, $Loss_{\text{XTYPE}}$ comes from the summation of the exception prediction losses from all the `try-catch` blocks. For each `try-catch` block, $loss_{Try-block}$ is calculated by adding the Binary Cross Entropy loss for the prediction of each exception of interest.

The overall training loss is calculated as follows. If the input does not contain any `try-catch` block, the overall loss will be the $Loss_{\text{XBlock}}$. If the input contains a `try-catch` block, the overall loss will be the summation of 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

To evaluate Neurex, we seek to answer the following questions:

RQ$_1$. [XBlock's 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$_2$. [XState's Effectiveness on `Try`-Block Statement Detection]. *How accurate is Neurex in predicting which statements in a given code snippet need to be placed in a* `try` *block?*

RQ$_3$. [XType's 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$_4$. [Statement Dependency Probing]. *How well Neurex learn the statement dependencies for grouping them into a* `try` *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 [25] 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 JDK and Android libraries. These are the well-established libraries that have been used in several prior research on the topics related to APIs [29, 35]. We then selected the methods with at least one `try-catch` block as positive samples, and 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 246,118 code snippets for training, 30,764 for validation, and 30,764 for testing. In 30,764 testing samples, there are an equal number of positive and negative ones (15,382).

For extrinsic evaluation, we used FuzzyCatch dataset, provided by the authors of XRank/XHand [25], which contains 609 Android incomplete code snippets with exception-related bugs (missing `try-catch` blocks or exception types). Finally, we used 553 code snippets because the others are not valid for the experiment.

8.2.2 XBlock's Effectiveness on `Try-Catch` Necessity Checking (RQ1). *Baselines.* We compared XBlock with GPT-3.5 [8]. Due to the time complexity of using OpenAI's ChatGPT, 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 our model 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 [25]) on the GitHub dataset. XRank computed the exception risk score for each API call in a code snippet. If a score of an API call in the code snippet is higher than a threshold, we consider the case 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 and negative samples; and the training and tuning partitions do not contain any duplicates.

To request responses from ChatGPT, we constructed a prompt with the format "question+code", where the question was *"Does the code below need to catch any exceptions?\n\n"*. Considering that ChatGPT's answers for the same prompt may vary, for each code snippet, we sent the prompt three times through Chat Completions API.

Labeling the responses has three steps. First, we checked 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 assigned labels to them. However, there are some cases in which it is hard to make the decision on whether or not the code needs any `try-catch` blocks. The common scenarios are (1) the response is not informative enough, as it only relays a general advice about exception handling, (2) the response states that it is uncertain whether a `try-catch` block is needed, or (3) the decision making depends on the background knowledge on either the project structure or certain method specifications. Thus, in evaluating GPT-3.5's performance, for these responses, given the benefit of doubts, we assigned the correct prediction labels to them, assuming the best performance of GPT-3.5. We assigned the final label for each case from the majority vote of the three attempts.

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 selected the model with the lowest overall validation loss.

Metrics. We use *Precision, Recall,* and *F1-score* to evaluate the performance of the approaches. They are calculated as follows. $Precision = \frac{TP}{TP+FP}$, $Recall = \frac{TP}{TP+FN}$, $F1\text{-score} = \frac{2*Recall*Precision}{Recall+Precision}$. TP: true positive, FN: false negative, and FP: false positive.

8.2.3 XState's Effectiveness on Try-block Statement Detection (RQ2). *Baselines.* We compared XState against GPT-3.5 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` blocks must match with the grouping of the corresponding statements in the corresponding (zero or multiple) `try` 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 code 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` block, regardless of the blocks themselves. Thus, we use *Accuracy* for the evaluation at the statement level, which is defined as the ratio between the number of correct tags on the 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 (i.e., correct predictions on the need of a `try-catch` block as well as on the statement tagging). Second, we also evaluated XSTATE as individual. We assume that XBLOCK predicted correctly on the positive instances then evaluated XSTATE individually at both instance and statement levels as explained.

8.2.4 XTYPE's Effectiveness on Exception Type Prediction (RQ3). *Baselines.* We compared XTYPE against GPT-3.5 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 between the predicted and oracle sets and the size of the oracle set), and 3) F1-score (harmonic mean of Precision and Recall).

