Which Exception Shall We Throw?

Hao Zhong
Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
zhonghao@sjtu.edu.cn

ABSTRACT

With the support of exception handling mechanisms, when an error occurs, its corresponding typed exception can be thrown. A thrown exception can be caught and the handling code will resolve the error (*e.g.*, closing resources), if the type of the thrown exception matches the type of the expected exceptions. Although this mechanism is critical for resolving runtime errors, bugs inside this process can have far-reaching impacts. Therefore, researchers have proposed various approaches to assist catching and handling such thrown exceptions and to detect corresponding bugs.

If the thrown exceptions themselves are incorrect, their errors will never be correctly caught and handled. Like bugs in catching and handling exceptions, wrong thrown exceptions have caused real critical bugs. However, to the best of our knowledge, no approach has been proposed to recommend which exceptions shall be thrown. Exceptions are widely adopted in programs, often poorly documented, and sometimes ambiguous, making the rules of throwing correct exceptions rather complicated. A project team can leverage exceptions in a way totally different from other teams. As a result, even experienced programmers can have difficulties in determining which exception shall be thrown, although they have the skills to implement its surrounding code. In this paper, we propose the first approach, ThEx, to predict which exception(s) shall be thrown under a given programming context. The basic idea is to learn a classification model from existing thrown exceptions in source files. Here, the learning features are extracted from various code information surrounding the thrown exceptions, such as the thrown locations and related variable names. Then, given a new context, ThEx can automatically predict its best exception(s).

We have evaluated ThEx on 12,012 thrown exceptions that were collected from nine popular open-source projects. Our results show that it can achieve high f-scores and mcc values (both around 0.8). On this benchmark, we also evaluated the impacts of our underlying technical details. Furthermore, we evaluated our approach in the wild, and used ThEx to detect anomalies from the latest versions of the nine projects. In this way, we found 20 anomalies, and reported them as bugs to their issue trackers. Among them, 18 were confirmed, and 13 have already been fixed.

ACM Reference Format:
Hao Zhong. 2022. Which Exception Shall We Throw?. In *ASE 2022.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.475/123_4

ASE 2022, 10-14 October, 2022, Ann Arbor, Michigan, United States

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

1 INTRODUCTION

In many programming languages, the exception handling mechanism is critical to resolve runtime errors [50, 67], and the bugs in exception handling have far-reaching impacts on software quality [25, 63]. When a runtime error occurs, a corresponding exception can be thrown, and its type indicates the type of the error [20]. For example, a Cassandra bug report [8] complains that the disk failure policy is not correctly activated. Figure 1a shows the handling code to activate the disk failure policy. The input of this method is an exception. As shown in Figure 1a, Line 3 checks whether the input exception is an instance of `FSError` or `CorruptSSTableException`. If it is, Line 6 calls a method to execute the code that handles the disk failure. Figure 1b shows an example that calls the handling code. In this example, the method uses a `try` statement to enclose many called methods. If one of such methods throw an exception, Line 4 catches the exception, and Line 5 calls the handling code as shown in Figure 1a. However, a disk failure is not handled, and Figure 1c shows its patch. In the buggy code, if the disk is full, Line 2 returns `null` to `directory`, and Line 4 throws `RuntimeException`. Even if the `getWriteDirectory` method is called inside a `try` statement like Figure 1b, its thrown exception will not be handled by the code in Figure 1a. As `RuntimeException` is not a subclass of `FSError` or `CorruptSSTableException`, the check in Line 3 of Figure 1a is not satisfied, and the disk failure policy is not activated. To fix this bug, in Figure 1c, Line 5 throws `FSWriteError`. As it is a subclass of `FSError`, the check in Line 3 of Figure 1a is satisfied, and the disk failure policy is activated. As illustrated in this example, thrown exceptions belong to the essential chain of the exception handling mechanism. Besides failing to activate the disk failure policy, throwing wrong exceptions can cause all critical bugs (*e.g.*, resource leaks caused by unhandled errors) that can be triggered by other components of the exception handling mechanism.

Some prior approaches [10, 38, 52] assist handling exceptions, and they recommend samples or method calls that shall appear in Line 5 of Figure 1b and the code of Figure 1a. The other prior approaches [39, 48, 49] assist catching exceptions, *e.g.*, predicting which exception shall be caught in Line 4 of Figure 1b. As the buggy code line lies in the wrong thrown exception (Line 4 of Figure 1c), none of the prior approaches can detect this bug. In summary, all the prior approaches aim to assist *catching* and *handling* exceptions.

Our contributions. Instead of another approach to assist programming in catching or handling thrown exceptions [10, 38], we propose ThEx, the first approach to predict which exceptions shall be thrown under a given programming context. The basic idea of ThEx is to learn a classification model from existing thrown exceptions in different contexts of the training set. Here, the learning features are extracted from various code information surrounding the thrown exceptions, such as the thrown locations and related variable names. Then, given a new context, ThEx can automatically predict its best exception(s).

```
public static void inspectThrowable(Throwable t){
 boolean isUnstable = false;...
 if(t instanceof FSError||t instanceof CorruptSSTableException)
    isUnstable = true;...
 if (isUnstable)
    killer.killCurrentJVM(t);
}
```

(a) The handling code

```
public int loadSaved(){...
 try{...
    in = new DataInputStream(...);...
 }catch (Throwable t){
    JVMStabilityInspector.inspectThrowable(t);...
 }finally{...}
}
```

(b) An example that calls the handling code

```
protected DataDirectory getWriteDirectory(long writeSize){
 DataDirectory directory = getDirectories().
       getWriteableLocation(writeSize);
 if (directory == null)
-  throw new RuntimeException(...);
+  throw new FSWriteError(...);
 return directory;
}
```

(c) The patch

Figure 1: CASSANDRA-11448

This paper makes the following unique contributions:

- **A new direction in researching exception handling mechanisms.** This paper introduces a new type of exception-related bugs, *i.e.*, the thrown exceptions themselves can be problematic. Like bugs in catching and handling exceptions, throwing wrong exceptions can cause serious real bugs.
- **The first approach in our new direction.** Towards our new direction, we proposed the first approach, called ThEx, that predicts which exception shall be thrown under a given programming context. It uses Eclipse JDT [2] to extract features from programming contexts, and trains a model that is the combination of Adaboost [30] and J48 [51].
- **Promising empirical evidences.** We conducted an evaluation on 12,012 thrown exceptions collected from nine popular open-source projects. Our results show that ThEx achieves high f-scores and mcc values (both around 0.8), and it works well on most types of thrown exceptions. Our evaluation also covers other interesting aspects such as the impacts of features and the effectiveness of learning from other projects.
- **Positive feedback from programmers.** We used ThEx to predict thrown exceptions in the wild. We find that ThEx can predict better exceptions than what were already written in source files. We reported 20 such cases as bugs to the corresponding developers, and 13 have already been fixed. The positive feedback from programmers highlights the importance of our work.

2 ILLUSTRATING EXAMPLE

Apache Commons `IO` [1] is a popular Java io framework. In this section, we use this project to illustrate how ThEx predicts thrown exceptions and its usage scenarios.

The procedure of ThEx. We call the code surrounding a thrown exception as its programming context. When an error occurs in a programming context, programmers shall throw its corresponding

```
public static void copyFileToDirectory(final File srcFile,
      final File destDir, final boolean preserveFileDate)
            throws IOException {
  if (destDir == null) {
     throw new NullPointerException(...);
  }
  if (destDir.exists() && destDir.isDirectory() == false) {
     throw new IllegalArgumentException(...);
  }...
  copyFile(srcFile, destFile, preserveFileDate);
}
```

(a) The `copyFileToDirectory` method

```
public static void moveFileToDirectory(final File srcFile,
      final File destDir, final boolean createDestDir)
            throws IOException {  ...
  if (!destDir.exists()) {
    throw new FileNotFoundException(...);
  }
  if (!destDir.isDirectory()) {
    throw new IOException(...);
  }
  moveFile(srcFile, new File(destDir, srcFile.getName()));
}
```

(b) The `moveFileToDirectory` method

```
try{  ...
  connection = openConnection(...);
  srcFile = ...;  destDir = ...;
  copyFileToDirectory(scrFile, destDir, true);
  connnection.close();
}catch(IOException e){
  //other handling code
  connnection.close();
}
```

