Code Recommendation for Exception Handling

Anonymous Author(s)

ABSTRACT

Exception handling is an effective mechanism to avoid unexpected runtime errors. However, novice programmers might fail to handle exceptions properly, causing serious errors like system crashing or resource leaking. In this paper, we introduce FuzzyCatch, a code recommendation tool for handling exceptions. Based on fuzzy logic, FuzzyCatch can predict if a runtime exception would occur in a given code snippet and recommend code to handle that exception. FuzzyCatch is implemented as a plugin for Android Studio. The empirical evaluation suggests that FuzzyCatch is highly effective. For example, it has top-1 accuracy of 77% on recommending what exception to catch in a try catch block and of 70% on recommending what method should be called when such an exception occurs. FuzzyCatch also achieves a high level of accuracy and outperforms baselines significantly on detecting and fixing real exception bugs.

1 INTRODUCTION

Exceptions are unexpected errors occurring while a program is running. If not handled properly, exceptions could lead to serious problems such as system crashes or resource leaks. For example, when a program tries to parse an integer from a string but the string does not represent a valid number, a NumberFormatException exception occurs. If the program ignores that exception and continues to read, it will crash. Thus, effective exception handling is important in software development. A prior study reports that correctly handling exceptions could improve 17% in the performance of software systems [42].

Most modern programming languages provide built-in support for exception handling. For example, in Java, we can wrap a try block around a code fragment where an exception might occur and add a catch block to handle that exception. However, programmers have to determine themselves what code fragment to protect with the try block, what exception type to handle in the catch block, and what to react when such an exception occurs.

Learning to handle exceptions properly is often challenging due to several reasons. First, modern software development relies strongly on API libraries. An API library often defines many specific exception types and exception handling rules. For example, in Java SDK, class FileNotFoundException is defined for the *'file not found'* exception. When such an exception happens, the software system could react by notifying users about the error and writing the relevant information (timestamp, filename, caller, etc.) to a system log for future debugging.

However, a major API library often consists of a large number of components. For example, the Android application framework contains over 3,400 classes, 35,000 methods, and more than 260 exception types [24]. Thus, it is very difficult to learn and memorize what method could cause what exception and what to react when a particular exception occurs. Secondly, popular API libraries are often upgraded quickly. For example, in 10 years from 2008 to 2018, 28 major versions of Android SDK have been released. The latest version of Java SDK is JDK 12 is released in 3/2019. Each of those major versions often includes very large changes, which could introduce new types of exceptions and new rules for exception handling.

Most importantly, the documentation for exception handling is generally insufficient. Kechagia *et al.* [21] found that 69% of methods in Android SDK have undocumented exceptions and 19% of the crashes could have been caused by insufficient documentation. Coelho *et al.* [11] found that documentation for many explicitly thrown runtime exceptions (or often called unchecked exceptions) is rarely provided.

Modern code editors can enforce catching of checked exceptions. For example, in Java, FileNotFoundException is a checked exception. Thus, when a new FileInputStream object is created to read from a file, Eclipse will ask the programmer to catch the FileNotFoundException (or throw it). However, such support is not available for runtime (unchecked) exceptions like IllegalArgumentException. In addition, popular code editors cannot suggest the reactions when a checked or unchecked exception happens.

In this paper, we introduce FuzzyCatch, a code recommendation plugin for Android Studio with advanced support for exception handling. Based on fuzzy logic rules learned from thousands of high-quality programs publicly available in app stores, FuzzyCatch can predict if a runtime exception potentially occurs in a code snippet. Then, as the programmer requests, it can generate the try-catch statement with catch block containing code to catch that exception and recover from it.

The key idea of FuzzyCatch is based on the observation that exception handling can be expressed by two general programming rules: i) *if call method m then catch an exception e*; and ii) *if an exception E occurs then perform action r to react*. For example, if you call the method Activity.unregisterReceiver then you have to catch an IllegalArgumentException. Or, if you call the Integer.parseInt then you have to catch an NumberFormatException. And, when an SQLiteDatabase occurs, you call method Cursor.close as a simple reaction to close the resource.

FuzzyCatch learns such rules from a large collection of high-quality code. For example, assume that no available documentation specifies that if we call method *m*, we need to catch an exception of type *E*. However, we observe from thousands of high-rated apps in Google Android Appstore that whenever *m* are called, 90% of the times an exception of type *E* is also caught. Such a high level of co-occurrence suggests a rule that if we call *m*, we should catch *E*. However, because those rules are likely *imperfect*, fuzzy logic is used to represent and combine them. The details of our approach are presented in Section 3.

We have conducted several experiments to evaluate the usefulness and effectiveness of our tool. We collected a dataset containing over 21 million methods from four thousand high rated mobile apps in Google Android Appstore. The 10-fold cross-validation of FuzzyCatch on this dataset are promising. FuzzyCatch has the top-1 accuracy of 77% for recommending exception types. That means,

in 77% of the cases, the first exception type that FuzzyCatch recommends catching is the one used by app developers. The top-1 accuracy of recommending reaction code is of 70%.

We collected a second dataset of 1,000 real exception related bugs. In its strictest mode, FuzzyCatch flags 734 cases (73%), i.e., suggesting they need exception handling. In its loosest mode, FuzzyCatch flags 821 cases (82%). However, the loosest mode would make around 35% unwanted warnings on code that programmers have never added exception handling code. The strictest mode would make only 7% of unwanted warnings. On the task of recommending handling actions, FuzzyCatch provides meaningful recommendations in roughly 65% of the bug fixes and outperforms the baselines significantly. The details of our evaluation process are presented in Sections 5.