As with XSTATE, we evaluated XTYPE in two ways. First, we evaluated XTYPE in connection with XBLOCK and XSTATE. We consider a case as correct if all three components give correct predictions. Second, we also evaluated XTYPE as individual: we evaluated XTYPE with the above metrics in the cases in which XBLOCK and XSTATE are correct at the instance level.

8.2.5 Statement Dependency Probing (RQ4).
We evaluate if NEUREX could learn the connections among the statements inside the same `try` block. We selected the instances that it predicted correctly in all three tasks. For a `try-catch` block in an instance, we randomly selected a statement S_1 and another statement S_2 inside the block. We then randomly selected another statement T outside of the block. We measured the cosine distances $d_1(S_1, S_2)$ and $d_2(S_1, T)$ for the statement embeddings. We repeated that for all triples of (S_1, S_2, T) in the GitHub dataset, and computed the cosine distances for the group of inside statement pairs and the group of inside-to-outside statement pairs. For each group, we constructed confidence intervals at 95% confidence via bootstrapping for the mean of the distances (the number of re-samples is 1,000). The comparison of the distances allows us to probe whether our model is able to distinguish the statements inside from those outside of the `try` blocks via statement embeddings.

8.2.6 Extrinsic Evaluation on Exception Bug Detection (RQ5).
Baselines. We compared with FuzzyCatch [25] in exception-related bug detection, i.e., missing `try-catch` blocks and/or exception types, or incorrect exception types.

Procedure. We trained NEUREX on the GitHub dataset and detected the exception-related bugs [12] in FuzzyCatch dataset of incomplete code snippets. If exception handling in a snippet matches exactly with NEUREX's suggestion, we consider it as correct.

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

9 EMPIRICAL RESULTS

9.1 XBLOCK: `Try-catch` Necessity Checking (RQ1)

As seen in Table 1, NEUREX achieves very high Precision, Recall and F1-score on the GitHub dataset—all above 98%.

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

GitHub dataset	Precision	Recall	F1-score
XRank [25]	0.810	0.530	0.630
NEUREX	**0.981**	**0.984**	**0.982**

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

Small dataset	Precision	Recall	F1-score
GPT-3.5 [8]	0.804	0.778	0.791
NEUREX	**0.994**	**1.0**	**0.997**

9.1.1 Comparison with XRank. 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 in comparison with the baseline. First, XRank's recall is around 0.53 in our balanced dataset. In XRank, if the association score of *only one API call* 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 all the API method calls in all 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 mistakenly considers them the same due to its IR approach. Moreover, unlike XRank, which considers only the association scores between the APIs and exception types, NEUREX considers the code in the block as the context to learn the dependencies among API elements and the relations between the API calls and exception types, leading to better `try`-block detection.

9.1.2 Comparison with GPT-3.5. As seen in Table 2, NEUREX relatively improves over GPT-3.5 *23.6%*, *28.5%*, and *26%*, 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 *a* with similar name, and predicted that the given code snippet needs a `try-catch` block since *a* requires a `try-catch` 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 makes an incorrect prediction.

9.1.3 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. We first get the *attribution scores* for the input (sub)tokens via Transformer Interpret [37]. A higher attribution score of a token means that the token has more contribution to the model's prediction. In Figure 9, we show the attribution score for each statement, which is calculated by averaging the attribution scores of all the sub-tokens in the statement. A positive attribution score means that

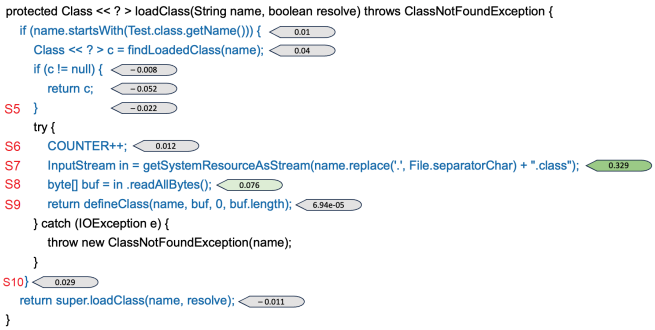


Figure 9: XBlock Puts Attention on the Right Tokens

Table 3: `Try-Catch` Necessity Checking Evaluated on Test Set Partitioned by 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	0.998	1.0	0.986	1.0
F1-score	0.0	0.991	0.999	1.0	0.993	1.0

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 (*S7*) and the statement that invokes the `readAllBytes` method call on the `InputStream` object (*S8*). This example illustrates that *Neurex is able to put the attention on the right tokens of those statements*, e.g., `InputStream`, `in`, `get`, `System`, `replace`, etc. (`InputStream` and `readAllBytes` need exception handling), leading to its correct prediction.