(c) A possible resource leak caused by this bug

Figure 2: Our found bug

exception, so that this error can be correctly handled. Given each `throw` location, ThEx encodes its programming context into a vector. For example, from Figure 2a, it builds two vectors (Lines 4 and 7). In total, a vector records nine features of the thrown exceptions (see Table 1 for the list our defined features). For example, the vector of Line 4 includes the checked variable name (`destDir`), its type (`File`), and the checked constant (`null`). The type of a thrown exception is extracted as the true label of a vector. In this example, the labels of the above two vectors are `NullPointerException` and `IllegalArgumentException`, respectively. Taking these labeled vectors as the training data, we then train a classification model. After the model is built, ThEx predicts which exception shall be thrown in each programming location. With the predictions, ThEx can have the following two usage scenarios:

Scenario 1. Assisting programming. ThEx is able to recommend suitable exceptions, since it encodes the consensuses into its mined models. Although it takes time to train our model, once it is trained, the prediction is sufficiently fast to be used in assist programming. ThEx is useful for programmers who have skills to implement programming tasks but are unfamiliar with the local consensus in throwing exceptions. For example, in Figure 2, as `destDir` is a method argument and it is checked against `null`, a new member of the IO project can be confused in throwing `IllegalArgumentException` or `NullPointerException`. The consensus of this project is that `IOException` and its subclasses are more suitable in these programming contexts, but the new member may not know this consensus. For example, in Figure 1, `RuntimeException` is vaguely defined, and `FSWriteError` is defined by `cassandra`. When a programmer does not know the handling code in Figure 1a, this programmer

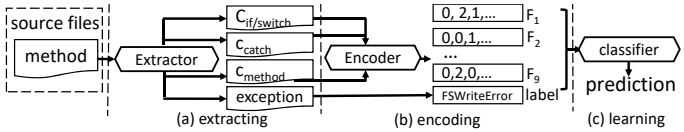

Figure 3: The overview

```
1 String calculateRawEncoding(final String bomEnc, final String
      xmlGuessEnc, final String xmlEnc) throws IOException{...
2   if (bomEnc.equals(UTF_8)) {
3     if (xmlGuessEnc != null && !xmlGuessEnc.equals(UTF_8)) {...
4       throw new XmlStreamReaderException(...);
5     }
6     return bomEnc;
7   }...
8 }
```

Figure 4: A method that throws `XmlStreamReaderException`, a subclass of `IOException`

is unlikely to throw the correct exception, even if this programmer is experienced and all the programming contexts are already written. As programmers can join and leave a project, especially in open source communities [43], they often are unfamiliar with the consensus of thrown exceptions. Wrong thrown exceptions affect the exception handling mechanism, and they often do not hinder programmers from implementing a programming task. As a result, the other approaches [48, 49] also assist writing exceptions, under the assumption that complete programming contexts are available. As both commercial companies and open source projects have strict criteria to select programmers, it is reasonable to assume that their programmers have the skills to implement assigned programming tasks, and the programming contexts of thrown exceptions shall be complete. Even if programmers have difficulties in fully implementing a task, our results in Section 4.5 show that ThEx can make reasonably good predictions, even if programmers cannot written full programming contexts.

Scenario 2. Detecting bugs in thrown exceptions. In this example, for the two locations in Figure 2a, ThEx predicts that they shall throw `IOException` and `FileNotFoundException`, respectively. After we reported this problem [3], it was confirmed and fixed. ThEx makes these predictions, because its model is trained on many thrown exceptions and most other locations of this project throw the two exceptions, when destination files are illegal. For example, the method in Figure 2b also checks a `destDir` variable that calls similar methods (*e.g.*, `exists()`). This bug can cause unhandled errors. For example, Figure 2c shows a call site of the `copyFileToDirectory` method. If the programmer is familiar with other methods of `IO`, this program can blindly believe that the `copyFileToDirectory` method also throws `IOException` and its subclasses, especially when the `copyFileToDirectory` method declares that it throws `IOException`. As a result, this programmer can catch `IOException` as shown in Line 6 of Figure 2c. When destination files are illegal, as thrown exceptions are different from the excepted ones, Line 6 fails to catch the error. For simplicity, we ignore the handling code in Line 7, but this programmer can write complicated code to handle the error, but such code will not be executed. Programmers are used to catching the same type of exceptions when the problems belong to the same type. As introduced in Section 1, this behavior has caused uncaught exceptions in real code. Besides the direct influence, unhandled errors can cause other problems. In this example, Line 5 opens a connection. When destination files are illegal, this connection is not closed, since Line 5 and Line 8 are both bypassed. As a result, it causes a resource leak.

When ThEx is used to detect bugs in thrown exceptions, its basic idea is in line with other anomaly detections [19]. For example, after Wasylkowski *et al.* [62] mine usage models of objects, they report the violations of their mined models as bugs. As an analogy, ThEx mines the majority of thrown exceptions as its model, and reports its violations as bugs. In summary, the second scenario is to detect bugs in thrown exceptions. It works as other anomaly detectors do, and reports bugs when its predictions are inconsistent with what were written in source files. In our evaluations in Section 4, some predictions are different from the true labels. Although such predictions are considered as false, some false predictions are violations, indicating bugs in thrown exceptions.

3 APPROACH

As shown in Figure 3, ThEx has three phases: extracting (Section 3.1), encoding (Section 3.2), and learning (Section 3.3).

Problem Definition. From the programming context of each `throw` location (l), we extract a vector, $\vec{l_c} = \{f_1, \ldots, f_9\}$. Table 1 defines the nine features of programming contexts. We then reduce our target problem to a classification problem:

$$classifier(\vec{l_c}) = l_e \quad (1)$$

where l_e is the exception type that is thrown at l. In the training data, l_e is extracted from thrown exceptions that were written by programmers. Our trained classification model can predict which exception shall be thrown at a throw location, if its programming context is given.

Java and many other languages (*e.g.*, C++) allow both checked and unchecked exceptions. As unchecked exceptions are not checked at compile-time, programmers are allowed to throw other unchecked exceptions. Figure 2 shows an example of unchecked exceptions. For checked exceptions, a method can declare multiple exceptions, and each exception can have more subclasses. It can become difficult to determine which exception shall be thrown. For example, the method in Figure 4 declares only a checked exception, but Line 4 throws another exception. It is legal to throw this exception, since `XmlStreamReaderException` is a subclass of `IOException`. In summary, our target problem is valid for both checked and unchecked exceptions.

3.1 Extracting Thrown Exceptions

From each `throw` statement, ThEx extracts a pair $\langle t, c \rangle$, where t denotes the type of the thrown exception and c denotes the context of t. To extract the pairs, ThEx builds Abstract Syntax Trees (ASTs) from source files, and analyzes ASTs to collect features and labels:

3.1.1 Extracting exception types. To extract t (the label) from a `throw` statement, ThEx analyzes four cases:

1. Instance creations. In source files, `throw` keywords are often followed by the creations of exceptions. ThEx resolves the types of created exceptions as t.

2. Variables. If a `throw` keyword is followed by a variable, ThEx resolves the type of the variable as t.

```
1 public Runnable associateWith(Runnable runnable) {
2     if (runnable instanceof Thread) {...
3         throw new UnsupportedOperationException(msg);
4     }
5     return new SubjectRunnable(this, runnable);}
```

(a) An exception thrown from `if` statements

```
1 public Object invoke(...) throws NoSuchMethodException,
      IllegalAccessException, InvocationTargetException {
2   try{...
3   }catch (ExecutionException e) {
4     throw new NoSuchMethodException();
5   }}
```

(b) An exception thrown from `catch` statements.

```
1 private HashingPasswordService assertHashingPasswordService(
      PasswordService service) {
2     if (service instanceof HashingPasswordService) {
3         return (HashingPasswordService) service;
4     }
5     String msg = ...;
6     throw new IllegalStateException(msg);}
```

(c) An exception thrown from the end of a method.

Figure 5: The locations of thrown exceptions

3. Method invocations. If a `throw` keyword is followed by a method invocation, ThEx extracts the return type of the method as t.

4. Cast expressions. If a `throw` keyword is followed by a `cast` expression, ThEx extracts casted types as t of these thrown exceptions.