9.1.4 Result on Test Set Partitioned by Number of `Try-Catch` Blocks. 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 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 XState: `Try`-Block Statement Detection (RQ2)

Table 4: `Try`-Block Statement Detection Result (XState as Individual, Instance Level) (RQ2)

GitHub dataset	Precision	Recall	F1-score
XState	**1.0**	**0.620**	**0.765**

9.2.1 Performance of XState as Individual. Table 4 displays the result as we evaluated XState individually at the instance level (i.e., assuming XBlock correctly predicts `try-catch` blocks). Comparing Table 4 with Table 6, the result of XState as individual is slightly higher than that of XBlock+XState due to no impact from XBlock. Moreover, XState manages to achieve a 100% precision, showing that XState is capable of giving correct predictions for all the statements when XBlock is correct (including those correctly predicted as 'not-needed' by XBlock, i.e., all the statements are outside).

Table 5: `Try`-Block Statement Detection Result (XState as Individual, Statement Level) (RQ2)

GitHub dataset	Statement Level	Accuracy
	XState	**0.874**

As an individual, as seen in Table 5, Neurex also achieves high accuracy at the statement level. Comparing Table 5 and Table 7, it has a slightly higher accuracy than XBlock+XState because XBlock by itself has achieved high performance (+98% F1-score).

9.2.2 Performance of XBlock+XState. Table 6 shows the result when we evaluated XState+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` *blocks* (could have multiple blocks) for 61% of the positive code snippets. It improves relatively over GPT-3.5 *76.2%*, *161.6%*, and *129.1%* in Precision, Recall, and F1-score, respectively. Examining the results, we found that GPT-3.5 did not work well for the code snippets with multiple `try-catch` blocks. It also does not recognize well multiple statements with dependencies that need to be placed in the same `try` block. Neurex recognizes well the dependencies among statements (see Section 9.4), leading to better grouping them into a `try` block.

Table 6: `Try`-Block Statement Detection 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 7: `Try`-Block Statement Detection Comparison (XBlock+XState, Statement Level) (RQ2)

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

Table 7 displays the result at the statement level (i.e., whether a statement needs to be inside a `try` block). As seen, XBlock+XState is able to correctly tag 87% of the statements on their locations with respect to a `try` block. In comparison, XBlock+ XState improves *44.9%* relatively over GPT-3.5 in accuracy at the statement level.

9.3 XType: Exception Type Suggestion (RQ3)

9.3.1 Performance of XType as Individual. As seen in Table 8, as individual, XType achieves 97.4%, 56.9%, and 71.8% in Precision, Recall, and F1-score, respectively, when considering all possible numbers of exception types on the Github dataset. While it performs better for the case of single exception type, the results for the other cases with multiple exceptions needed to be caught are also consistently high. The result is still high even for the cases of more exception types. Precision ranges from 93.8% to 100%; recall ranges from 46.8% to 75%; and F1-score ranges from 62.5% to 84.8%.

Table 9 shows that the accuracy for all exception types is high: 73.3%. That is, XType predicts correctly almost 3 out 4 predicted exception types, regardless of the number of `try-catch` blocks and that of exception types needed to be handled in each `catch` clause.

Table 8: Exception Type Recommendation Result (XType as Individual) (RQ3)

Github	Number of Exception Types						
	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 Accuracy (XType as Individual) (RQ3)

GitHub dataset	Accuracy
XType	**0.733**

9.3.2 Performance of **Neurex** *(XBlock+XState+XType).* 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, it correctly decides the need of the `try-catch` block(s), the number of blocks and respective 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 due to the impacts 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.* Moreover, the higher performance of XBlock+XState+XType over the multiplication of individual component's accuracy indicates the benefit of our multi-tasking.

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

GitHub dataset	Precision	Recall	F1-score
Neurex = XBlock + XState + XType	**0.959**	**0.451**	**0.613**

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

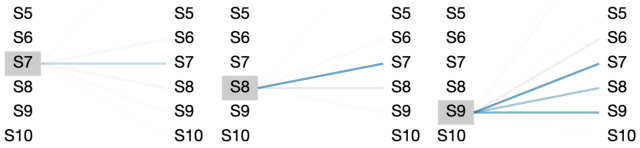
Small dataset	Precision	Recall	F1-score
GPT-3.5	0.492	0.181	0.264
Neurex = XBlock + XState + XType	**0.724**	**0.682**	**0.702**

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 all 380 instances, it predicted correctly all three tasks in 219 instances, resulting a total of 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%) or `APIException` as an answer. For the cases of multiple `try-catch` blocks, the common errors are the use of `Exception` for all the blocks.

9.4 Dependency Probing (RQ4)

9.4.1 Attentions between Statements. Figure 10 shows the attention weights, at the output layer, for the example in Figure 9. We used



(a) Rel. from S7 **(b) Rel. from S8** **(c) Rel. from S9**

Figure 10: Dependency Probing via Attentions for Figure 9

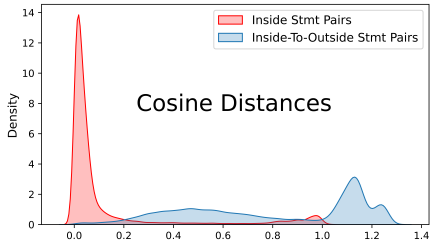


Figure 11: Distribution of Distances of Statement Embeddings

BertViz [6] to display the weights in which a darker line means a higher attention weight. As seen, the statement *S7* (creation of an `InputStream`) puts most attention on itself (Figure 10a). *S8* (invoking `readAllBytes()` on the `InputStream` object) indeed has most attention on *S7* (Figure 10b). As seen in Figure 10c, *S9* pays attention more to *S7* and *S8* (beside itself). These connections reflect well the *data dependencies* among the statements. None of the three statements puts much attention on any statement outside the `try` block. Importantly, these statements contain the crucial API elements with respect to correctly catching the `IOException`. Thus, this example shows that Neurex captures the dependencies among the statements with crucial API elements, leading to correct exception handling.

9.4.2 Distances between Statement Embeddings. As seen in Figure 11, the distribution of the cosine distances for all the inside-to-outside statement pairs is largely to the right of the distribution for all the inside statement pairs. That is, Neurex tends to *encode statements in a way that the statements in the same* `try` *block are closer to each other in the embedding space*, leading to better statement grouping. Specifically, the confidence interval of the mean of cosine distances for the inside statement group is from 0.1262–0.1268 with 95% confidence, and the confidence interval for the inside-to-outside statement group is from 0.7987–0.8001 with 95% confidence.

9.5 Exception-Related Bug Detection (RQ5)

Table 12: Exception-Related Bug Detection (RQ5)

	FuzzyCatch Dataset	
	Neurex	FuzzyCatch [25]
Recall	**0.95**	0.76
Precision	**1.0**	0.54
F1-score	**0.97**	0.62

Table 12 displays the comparison result on exception-related bug detection. As seen, Neurex can be used to detect well real-world exception-related bugs in which a code snippet needs but did not

```
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   + }...
```

Figure 12: NEUREX Detected Exception-related Bug #106 (missing `try-catch`) in FuzzyCatch Dataset

have a `try-catch` block, incorrectly caught or missed some exception types. In comparison, NEUREX improves relatively 85.2% in Precision, 25% in Recall, and 56.5% in F1-score over FuzzyCatch [25]. If the association score between a single API method call in the code snippet and one exception type surpasses the pre-defined threshold, Fuzzy-Catch will determine that the snippet contains a bug. Its result is more on the "Yes" (buggy) side. Thus, its precision is slightly better than the probability of a coin toss (54%).