3.1.2 Extracting contexts. Table 1 shows our features that are extracted from the contexts of thrown exceptions. Column "Context" shows our considered programming contexts. Row "The direct parent" lists the direct parent of a thrown exception. If an exception is thrown in an `if` statement within a `catch` clause, this feature is set to `if`. If F_1 is `if/switch`, ThEx extracts F_2 to indicate whether method arguments are checked by the `if` or `switch` statements. If F_1 is `catch`, ThEx extracts the caught exception as F_3. Exceptions can be propagated [28]. For the thrown exception in Line 4 of Figure 5b, ThEx extracts `ExecutionException` as the caught exception. We select this feature, because caught exceptions have some connections to thrown exceptions. As shown in Figure 1, handling exceptions often requires global features from call chains, but it typically requires only local features to determine which exceptions shall be thrown at a code location. As a result, if an exception is thrown inside multiple statements, F_1 considers only the direct parent statement. The information from call chains provides more inputs, but their impacts are mixed. Due to the polymorphism and other issues, if we use static analysis, the features from call chains can contain noises. It needs advanced techniques to handle the mixed impacts.

As shown in Figure 5c, some exceptions are thrown from other locations, and most of them are thrown from the ends of methods. These methods often contain `if` or `switch` statements before the `throw` statements. As these statements determine the condition of thrown exceptions, ThEx extracts F_2 from such `if` or `switch` statements. As exceptions thrown from `if/switch` statements, F_2 are boolean values indicating whether method arguments are checked.

As shown in Table 1, ThEx extracts variable names, constant values, called types and methods as F_4 to F_7. When it extracts the four features, ThEx analyzes different scopes according to the locations of thrown exceptions. In particular, for exceptions thrown in `if/switch` statements and at the ends of methods, it analyzes the scope of checked conditions, because the variables, constants and methods in checked conditions are useful to determine the types of exceptions. For example, in Figure 5a, the comparison between two types determines whether the argument is supported or not. As a result, from Line 2 of Figure 5a, it extracts `runnable` as F_4, and `Runnable` and `Thread` as F_6. It does not extract F_5 and F_7 from Line 2, because this line does not compare the variable against any constants or call any methods. For exceptions thrown in `catch` clauses, ThEx extracts F_4 to F_7 from the bodies of `try` statements. For example, in Figure 5b, it extracts the features (F_4 to F_7) from Line 3.

Table 1: Our features.

Context	Feature	Context
The direct parent `dp`	F_1	`if/switch`, `catch`, or `other`
`if/switch`	F_2	arguments are checked
`catch`	F_3	caught exceptions
`other`	F_2	arguments are checked
Elements in `dp`	F_4	variable names
	F_5	constant values
	F_6	called types
	F_7	called methods
Element outside `dp`	F_8	exceptions in method headers
	F_9	exceptions thrown by other lines

if/switch, $F2$: arguments are checked in this if or switch statement; other, $F2$: arguments are checked before this exception is thrown.

A method header can declare its thrown checked exceptions. ThEx extracts such exceptions as F_8. For example, from Line 1 of Figure 5b, ThEx extracts `NoSuchMethodException`, `IllegalAccessException`, and `InvocationTargetException` as F_8. As they are checked exceptions, when programmers call the `invoke` method, they have to handle the three exceptions. However, it is legal for the code inside the `invoke` method to throw any other exceptions. As a result, even if exceptions are listed in F_8, programmers may not determine which exceptions shall be thrown at a given code location, and ThEx can predict other exceptions to throw than those listed in F_8.

A method can throw more than one exception. For a thrown exception, ThEx extracts the types of other thrown exceptions as its F_9. For example, F_9 of the thrown exception in Line 4 of Figure 5b includes `IllegalAccessException` and `InvocationTargetException`.

3.2 Encoding Features

The second step is to encode thrown exceptions into vectors. F_1, F_2, F_3, F_8, and F_9 have limited values. ThEx encodes them into nominal values. F_4 to F_7 have many values, especially when exceptions are thrown from `catch` clauses. ThEx considers only the top ten items.

In natural language processing [26], words are often translated to their lower cases. The lower case and the upper case of a natural language word often denote the same item, and treating them as two words can reduce the frequency of the item. F_4 is like words in natural languages, in that variables are often named in an ad hoc way. If two variable names across methods are the upper case and the lower case of a word, they often refer to the same thing. In addition, variable names can be written in camel cases, and the combination of their words is similar to phrases in natural languages. Due to the above consideration, ThEx splits variable names into words by capitalized characters, and transfers their split

words into lower cases. After splitting, some words become too short to convey meanings. We remove a split word, if it contains fewer than three characters.

As Java is case sensitive, the upper case and lower case of a method/class name or a constant value denote different things. As these names and values are extracted from source code, programmers will not accidentally write wrong cases either. As a result, we do not translate the features from F_5 to F_7 into lower cases. Another difference is that these names and values can be written in camel cases, we leave them as they are, because the combinations of camel cases denote different methods, classes or values. Here, variable names (F_4) are more ad-hoc than constants (F_5) and method names (F_7), since variable names are usually defined locally. As a result, we handle them differently.

For F_4 to F_7, we encode their items with Tf-Idf [60]. Tf-Idf is a classical technique to encode words into vectors by their Tf and Idf frequencies. A document of F_4/F_7 is all the split variable names or fully qualified names of called methods that appear in the context of a thrown exception. A corpus is the documents of all the thrown exceptions. Some recent techniques [47] can present more meaning comparison between words. Their models are trained on Google newspapers, but our extracted words are code names that do not appear in their training set. As a result, we do not choose them to encode our features.

3.3 Learning Model

ThEx combines two classification techniques: J48 [51] and AdaBoost [30]. J48 is an algorithm to generate decision trees, and is a supervised classification technique. A built decision tree classifies instances by its if-else nodes. Each interior node denotes a check on a variable, and each leaf denotes a class. Adaboost is a meta-level learning technique that combines the outputs of weak classifiers into a weighted sum to predict the final output. The training set of ThEx is a set of labeled data $(\vec{f_i}, l_i)$, where $\vec{f_i}$ is the feature vector, and l_i is the label of a node. Adaboost repeatedly tunes its weights during the training process. We use $d_t(i)$ to denote the weight of the ith instance of the training data in the tth iteration. In the next iteration $t+1$, the weight is updated as follows:

$$d_{t+1}(i) = \frac{d_t(i) exp(-\alpha_t h_t(\vec{f_i}) l_i)}{z_t} \tag{2}$$

where $\alpha_t = \frac{1}{2} ln(\frac{1-\varepsilon_t}{\varepsilon_t})$ is the weight updating parameter, $h_t(\vec{f_i})$ is the prediction on feature vector $\vec{f_i}$, and z_t is a normalization factor that ensures that the all new weights sum to one. Here, ε_t is the error in the current model over the training set. After imposing cost $cost(i)$ on the ith instance, the above equation is modified to:

$$d_{t+1}(i) = \frac{d_t(i) exp(-\alpha_t cost(i) h_t(\vec{f_i}) l_i)}{z_t} \tag{3}$$

Adaboost can be integrated with various classifiers, and can lead to minor differences [69]. We select J48, because its result is the best after we tried other classifiers of WEKA and our evaluation results show that its f-scores are already around 0.8. As we discussed in Section 7, more advanced classification techniques can improve our effectiveness, which we leave for other researchers.

Our basic idea is to learn the knowledge of throwing exceptions from known instances. In a project, some exceptions are rarely used, and it is infeasible even for programmers to learn their usages. To handle the problem, we remove an exception if its instances are fewer than ten. For each instance $(\vec{f_i}, l_i)$, we predict l_i based on $\vec{f_i}$. As we introduced in Section 3.1, $\vec{f_i}$ does not contain any information to leak the label l_i.

4 EVALUATION ON BENCHMARK

We build the extractor of ThEx upon JDT [2], because it is the Java compiler of Eclipse. JDT parses each source file into an abstract syntax tree (AST), and allows customized visitors to traverse the tree. Based on JDT, our visitors traverse the ASTs to collect our features. We build the encoder and the miner of ThEx on WEKA [34], because WEKA is a popular mining framework. In particular, our Tf-Idf encoder is built on the `StringToWordVector` filter, and our miner is built on the J48 and Adaboost classifiers of WEKA. We use the default settings of the two classifiers. With ThEx, we conducted evaluations to explore the following research questions:

(RQ1) What is the overall effectiveness (Section 4.3)?
(RQ2) What are the impacts of exception types (Section 4.4)?
(RQ3) What are the impacts of features (Section 4.5)?
(RQ4) How effective is ThEx, if its model is learned from other projects (Section 4.6)?

Our project website is as follows: https://github.com/drhaozhong/thex

4.1 Benchmark

Table 2 shows the benchmark of our evaluation. Throwing which exceptions is a programming issue. We select libraries, since it is feasible to discuss programming issues with library developers. In the contrast, it is odd for end users to discuss programming issues with application developers. We select the nine libraries, because they are popular and under careful and active maintenance. In total, the nine libraries have more than two million lines of code, and contain 12,012 thrown exceptions. Column "Location" lists the locations of thrown exceptions. In total, 69.7% exceptions are thrown inside `if` statements, and 15.4% exceptions are thrown inside `catch` clauses. Column "Exception" lists the types of thrown exceptions. In total, the nine projects throw 399 types of exceptions, and 31.3% of them are frequent. We consider that a type is frequent, if at least ten exceptions of this type are thrown from a project. Among the frequent ones, Subcolumn "J2SE" lists the types defined by J2SE. In total, about half of frequent types are defined by J2SE. Some J2SE exception types appear in most projects. For example, in six out of the nine projects, the most popular exception type is `java.lang.IllegalArgumentException`. Column "Frequent location" shows the locations that throw frequent exceptions. In total, the frequent exceptions account for more than 90% locations.

Despite of a different research angle, as the prior approaches [48, 49] did, we use the exception code that is written by programmers as the true labels. Our features in Table 1 do not contain any indirect hints of the true labels, and do not leak them as our inputs.

Table 2: Our subjects.

Project	LOC	Location					Exception			Frequent location
		if	switch	catch	other	total	J2SE	frequent	total	
asm	43,552	267 (69.7%)	76 (19.8%)	10 (2.6%)	30 (7.8%)	383	4(66.7%)	6(46.2%)	13	367
commons-io	29,988	369 (84.8%)	3 (0.7%)	16 (3.7%)	47 (10.8%)	435	7(77.8%)	9(40.9%)	22	394
itext	225,099	1,140 (79.7%)	40 (2.8%)	198 (13.8%)	52 (3.6%)	1,430	7(46.7%)	15(32.6%)	46	1,331
jfreechart	133,284	430 (87.4%)	4 (0.8%)	46 (9.3%)	12 (2.4%)	492	4(57.1%)	7(46.7%)	15	464
jmonkey	209,110	1,035 (62.6%)	153 (9.3%)	313 (18.9%)	152 (9.2%)	1,653	8(53.3%)	15(41.7%)	36	1,572
lucene	1,727,376	2,835 (71.2%)	143 (3.6%)	540 (13.6%)	466 (11.7%)	3,984	18(62.1%)	29(34.5%)	84	3,818
pdfbox	157,676	313 (68.0%)	25 (5.4%)	104 (22.6%)	18 (3.9%)	460	7(53.8%)	13(43.3%)	30	425
poi	398,693	1,689 (61.4%)	198 (7.2%)	519 (18.9%)	343 (12.5%)	2,749	11(45.8%)	24(25.8%)	93	2,546
shiro	34,209	297 (69.7%)	1 (0.2%)	109 (25.6%)	19 (4.5%)	426	3(42.9%)	7(11.7%)	60	296
total	2,958,987	8,375 (69.7%)	643 (5.4%)	1,855 (15.4%)	1,139 (9.5%)	12,012	69(55.2%)	125(31.3%)	399	11,213

if: exceptions thrown inside `if` statements; switch: exceptions thrown inside `switch` statements; catch: exceptions thrown inside `catch` clause; and other: exceptions thrown from other locations; J2SE: exceptions defined in J2SE; and frequent: exception types with more than 10 thrown locations.

Table 3: Overall result.

Project	Precision	Recall	F-score	MCC	ROC
asm	0.883	0.883	0.883	0.831	0.967
commons-io	0.885	0.886	0.884	0.843	0.979
itext	0.842	0.846	0.843	0.819	0.969
jfreechart	0.927	0.927	0.925	0.874	0.983
jmonkey	0.801	0.803	0.800	0.768	0.955
lucene	0.845	0.847	0.845	0.814	0.965
pdfbox	0.871	0.866	0.865	0.836	0.958
poi	0.722	0.724	0.722	0.671	0.920
shiro	0.708	0.709	0.704	0.604	0.890

4.2 Measures

By comparing true labels with outputs, we classify results into false negatives (FNs), false positives (FPs), true negatives (TNs), and true positives (TPs). We then calculate the following metrics:

$$precision = \frac{TP}{TP + FP} \tag{4}$$

$$recall = \frac{TP}{TP + FN} \tag{5}$$

$$fscore = \frac{2 \times precision \times recall}{precision + recall} \tag{6}$$

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{7}$$

Besides the above precisions, recalls, f-scores and Matthews correlation coefficient (MCC), we use the area under a receiver operating characteristic curve (ROC) as an addition measure. F-score shows the balance between precision and recall; MCC shows the robustness of a classifier on imbalanced data; and ROC shows the robustness of a classifier on threshold settings. They are indeed complementary. For all the three measures, a value closer to one indicates a better classifier. All the above measures are calculated by WEKA. In particular, for each type of predicted exceptions, it calculates the above measure values, and further calculates weighted averages of all the types. In our research questions, we report their weighted averages.

4.3 RQ1. Overall Effectiveness

4.3.1 Setup. In this research question, we evaluated the overall effectiveness of ThEx. To ensure the reliability of our results, we conducted a ten-fold cross validation on each project of our benchmark. In each fold, the thrown exceptions are randomly divided into ten portions of the same size. Among them, nine portions construct the training set, and the remaining one is used as the testing set.

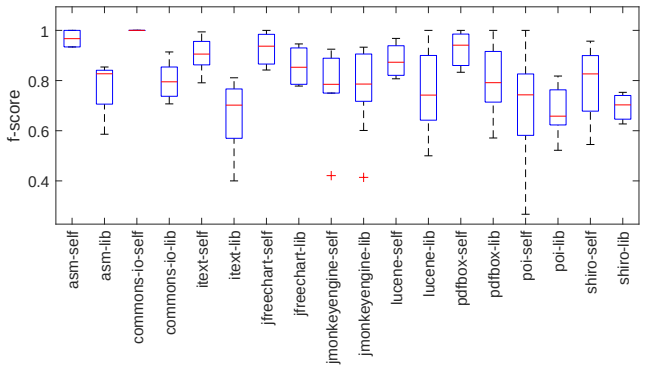

Figure 6: The f-score values of thrown exception types.

4.3.2 Result. Table 3 shows the results. The results are quite positive. In seven out of the nine projects, the f-scores of ThEx are more than 0.8. In `jfreechart`, the f-score is 0.925, which is close to a perfect classifier. Column "Instance" shows that `lucene` has more than 3,000 instances. Even for this project, our f-score is 0.845. Table 2 shows that about 30% exception types are frequent. Exception types, especially those defined by J2SE, can be considered as API classes, and the frequencies of API calls are typically imbalanced [68]. For imbalanced benchmarks, MCC is a more reliable measure than f-score [13]. Column "MCC" shows that the MCC values of the seven projects are also close to one, and it reconfirms that the results of the seven projects are quite positive.

The f-scores of `pdfbox` and `shiro` are still reasonably high (>0.7). Columns ROC and F-score indicate the interplay between precision and recall at various threshold settings. The values are close to 1, indicating ThEx achieved a reasonable balance between them. Our f-score values vary among projects, since some projects have more training data and throw exceptions more consistently than others. A higher f-score value indicates that its exceptions are thrown more consistently, and can indicate better code quality. Our observations lead to a finding:

Finding 1. ThEx achieved 0.7 to 0.8 f-score and MCC values in predicting correct thrown exceptions.

The high f-score and MCC values indicate that ThEx makes accurate recommendations, once programmers write complete contexts. This is useful for skillful programmers who are unfamiliar with the local consensus on how to throw exceptions. For infrequent

Table 4: The impacts of features.