Figure 12 shows an exception-related bug detected by NEUREX and the correctly suggested fix. The code snippet uses the API method `getField` of the JDK library at line 6. NEUREX is able to detect the missing of a `try-catch` block and to recommend the fix by adding such a block around the statements from line 4 to line 8, and adding the `catch` clauses with two exception types `NoSuchFieldException` and `IllegalAccessException`. It was also able to exclude the lines 1–2 from the `try` block. Notably, NEUREX performed correctly even without the knowledge of the JDK documentation. This highlights the potential of NEUREX to serve as a valuable complement to API documentation, aiding developers in ensuring accurate API usages.

9.6 Impact of Fine-Tuning (RQ6)

Table 13 shows the comparison result between our model with fine-tuning and CodeBERT without it. As seen, fine-tuning contributes 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 with 30,764 samples, CodeBERT assigned the negative label (i.e., no `try-catch` block) to only 236 samples.

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

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

9.7 Limitations and Threats to Validity

9.7.1 Limitations. NEUREX still has several limitations. First, it cannot generate new exception types that were not in the training corpus. Second, it does not support the generation of exception handling code inside the `catch` body. Each project might have a different way to handle exception types in the `catch` body. Thus, we chose not to support that function. Third, NEUREX needs training data, thus, does not work for a new library without any API usage yet. Fourth, it is possible that the model produces the labels `O`, `B-Try`, and `I-Try` that do not correspond to the legal way of a `try-catch` block. Finally, XBLOCK might produce a conflicting result with XSTATE.

9.7.2 Threats to Validity. NEUREX is specifically for Java. Our empirical evaluation was only on Android and JDK libraries. The results might vary for other libraries. Our data might not be representative. However, we used well-established projects and libraries. We used only a balanced dataset, which might not reflect well the ratio in practice. FuzzyCatch [25] does not suggest `try` block statements and exceptions, thus, we compared only with XRank (part of FuzzyCatch). For GPT-3.5, we ran only on a sampled dataset.

10 RELATED WORK

The automated approaches to recommend exception handling can be classified into four categories (Section 1). The closest work to NEUREX is the state-of-the-art *information retrieval* approaches [25], which provides more flexibility than others. XRank [25] recommends a ranked list of API calls that might need exception handling and XHand [25] 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 uses Groum [26], a dependency graph among APIs, to collect the API calls, but did not use dependencies in computing association scores.

In addition to exception handling recommendation research, ThEx [42] predict which exception(s) will be thrown under a given programming context. Other approaches support exception-relevant tasks [14, 17, 18] but do not have same functionality as NEUREX.

There exist research works on API usage mining including exception handling mining and error detection [1, 2, 7, 16, 19, 21, 23, 24, 26–28, 30, 31, 33, 34, 36, 38, 39, 44]. Similar to IR, the mining approaches cannot handle the API name ambiguities.

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. Our empirical evaluation shows that 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 results also demonstrate that NEUREX improves over the state-of-the-art approaches in both intrinsic and extrinsic tasks including exception-related bug detection.

ACKNOWLEDGMENTS

This work was supported in part by the US NSF grant CNS-2120386 and the NSA grant NCAE-C-002-2021.

REFERENCES

[1] Mithun Acharya and Tao Xie. 2009. Mining API Error-Handling Specifications from Source Code. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009* (York, UK) *(FASE '09)*. Springer-Verlag, Berlin, Heidelberg, 370–384. https://doi.org/10.1007/978-3-642-00593-0_25

[2] Sven Amann, Sarah Nadi, Hoan A. Nguyen, Tien N. Nguyen, and Mira Mezini. 2016. MUBench: A Benchmark for API-Misuse Detectors. In *Proceedings of the 13th Working Conference on Mining Software Repositories (MSR '16)*. ACM Press. https://doi.org/10.1145/2901739.2903506

[3] Eiji Adachi Barbosa and Alessandro Garcia. 2018. Global-Aware Recommendations for Repairing Violations in Exception Handling. *IEEE Transactions on Software Engineering* 44, 9 (2018), 855–873. https://doi.org/10.1109/TSE.2017.2716925

[4] Eiji Adachi Barbosa, Alessandro Garcia, and Mira Mezini. 2012. Heuristic Strategies for Recommendation of Exception Handling Code. In *Proceedings of the 26th Brazilian Symposium on Software Engineering*. 171–180. https://doi.org/10.1109/SBES.2012.22

[5] Eiji Adachi Barbosa, Alessandro Garcia, Martin P. Robillard, and Benjamin Jakobus. 2016. Enforcing Exception Handling Policies with a Domain-Specific Language. *IEEE Transactions on Software Engineering* 42, 6 (2016), 559–584. https://doi.org/10.1109/TSE.2015.2506164

[6] BERT Viz [n. d.]. BERT Viz. https://github.com/jessevig/bertviz.