Δfeature	asm	common-io	itex	jfreechart	jmonkey	lucene	pdfbox	poi	shiro	reduce
-F1	**-0.015**	**-0.013**	0.009	**-0.010**	-0.005	**-0.010**	0.000	0.002	**-0.018**	5
-F2	**-0.012**	**-0.012**	-0.003	-0.008	-0.005	**-0.019**	-0.001	**-0.032**	**-0.020**	5
-F3	-0.008	0.005	0.006	-0.002	0.002	-0.008	**0.032**	0.003	**0.026**	-
-F4	-0.008	**-0.031**	**0.020**	-0.007	-0.001	**-0.011**	-0.002	**-0.017**	0.007	4
-F5	0.002	**0.010**	0.007	0.001	-0.004	-0.005	-0.005	**-0.018**	-0.001	2
-F6	-0.006	0.001	-0.003	-0.006	-0.003	**-0.010**	-0.002	**-0.027**	-0.005	2
-F7	-0.008	**-0.010**	0.001	0.000	-0.008	**-0.040**	0.000	**-0.015**	**0.026**	3
-F8	-0.009	-0.008	**-0.013**	0.002	**-0.018**	**-0.048**	**-0.078**	**-0.031**	-0.001	5
-F9	**-0.021**	-0.007	-0.007	**-0.018**	**-0.034**	**-0.070**	**-0.021**	**-0.049**	**-0.036**	7

exceptions, a heuristic approach can work better, and we discuss this issue in Section 7.

4.4 RQ2. The Impacts of Exception Types

4.4.1 Setup. Programmers can define their own exceptions to handle a specific type of errors. We call such exceptions as self-defined exceptions. In this research question, we explore how consistent programmers throw self-defined exceptions and library exceptions. In addition, to present more detailed results of each category, we draw plot boxes, so that it is feasible to present the distributions of all exception types.

4.4.2 Result. Figure 6 shows the results. For the `asm` project, `asm-self` denotes the exceptions defined by `asm`, and `asm-lib` denotes the exceptions defined by the libraries of `asm`. The other projects have similar names in the horizontal axis. Except `poi-lib`, the medians of all the situations are more than 0.7. This result indicates that ThEx works well on most exception types, albeit frequent or less frequent ones. Except `jmonkeyengine`, all the other projects have higher f-score values for self-defined exceptions than those for exceptions in libraries. Even for `jmonkeyengine`, the minimum and the third quartile of self-defined exceptions are more than those of the libraries. The observations lead to the following finding:

Finding 2. ThEx achieves better f-score values when it predicts self-defined exceptions than those defined in libraries.

The result indicates that programmers have more consensus on how to throw their self-defined exceptions than those defined in their libraries. If wrong exceptions are thrown in libraries, client programmers cannot fix them directly, but have to report them to library developers. Although the process is not straightforward, our results in Section 5 show that libraries developers pay attention to wrong thrown exceptions, and fixed our reported bugs.

4.5 RQ3. The Impacts of Features

4.5.1 Setup. As a reference, the prior approaches [48, 49] also need complete programming contexts to predict which exceptions shall be caught. Still, it is interesting to explore the impacts of missing features. We use the results in RQ1 as the baselines. With the same setting, for each project, we iteratively remove one feature, and conduct ten-fold cross validation to collect the f-scores. We compare the f-scores with RQ1 to show the impacts of removing a feature.

4.5.2 Result. Table 4 shows the results. In this table, we highlight a delta, if it is greater or equal to 0.01. If removing a feature reduces a f-score by at least 0.01, we consider this feature as a key feature. We have to choose a small threshold (0.01) to determine key features, because Table 4 shows that the impact of a feature is typically small. Even the worst case reduces the f-score by only 0.049, and after the reduction, the f-score is reasonably high (0.673). Indeed, removing features can even increase f-scores, in that in some projects, the quality of these features can be low and misleading. Due to the above consideration, instead of the values of deltas, we use numbers of influenced projects to show the impacts of features.

Column "reduce" shows reduced cases. The results show that F_9 ranks the first: when a method throws more than one exception, the exceptions thrown from other lines are strong indicators to which exceptions shall be thrown. The other important features are F_1, F_2, and F_5. The results show that the thrown locations, caught exceptions, and exceptions defined in method headers are also strong indicators. The observations lead to a finding:

Finding 3. The scopes (F_1), checked arguments (F_2), exceptions thrown in method headers (F_8), and exceptions defined in other lines (F_9) have more visible impacts than other features.

The results indicate that ThEx can assist selecting which exceptions to throw even if programmers do not have the sufficient skills to write full programming contexts. Feature selection techniques [44] can identify redundant features, and refine our model.

4.6 RQ4. Cross-Project Learning

4.6.1 Setup. ThEx learns from thrown exceptions that are written in a project, but some new projects may not throw many exceptions. To enable ThEx for new projects, it is interesting to explore the effectiveness of cross-project learning. In this research question, we use the data of a project as the testing set, and the data from all the other eight projects as the training set.

As we did in RQ1, if the instances of an exception type are fewer than ten, we remove them from our benchmarks, in that there are no sufficient instances for mining. The training set has no instance of those self-defined exception types. As their instances do not appear in the training set, the trained model cannot predict such exceptions. For those exceptions, we remove their instances from the testing set. For all the strategies, we conduct ten-fold cross validations as we did in RQ1 to collect the measure values.

4.6.2 Result. Table 5 shows the results. Each row denotes a project that is used as the testing data. For example, the first row is the result, when `asm` is used as the testing set and the other projects are used as the training set. To help the comparison, in this table, we present the deltas over Table 3. The numbers inside brackets denote that values are reduced. The results show that on all the projects, learning from all other projects is less effective than learning from the same projects. For `asm`, `itext`, `lucene`, `pdfbox`, and `shiro`,

Table 5: The results of learning from other projects.

Project	ΔP	ΔR	ΔF	ΔM	ΔRO	ΔE
asm	-0.198	-0.204	-0.211	-0.356	-0.163	-2
commons-io	-	-0.174	-	-	-0.094	-3
itext	-0.240	-0.234	-0.266	-0.426	-0.151	-8
jfreechart	-0.100	-0.105	-0.107	-0.236	-0.052	-3
jmonkey	-0.234	-0.310	-0.340	-0.368	-0.182	-6
lucene	-0.140	-0.189	-0.182	-0.234	-0.103	-17
pdfbox	-0.230	-0.263	-0.259	-0.356	-0.116	-6
poi	-	-0.156	-	-	-0.114	-13
shiro	-0.057	-0.114	-0.122	-0.288	-0.141	-4

Δ: the changes over Table 3; -: a value is not a number.

their f-scores are reduced to around 0.6. As two extreme cases, the f-score of `jmonkey` is reduced to 0.460, but the f-score of `jfreechart` is still high (0.818). For `commons-io` and `poi`, their precisions become not a number. After manual inspection, we found that their predictions do not include some types of exceptions. For these types of exceptions, their true positives and false positives are both zero, and according to Equation 4, their precisions become not a number. Column "ΔE" shows the number of predictable exception types. When it learns from all other projects, although the training sets become much larger, Column "ΔE" shows that for all the projects, their predictable exception types become fewer, in that other projects have no instances of self-defined exception types. The above results lead to a finding:

Finding 4. When the training data come from all other projects, f-scores of six projects are reduced to around 0.6, and fewer types of exceptions can be predicted.

When constructing our dataset, we did not select projects that are developed by overlapped leaders or select projects from the same companies. If other researchers do that, their results can be better than what we reported. Still, ThEx already achieved high f-score values on some projects (*e.g.*, `jfreechart`) under our setting. Besides the above workaround, more advanced techniques can be useful (see Section 7 for more discussions).

5 EVALUATION IN THE WILD

In Section 4, when we construct our benchmark, we use what were written in source files as the gold standards. Although it thus allows us to calculate the f-score values as shown in Table 3, this gold standard is not fully correct. Like most other gold standards, in the wild, there are anomalies, and researchers have used anomalies to detect bugs in source files [62] or network intrusions [55]. Motivated by their work, we use ThEx to detect anomalies in thrown exceptions.

5.1 Setup

In this section, we use ThEx to detect anomalies from the latest versions of all the projects in Table 2. To locate those anomalies, we dump the inconsistent predictions. For each pair of inconsistent predict and label, we manually inspect them to determine whether such anomalies indicate bugs in source files. In particular, for a given thrown exception in source files and a predicted exception that shall be thrown, we use the following criteria one by one to determine which one is better:

1. Which exceptions are thrown under similar programming contexts? For a thrown exception, we search for its similar programming contexts by our defined features, and check their thrown exceptions. If our prediction is identical with the majority, we consider that our prediction is better.

2. Which exceptions are thrown by similar API methods? As all our subjects are libraries, if a wrong exception is thrown by an API method, we locate its similar API methods (*e.g.*, an API method with the identical name, but its parameters are different), and check their thrown exceptions. If our prediction is identical with the majority, we consider that our prediction is better.

3. Which exception is more suitable according to their documents? We read the documents and the names of exceptions to learn which is the best under a given programming context. If we the contexts are similar, we consider that an exception is better if it is more specific. For example, we consider that `IllegalArgumentException` is better than `Exception`, since `IllegalArgumentException` is a subclass of `Exception`.

To obtain the feedback from developers, we reported twenty better predictions to their developers. For example, `itext` defines `PdfException`, and it defines more than two hundred constant strings. After we read these strings, we find that `itext` uses such strings to distinguish different types of problems. For example, a string is "Unsupported XObject type", and another string is "Cannot flush object". Typically, programmers in other projects will throw `UnsupportedOperationException` and `IOException` for the above two problems, respectively. As even some `itext` programmers would rather not throw the same exception for different problems, our trained model sometimes predicts to replace `PdfException` with other exceptions. As such recommendations can violate the design principles of `itext`, we do not consider them as better predictions.

5.2 Feedback from Library developers

Table 6 shows our reported 20 bugs, and 13 of them were fixed. We next introduce some samples:

1. The `commons-io` project. As described in our bug report [7], the `FileUtils` class throws inconsistent exceptions:

```
public ...toByteArray(final InputStream input,final int size){
 if (size < 0) {
   throw new IllegalArgumentException(...);
 }...
 if (offset != size) {
   throw new IOException(...);
 }...}
```

This bug report is marked as fixed, after we report it. When fixing this bug, a programmer left a message:

> This case in particular is interesting because the exception is thrown because the expected input does not match the actual file, so either the input is wrong (IAE) OR something went wrong while reading the file (IOEx). So either exception might be valid here. One general rule could be that IOEx all come from the JRE...

As we introduced in Section 2 and illustrated in this bug report, it is acceptable to throw multiple exceptions at many code locations. However, in a single project, programmers shall consistently throw the same type of exceptions, when the encountered problems belong to the same type. The `common-io` programmers agree that their code shall be consistent to throw exceptions, and they pointed out that this consensus roots from JRE.

2. The `pdfbox` project. As described in our bug report [9], the `Type1Parser` class throws inconsistent exceptions:

```
private void parseASCII(byte[] bytes){
 if (bytes.length == 0){
```

```
3    throw new IllegalArgumentException("byte[] is empty");
4   }
5   if (bytes.length < 2 || ...)){
6    throw new IOException("Invalid start ...");
7   }...
8  }
```

Lines 2 and 4 throw different exceptions, when the lengths of `bytes` are checked. This bug report is confirmed and fixed after we reported it. A programmer left a message:

> I agree it's better to have a checked exception, thanks for pointing that out.

3. The `asm` project. As described in our bug report [6], the `InstructionAdapter` class has the following method:

```
1 public void invokevirtual(...){
2  if (api < Opcodes.ASM5) {
3    if (isInterface) {
4      throw new IllegalArgumentException(...);
5    } ... }... }
```

ThEx predicted that Line 4 shall throw `UnsupportedOperationException` in that all the other methods of this class throw this exception. This bug report is fixed after we reported it.

4. The `itext` project. As described in our bug report [4], the `MatrixUtil` class has a method:

```
1 public static void makeTypeInfoBits(...) ... {
2  if (!QRCode.isValidMaskPattern(maskPattern)) {
3    throw new WriterException("Invalid mask pattern");
4  }
5 ...}
```

ThEx predicted that Line 3 shall throw `IllegalArgumentException`, because other methods throw this exception. We submitted a pull request, and it is merged.

5. The `poi` project. As we described in our bug report [5], the `SXSSFCell` class has the following method:

```
1 private boolean convertCellValueToBoolean() { ...
2  switch (cellType) {
3  case BOOLEAN:
4    return getBooleanCellValue();
5  ...
6  default: throw new RuntimeException("Unexpected...");
7 }
```

ThEx predicted that Line 6 shall throw `IllegalStateException` in that other methods throw this exception. This bug is fixed, after we reported it. After reading discussions of our bug reports, we find that even experienced programmers can disagree with which exceptions shall be thrown. The observation highlights the significance of an automatic tool like ThEx. Although it is reasonable to throw either exception in some cases, once it is decided, all the code locations of a library shall throw the decided one. Otherwise, its client code can fail to catch thrown exceptions.

Our found bugs confirm that even professional programmers can be unfamiliar with the consensus of throwing exceptions. These bugs can be avoided, if ThEx recommends the correct thrown exceptions at the development phase. Our target bugs are across the borderline of libraries and clients. For example, the Cassandra bug in Section 1 involves `RuntimeException` that is defined in J2SE and `FSWriteError` that is defined by Cassandra itself. When a wrong exception is thrown in library code, client programmers have to fix them as workarounds [59], since they cannot access library code. We notice that many bug reports on libraries are silently ignored, since library developers have heavy workload. As our subjects are libraries, most of our found bugs reside in the library side.

Table 6: Our reported bugs

URL	Status
https://issues.apache.org/jira/browse/IO-661	fixed
https://issues.apache.org/jira/browse/IO-696	fixed
https://issues.apache.org/jira/browse/IO-704	not a problem
https://issues.apache.org/jira/browse/IO-705	fixed
https://github.com/itext/itext7/pull/51	fixed
https://github.com/jMonkeyEngine/jmonkeyengine/issues/1313	fixed
https://issues.apache.org/jira/browse/LUCENE-9296	open
https://issues.apache.org/jira/browse/LUCENE-9343	open
https://issues.apache.org/jira/browse/PDFBOX-5084	duplicated
https://issues.apache.org/jira/browse/PDFBOX-5080	fixed
https://github.com/jfree/jfreechart/issues/205	fixed
https://bz.apache.org/bugzilla/show_bug.cgi?id=64274	won't fix
https://bz.apache.org/bugzilla/show_bug.cgi?id=64964	fixed
https://bz.apache.org/bugzilla/show_bug.cgi?id=65085	fixed
https://bz.apache.org/bugzilla/show_bug.cgi?id=65084	open
https://issues.apache.org/jira/browse/SHIRO-751	fixed
https://issues.apache.org/jira/browse/SHIRO-810	open
https://gitlab.ow2.org/asm/asm/-/issues/317922	fixed
https://gitlab.ow2.org/asm/asm/-/issues/317930	fixed
https://gitlab.ow2.org/asm/asm/-/issues/317931	fixed

However, even if these bugs in libraries do not introduce visible and serious bugs as they do in client code, our results show that library developers are willing to fix wrong thrown exceptions, since they have far-reaching impacts on many clients.

6 THREATS TO VALIDITY

The threat to internal validity includes our true labels in Section 4. We consider thrown exceptions that are written in source files as true labels, but our results in Section 5 show that ThEx can predict better exceptions than those labels. This threat can be reduced by inspecting more bug fixes on thrown exceptions. The threat to external validity includes our limited subjects. This threat can be reduced by selecting more projects. The threat to constructive validity includes the availability of our extracted features.

7 RESEARCH ROADMAP

Our research direction has at least two application scenarios:

1. Recommending thrown exception. When they join a new project, programmers are often unfamiliar with the consensus of throwing correct exceptions. For this application, Finding 1 in our evaluation shows that the f-score values of ThEx are around 0.8. The results are obtained, when programming contexts are complete. Practically, a tool can make predictions after most programming contexts are already written, and the assumption is reasonable, since unknowing the correct exceptions often does not hinder programming tasks. Indeed, other approaches [48, 49] also need complete programming contexts to predict which exceptions shall be caught. As we did, they also assume that programming can write complete programming contexts, even if they fail to identify missing exceptions. Even if a programmer cannot write complete programming contexts, it is feasible to make early predictions. Table 4 shows that losing a feature does not significantly reduce our prediction results. Although losing more features can further reduce our effectiveness, Zhou *et al.* [70] use code synthesis [37, 46] to generate code according to what is already written, and search the clones of generated code. The synthesized code can be used as a seed to find real-code clones. Similar ideas can be used to complement the

features of ThEx. Meanwhile, many exceptions are poorly documented. Buse and Weimer [16] proposed approaches to document exception handling code snippets. Their approach can generate exception documents that are useful to select suitable exceptions. Barbosa *et al.* [12] propose a domain-specific language to enforce handling exceptions. Their language can be extended to enforce writing correct thrown exceptions.