[7] Raymond P L Buse and Westley R Weimer. 2008. Automatic Documentation Inference for Exceptions. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'08)*. ACM Press. https://doi.org/10.1145/1390630.1390664

[8] ChatGPT [n. d.]. OpenAI. https://openai.com/.

[9] codeBERT [n. d.]. CodeBERT MLM. https://github.com/microsoft/CodeXGLUE.

[10] Barthélémy Dagenais and Martin P. Robillard. 2010. Creating and evolving developer documentation: understanding the decisions of open source contributors. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Santa Fe, New Mexico, USA) *(FSE '10)*. Association for Computing Machinery, New York, NY, USA, 127–136. https://doi.org/10.1145/1882291.1882312

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[12] FuzzyCatch bugs [n. d.]. FuzzyCatch bugs. ebrand.ly/ExDataset.

[13] IOB2 [n. d.]. Inside-outside-beginning. https://en.wikipedia.org/wiki/Inside-outside-beginning_(tagging).

[14] Xiangyang Jia, Songqiang Chen, Xingqi Zhou, Xintong Li, Run Yu, Xu Chen, and Jifeng Xuan. 2021. Where to Handle an Exception? Recommending Exception Handling Locations from a Global Perspective. In *Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension (ICPC)*. 369–380. https://doi.org/10.1109/ICPC52881.2021.00042

[15] Maria Kechagia and Diomidis Spinellis. 2014. Undocumented and Unchecked: Exceptions That Spell Trouble. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 312–315. https://doi.org/10.1145/2597073.2597089

[16] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. 2016. How Good Are the Specs? A Study of the Bug-finding Effectiveness of Existing Java API Specifications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. ACM Press, 602–613. https://doi.org/10.1145/2970276.2970356

[17] Yuhang Li, Shi Ying, Xiangyang Jia, Yisen Xu, Lily Zhao, Guoli Cheng, Bingming Wang, and Jifeng Xuan. 2018. EH-Recommender: Recommending Exception Handling Strategies Based on Program Context. In *Proceedings of the 23rd International Conference on Engineering of Complex Computer Systems (ICECCS)*. 104–114. https://doi.org/10.1109/ICECCS2018.2018.00019

[18] Kai Lin, Chuanqi Tao, and Zhiqiu Huang. 2021. Exception Handling Recommendation Based on Self-Attention Network. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 282–283. https://doi.org/10.1109/ISSREW53611.2021.00080

[19] Martin Monperrus and Mira Mezini. 2013. Detecting missing method calls as violations of the majority rule. *ACM Trans. Softw. Eng. Methodol.* 22, 1, Article 7 (mar 2013), 25 pages. https://doi.org/10.1145/2430536.2430541

[20] Taiza Montenegro, Hugo Melo, Roberta Coelho, and Eiji Barbosa. 2018. Improving developers awareness of the exception handling policy. In *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 413–422. https://doi.org/10.1109/SANER.2018.8330228

[21] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Bayesian Specification Learning for Finding API Usage Errors. In *Proceedings of the 11th ACM Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '17)*. ACM Press, 151–162. https://doi.org/10.1145/3106237.3106284

[22] Neurex [n. d.]. Neurex Website. https://github.com/anonymous-000000/neurex.