2. Detecting bugs in thrown exceptions. A trained model can make predictions that are different from labels. Although mispredictions are less common, Wasylkowski *et al.* [62] show that such anomalies can indicate bugs. Indeed, after inspecting our wrong predictions, we have detected twenty bugs. As we have identified, such bugs can be introduced, if programmers are unfamiliar with exceptions. Meanwhile, bugs in thrown exceptions can indicate design flaws in exceptions. After the flaws are resolved, such bugs can be avoided at the early stage. After bugs are detected, it can also be interesting to analyze their impacts, and some static approaches [21, 53] can assist the analysis. It is also feasible to generate test cases to highlight the impacts of wrong thrown exceptions, since in another research topic, Wang *et al.* [61] generate stack traces to highlight dependency conflict issues. It is worth exploring the impacts, and the results are useful to understand the importance of wrong thrown exceptions.

For both applications, it is feasible to extend ThEx to more languages. Even if a language has no exceptions, it typically has different categories of errors. ThEx can be extended to predict which type of errors to throw. Other languages can require other features and learning techniques to train their prediction models. In addition, as driven by techniques, researchers can adapt two ways to make further improvements.

1. More advanced techniques. As the first exploration, we build ThEx upon a classical classifier, but deep learning techniques have been used to resolve software engineering problems (*e.g.*, malware detection [41]). Still, a project can throw quite imbalanced exceptions. Gong and Zhong [32] show that even deep learning techniques may not effectively handle such situations. It needs more advanced techniques to handle those challenging situations.

2. Handling exceptions with few instances. Like other mining-based approaches, we can encounter the cold start problem [14], which occurs in all recommendation systems. In the literature, the cold start problem is intensively studied [29, 36, 45]. For these exceptions, heuristic-based approaches and the recent research on small benchmarks [54] can work better, but such approaches can be imprecise due to various analysis challenges. With their support, it can eventually accumulate sufficient data, and thus enable learning-based approaches like ours. Alternative, as project leaders often decide which exceptions shall be thrown, it is feasible to learn from the other projects of the project leaders, although cross-project learning is still an open challenge in mining software repositories.

8 RELATED WORK

The empirical studies on exception handling. Researchers have conducted various empirical studies to understand exception handling. Cabral and Marques [17] analyzes the differences between Java and DotNet. Sena *et al.* [56] analyze the exception handling inside a Java library. Cacho *et al.* [18] analyze the evolution of exception handling code across versions. Coelho *et al.* [24] analyze the exception handling in Android code. Asaduzzaman *et al.* [11] analyze how exceptions are used in Java code. Bruntink *et al.* [15] analyze the exception handling in embedded systems. Koopman and DeVale [42] analyze the exception handling in operating systems. Shah *et al.* [57] analyze different viewpoints on exception handling. Chen *et al.* [22] analyze exception-related bugs in clouds. The above studies show the importance of handling thrown exceptions. Kechagia *et al.* [40] report that most crashes were caused by memory exhaustion, race conditions or deadlocks, and missing or corrupt resources. Our work shows that thrown exceptions can also be faulty, and need to be improved.

Testing with exceptions. Exceptions are useful in testing. Sinha *et al.* [58] construct control flow graphs for exceptions and use such graphs as the criteria for test generation. Cornu *et al.* [27] inject faults to trigger exceptions. Zhang *et al.* [66] enumerate the patterns of external resources to trigger exceptions. Goffi *et al.* [31] infer the conditions of thrown exceptions from API documents, and test whether such exceptions are thrown. Xu and Zhong [65] compare the inconsistencies between exception types and their messages. The above approaches use thrown exceptions to guide test generation or their inconsistencies, but ThEx predicts which exceptions shall be thrown given programming locations.

Analyzing stack traces. Various approaches have been proposed to analyze stack traces. Gu *et al.* [33] identify faults from stack traces. Han *et al.* [35] mine stack traces to locate performance bugs. Wong *et al.* [64] use stack traces to assist fault localization. Chen and Kim [23] reproduce crashes based on stack traces. Our work shows that thrown exceptions can be wrong, and stack traces can contain errors. The above approaches can be improved, if such errors are removed.

9 CONCLUSION

The exception handling mechanism is critical to throw, catch, and handle runtime errors. The prior approaches work on the bugs in catching and handling exceptions, but ignore the bugs in throwing exceptions. Like bugs in other phases of the exception handling mechanism, an incorrectly thrown exception can lead to disastrous consequences. It is often legal to throw multiple types of exceptions, but programmers must follow the local consensuses. Such consensuses are difficult to be obtained and often are unknown to new members. To the best of our knowledge, no prior approach can assist throwing proper exceptions nor detect bugs in thrown exceptions. To improve the state of the art, we propose the first approach that predicts which exceptions shall be thrown. Its basic idea is to learn a classification model from what were already written in source files. We evaluated our approach on nine open source projects, its f-scores and mcc values are both around 0.8. We reported twenty found anomalies as bugs to their developers, and they fixed 13 bug reports.

ACKNOWLEDGMENTS

We appreciate reviewers for their insightful comments. This work is sponsored by the CCF-Huawei Innovation Research Plan No. CCF2021-admin-270-202111.

REFERENCES

[1] 2020. Apache Commons IO. https://commons.apache.org/proper/commons-io/. (2020).
[2] 2020. Eclipse Java development tools. http://www.eclipse.org/jdt/. (2020).
[3] 2020. FileUtils throws inconsistent exceptions. https://issues.apache.org/jira/browse/IO-661. (2020).
[4] 2020. Fixing an exception error in MatrixUtil. https://github.com/itext/itext7/pull/50. (2020).
[5] 2020. HSSFCell.convertCellValueToBoolean shall throw more specific exception. https://bz.apache.org/bugzilla/show_bug.cgi?id=64964. (2020).
[6] 2020. InstructionAdapter shall throw consistent exceptions. https://gitlab.ow2.org/asm/asm/-/issues/317922. (2020).
[7] 2020. IOUtils.toByteArray throws inconsistent exceptions. https://issues.apache.org/jira/browse/IO-696. (2020).
[8] 2020. Running OOS should trigger the disk failure policy. https://issues.apache.org/jira/browse/CASSANDRA-11448. (2020).
[9] 2020. Type1Parser.parseASCII throws inconsistent exceptions. https://issues.apache.org/jira/browse/PDFBOX-5080. (2020).
[10] Mithun Acharya and Tao Xie. 2009. Mining API Error-Handling Specifications from Source Code. In *Proc. FASE*. 370–384.
[11] Muhammad Asaduzzaman, Muhammad Ahasanuzzaman, Chanchal K Roy, and Kevin A Schneider. 2016. How developers use exception handling in Java?. In *Proc. MSR*. 516–519.
[12] Eiji Adachi Barbosa, Alessandro Garcia, Martin P Robillard, and Benjamin Jakobus. 2015. Enforcing exception handling policies with a domain-specific language. *IEEE Transactions on Software Engineering* 42, 6 (2015), 559–584.
[13] Mohamed Bekkar, Hassiba Kheliouane Djemaa, and Taklit Akrouf Alitouche. 2013. Evaluation measures for models assessment over imbalanced data sets. *Evaluation* 3, 10 (2013).
[14] JesúS Bobadilla, Fernando Ortega, Antonio Hernando, and JesúS Bernal. 2012. A collaborative filtering approach to mitigate the new user cold start problem. *Knowledge-based systems* 26 (2012), 225–238.
[15] Magiel Bruntink, Arie Van Deursen, and Tom Tourwé. 2006. Discovering faults in idiom-based exception handling. In *Proc. ICSE*. 242–251.
[16] R.P.L. Buse and W.R. Weimer. 2008. Automatic documentation inference for exceptions. In *Proc. ISSTA*. 273–282.
[17] Bruno Cabral and Paulo Marques. 2007. Exception handling: A field study in Java and. net. In *Proc. ECOOP*. 151–175.
[18] Nélio Cacho, Eiji Adachi Barbosa, Juliana Araujo, Frederico Pranto, Alessandro Garcia, Thiago Cesar, Eliezio Soares, Arthur Cassio, Thomas Filipe, and Israel Garcia. 2014. How does exception handling behavior evolve? an exploratory study in java and c# applications. In *Proc. ICSME*. 31–40.
[19] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM computing surveys* 41, 3 (2009), 1–58.
[20] Byeong-Mo Chang and Kwanghoon Choi. 2016. A review on exception analysis. *Information and Software Technology* 77 (2016), 1–16.
[21] Ramkrishna Chatterjee, Barbara G. Ryder, and William A Landi. 2001. Complexity of points-to analysis of Java in the presence of exceptions. *IEEE Transactions on Software Engineering* 27, 6 (2001), 481–512.
[22] Haicheng Chen, Wensheng Dou, Yanyan Jiang, and Feng Qin. 2019. Understanding Exception-Related Bugs in Large-Scale Cloud Systems. In *Proc. ASE*. 339–351.
[23] Ning Chen and Sunghun Kim. 2014. Star: Stack trace based automatic crash reproduction via symbolic execution. *IEEE transactions on software engineering* 41, 2 (2014), 198–220.
[24] Roberta Coelho, Lucas Almeida, Georgios Gousios, Arie Van Deursen, and Christoph Treude. 2017. Exception handling bug hazards in Android. *Empirical Software Engineering* 22, 3 (2017), 1264–1304.
[25] Roberta Coelho, Awais Rashid, Alessandro Garcia, Fabiano Ferrari, Nélio Cacho, Uirá Kulesza, Arndt von Staa, and Carlos Lucena. 2008. Assessing the impact of aspects on exception flows: An exploratory study. In *Proc. ECOOP*. 207–234.
[26] Ronan Collobert and Jason Weston. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proc. ICML*. 160–167.
[27] Benoit Cornu, Lionel Seinturier, and Martin Monperrus. 2015. Exception handling analysis and transformation using fault injection: Study of resilience against unanticipated exceptions. *Information and Software Technology* 57 (2015), 66–76.
[28] Guilherme B De Padua and Weiyi Shang. 2017. Revisiting exception handling practices with exception flow analysis. In *Proc. SCAM*. 11–20.
[29] Mehdi Elahi, Francesco Ricci, and Neil Rubens. 2016. A survey of active learning in collaborative filtering recommender systems. *Comput. Sci. Rev.* 20 (2016), 29–50. DOI: http://dx.doi.org/10.1016/j.cosrev.2016.05.002
[30] Yoav Freund and Robert E. Schapire. 1996. Experiments with a new boosting algorithm. In *Proc. ICML*. San Francisco, 148–156.
[31] Alberto Goffi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *Proc. ISSTA*. 213–224.
[32] Siyi Gong and Hao Zhong. 2022. A study on identifying code author from real development. In *Proc. ESEC/FSE*. to appear.
[33] Yongfeng Gu, Jifeng Xuan, Hongyu Zhang, Lanxin Zhang, Qingna Fan, Xiaoyuan Xie, and Tieyun Qian. 2019. Does the fault reside in a stack trace? Assisting crash localization by predicting crashing fault residence. *Journal of Systems and Software* 148 (2019), 88–104.
[34] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* 11, 1 (2009), 10–18.
[35] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. 2012. Performance debugging in the large via mining millions of stack traces. In *Proc. ICSE*. 145–155.
[36] Chen He, Denis Parra, and Katrien Verbert. 2016. Interactive recommender systems: A survey of the state of the art and future research challenges and opportunities. *Expert Systems with Applications* 56 (2016), 9–27.
[37] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proc. 34th ICSE*. 837–847.
[38] Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray. 2016. Automatically detecting error handling bugs using error specifications. In *Proc. USENIX Security*. 345–362.
[39] 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 *Proc. ICPC*. 369–380.
[40] Maria Kechagia, Dimitris Mitropoulos, and Diomidis Spinellis. 2015. Charting the API minefield using software telemetry data. *Empirical Software Engineering* 20, 6 (2015), 1785–1830.
[41] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. 2016. Deep learning for classification of malware system call sequences. In *Proc. AusAI*. 137–149.
[42] Phil Koopman and John DeVale. 2000. The exception handling effectiveness of POSIX operating systems. *IEEE Transactions on Software Engineering* 26, 9 (2000), 837–848.
[43] Josh Lerner, Parag A Pathak, and Jean Tirole. 2006. The dynamics of open-source contributors. *American Economic Review* 96, 2 (2006), 114–118.
[44] Zexuan Li and Hao Zhong. 2021. Revisiting textual feature of bug-triage approach. In *Proc. ASE*. 1183–1185.
[45] Blerina Lika, Kostas Kolomvatsos, and Stathes Hadjiefthymiades. 2014. Facing the cold start problem in recommender systems. *Expert Systems with Applications* 41, 4 (2014), 2065–2073.
[46] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. 2020. A Self-Attentional Neural Architecture for Code Completion with Multi-Task Learning. In *Proc. ICPC*. 37–47.
[47] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Proc. NIPS*. 3111–3119.
[48] Tam Nguyen, Phong Vu, and Tung Nguyen. 2019. Recommending exception handling code. In *Proc. ICSME*. 390–393.
[49] Tam Nguyen, Phong Vu, and Tung Nguyen. 2020. Code recommendation for exception handling. In *Proc. ESEC/FSE*. 1027–1038.
[50] Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani. 2001. A study of exception handling and its dynamic optimization in Java. In *Proc. OOPSLA*. 83–95.
[51] J Ross Quinlan. 2014. *C4. 5: programs for machine learning*. Elsevier.
[52] Mohammad Masudur Rahman and Chanchal K Roy. 2014. On the use of context in recommending exception handling code examples. In *Proc. SCAM*. 285–294.
[53] Martin P Robillard and Gail C Murphy. 2003. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology* 12, 2 (2003), 191–221.
[54] Abhishek Samanta, Aheli Saha, Suresh Chandra Satapathy, Steven Lawrence Fernandes, and Yo-Dong Zhang. 2020. Automated detection of diabetic retinopathy using convolutional neural networks on a small dataset. *Pattern Recognition Letters* 135 (2020), 293–298.
[55] Ramasubramanian Sekar, Ajay Gupta, James Frullo, Tushar Shanbhag, Abhishek Tiwari, Henglin Yang, and Sheng Zhou. 2002. Specification-based anomaly detection: a new approach for detecting network intrusions. In *Proc. CCS*. 265–274.
[56] Demóstenes Sena, Roberta Coelho, Uirá Kulesza, and Rodrigo Bonifácio. 2016. Understanding the exception handling strategies of Java libraries: An empirical study. In *Proc. MSR*. 212–222.
[57] Hina Shah, Carsten Gorg, and Mary Jean Harrold. 2010. Understanding exception handling: Viewpoints of novices and experts. *IEEE Transactions on Software Engineering* 36, 2 (2010), 150–161.
[58] Saurabh Sinha and Mary Jean Harrold. 2000. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering* 26, 9 (2000), 849–871.
[59] Daohan Song, Hao Zhong, and Li Jia. 2020. The Symptom, Cause and Repair of Workaround. In *Proc. ASE*. 1264–1266.
[60] Pascal Soucy and Guy W Mineau. 2005. Beyond TFIDF weighting for text categorization in the vector space model. In *IJCAI*, Vol. 5. 1130–1135.
[61] Ying Wang, Ming Wen, Rongxin Wu, Zhenwei Liu, Shin Hwei Tan, Zhiliang Zhu, Hai Yu, and Shing-Chi Cheung. 2019. Could I have a stack trace to examine the

dependency conflict issue?. In *Proc. ICSE*. 572–583.
[62] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting Object Usage Anomalies. In *Proc. ESEC/FSE*. 35–44.
[63] Westley Weimer and George C Necula. 2008. Exceptional situations and program reliability. *ACM Transactions on Programming Languages and Systems* 30, 2 (2008), 1–51.
[64] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *Proc. ICSME*. 181–190.
[65] Lin Xu and Hao Zhong. 2021. Detecting inconsistent thrown exceptions. In *Proc. ICPC*. 391–395.
[66] Pingyu Zhang and Sebastian Elbaum. 2012. Amplifying tests to validate exception handling code. In *Proc. ICSE*. 595–605.
[67] Yizhou Zhang, Guido Salvaneschi, Quinn Beightol, Barbara Liskov, and Andrew C Myers. 2016. Accepting blame for safe tunneled exceptions. In *Proc. PLDI*. 281–295.
[68] Hao Zhong and Hong Mei. 2019. An empirical study on API usages. *IEEE Transactions on Software Engineering* 45 (2019), 319–334. Issue 4.
[69] Hao Zhong and Hong Mei. 2020. Learning a graph-based classifier for fault localization. *Science China Information Sciences* 63 (2020), 1–22.
[70] Shufan Zhou, Beijun Shen, and Hao Zhong. 2019. Lancer: Your code tell me what you need. In *Proc. ASE*. 1202–1205.