[23] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. 2016. API Code Recommendation Using Statistical Learning from Fine-Grained Changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 511–522. https://doi.org/10.1145/2950290.2950333

[24] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N. Nguyen. 2012. Graph-based Pattern-oriented, Context-sensitive Source Code Completion. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Computer Society Press, 69–79. https://doi.org/10.1109/icse.2012.6227205

[25] Tam Nguyen, Phong Vu, and Tung Nguyen. 2020. Code Recommendation for Exception Handling. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1027–1038. https://doi.org/10.1145/3368089.3409690

[26] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-based Mining of Multiple Object Usage Patterns. In *Proceedings of the 7th ACM Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Amsterdam, The Netherlands) *(ESEC/FSE '09)*. ACM Press, 383–392. https://doi.org/10.1145/1595696.1595767

[27] Tam The Nguyen, Hung V. Pham, Phong M. Vu, and Tunh Thanh Nguyen. 2015. Recommending API Usages for Mobile Apps with Hidden Markov Model. In *Proceedings of the 30th ACM/IEEE International Conference on Automated Software Engineering (ASE '15)*. IEEE Computer Society Press, 795–800. https://doi.org/10.1109/ase.2015.109

[28] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. 2016. Learning API Usages from Bytecode: A Statistical Approach. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE CS, 416–427. https://doi.org/10.1145/2884781.2884873

[29] Hung Phan, Hoan Anh Nguyen, Ngoc M. Tran, Linh H. Truong, Anh Tuan Nguyen, and Tien N. Nguyen. 2018. Statistical Learning of API Fully Qualified Names in Code Snippets of Online Forums. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 632–642. https://doi.org/10.1145/3180155.3180230

[30] Michael Pradel and Thomas R. Gross. 2009. Automatic Generation of Object Usage Specifications from Large Method Traces. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society Press, 371–382. https://doi.org/10.1109/ASE.2009.60

[31] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. 2012. Statically Checking API Protocol Conformance with Mined Multi-object Specifications. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Computer Society Press, 925–935. https://doi.org/10.1109/icse.2012.6227127

[32] Mohammad Masudur Rahman and Chanchal K. Roy. 2014. On the Use of Context in Recommending Exception Handling Code Examples. In *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation*. 285–294. https://doi.org/10.1109/SCAM.2014.15

[33] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2013. Automated API Property Inference Techniques. *IEEE Transactions on Software Engineering* 39, 5 (2013), 613–637. https://doi.org/10.1109/tse.2012.63

[34] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui. 2015. Mining Multi-level API Usage Patterns. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER '15)*. IEEE Computer Society Press, 23–32. https://doi.org/10.1109/saner.2015.7081812

[35] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. 2014. Live API Documentation. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE'14)*. ACM, 643–652. https://doi.org/10.1145/2568225.2568313

[36] Suresh Thummalapenta and Tao Xie. 2009. Mining Exception-Handling Rules as Sequence Association Rules. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 496–506. https://doi.org/10.1109/ICSE.2009.5070548

[37] Transformers Interpret [n. d.]. Transformers Interpret. https://github.com/cdpierse/transformers-interpret.

[38] Christoph Treude and Martin P. Robillard. 2016. Augmenting API Documentation with Insights from Stack Overflow. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM Press, 392–403.

https://doi.org/10.1145/2884781.2884800

[39] Qian Wu, Guangtai Liang, Qianxiang Wang, Tao Xie, and Hong Mei. 2011. Iterative Mining of Resource-releasing Specifications. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. IEEE Computer Society Press, 233–242. https://doi.org/10.1109/ase.2011.6100058

[40] Feifei Zhai, Saloni Potdar, Bing Xiang, and Bowen Zhou. 2017. Neural Models for Sequence Chunking. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California, USA) *(AAAI'17)*. AAAI Press, 3365–3371.

[41] Tianyi Zhang, Di Yang, Crista Lopes, and Miryung Kim. 2019. Analyzing and Supporting Adaptation of Online Code Examples. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, 316–327. https://doi.org/10.1109/ICSE.2019.00046

[42] Hao Zhong. 2023. Which Exception Shall We Throw?. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 116, 12 pages. https://doi.org/10.1145/3551349.3556895

[43] Hao Zhong and Hong Mei. 2018. Mining Repair Model for Exception-related Bug. *Journal of Systems and Software* 141 (2018), 16–31. https://doi.org/10.1016/j.jss.2018.03.046

[44] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs Documentation and Code to Detect Directive Defects. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE '17)*. IEEE Computer Society Press, 27–37. https://doi.org/10.1109/icse.2017.11