The key contributions of our paper include:

- A novel approach based on fuzzy logic for exception handling code recommendation
- A plugin for Android Studio implemented that approach for Java and Android mobile app development
- An extensive evaluation of our approach and our tool.

2 MOTIVATION EXAMPLES

Although, most modern programming languages provide built-in support for exception handling. Programmers still have to determine themselves what code fragment to protect with the try block, what exception type to handle in the catch block, and what to react when such an exception occurs. The previous studies [15, 33] shows that programmers might easily forget to handle an exception that potentially occurs in a code snippet. Such mistakes could cause serious issues to the developing app, i.e. crashing. For example, Figure 9 (Section 5) shows an serious bug caused by not catching a RuntimeException.

Furthermore, the studies also suggest that in practice, novice and even experienced programmers might not always handle exceptions properly. For example, Figure 2 shows an exception related bug fix in the WordPress mobile app. The programmer calls method Activity.unregisterReceiver() with variable mediaUpdate passed as its argument. At runtime, this call causes an IllegalArgumentException and crashes the mobile app. This error is fixed by a try catch block that handles that exception. However, the catch block has no code. Although this bug fix makes the app not crash again, it is a bad programming practice to catch an exception and do nothing.

```
- unregisterReceiver(mediaUpdate);
+ try {
+     unregisterReceiver(mediaUpdate);
+ } catch (IllegalArgumentException e) { }
```

Figure 1: An Example of Swallowing an Exception

Figure 2 shows another exception related bug fix of Cursor object. In this example, an exception occurs when using Cursor object. To handle this bug, the programmer added a try-catch block to cover the code portion that uses the Cursor object, and returned null in the catch block. This action of handling may lead to memory leak bugs. After exception occurs and the function returns, the Cursor object still lives and holds system resources until it is collected by the garbage collector. This may prevent other parts of the program to access the resources. The proper way to handle this exception is calling close method on the cursor object before the return statement. The close method will release resources that hold by the cursor object before the function lost the reference to it.

COMMIT MESSAGE: "catch exception when reading message id from database."

```
. . .
+ try {
      if (cursor.getCount() == 0) {
          return null;
      } else {
          cursor.moveToFirst();
          return new Pair<>(cursor.getLong(), cursor.getString(1));
      }
+ } catch (Exception e) {
+     return null;
+ }
```

Figure 2: An Exception Related Bug Fix

The examples indicate the need exception handling recommendation tools for programmers. Such programming tools could warn programmers about potential exceptions that might occur in the code and assist programmers to handle the exceptions correctly.

3 APPROACH

In this section, we discuss the key techniques of FuzzyCatch. We will start by introducing XRank, a fuzzy logic system to rank and suggest exception types to catch given a piece of code. Then we describe XHand, a fuzzy logic system for recommending repairing actions when a particular exception occurs in that piece of code.

3.1 Recommending Exception Types

We formulate the problem of recommending exception types as the following: Given a code snippet C, determine of the runtime exception(s) e with a high possibility to occur when C is executed.

Our key idea to solve this problem is to learn from the exception handling code already written in high-quality software systems. We first collect a large collection of high-quality code D. For each method m called in C, we look at all every call of m in D. If those calls often associate with catching an exception e then it is likely that calling m could cause e. Since there might be several methods called in C and several exceptions associating with them in D, we need to combine and rank those exceptions.

This is an example from our experiment data. As discussed in detail in Section 5, we collected roughly 13,000 highly-rated Android mobile apps. In those apps, we found 38,218 calls to method Integer.parseInt. Among them, 12,469 calls (32.6%) appear in try-catch blocks catching NumberFormatException. And this is the exception most co-occurred with Integer.parseInt. Thus, even without documentation, we can infer that calling Integer.parseInt could cause NumberFormatException. Of course, Java API documentation comfirms that.

Similarly, we found that, among 23,406 method calls to Cursor.moveToFirst, 8,159 calls (34.8%) co-occur with catching SQLiteException. That means, we could infer that the method call Cursor.moveToFirst might cause SQLiteException. This is also confirmed by Android API documentation. Of course, calls to Integer.parseInt and Cursor.moveToFirst often also co-occur with catching Exception and Throwable, the most general exception types in Java.

Now, assume that a programmer is writing a code snippet containing a call to Cursor.moveToFirst to read data from a database query and a call to Integer.parseInt to parse such data. The above exception handling rules learned from high quality code suggest him/her to catch NumberFormatException and SQLiteException (or the more general ones Exception and Throwable). However, those rules are *imperfect* (e.g., recommending to catch NumberFormatException is applicable for only 32.6% of calls to Integer.parseInt).

In this paper, we address that issue via fuzzy logic. We developed XRank, a fuzzy logic system in FuzzyCatch to rank and recommend exception types. Basically, XRank contains a collection of rules *"if call m then catch e"*, in which m is an method (e.g., Integer.parseInt) and e is an exception type (e.g., NumberFormatException). Because those rules are imperfect, each rule has a confidence level $\mu_m(e)$.

XRank uses fuzzy sets to represent those rules. For each method m, X_m is defined as *"the set of exceptions to catch when calling m"*. As a fuzzy set, X_m has a membership function $\mu_m()$ which calculates membership score $\mu_m(e)$ for every available exception type e.

In traditional fuzzy logic systems, membership functions are defined by domain experts. However, in FuzzyCatch, we define membership functions empirically, i.e., the membership scores $\mu_m(e)$ is estimated from the training data (a repository of high-quality code). The key assumption is that the more m and e co-occur in high-quality code written by professional programmers, the more confidence that if we call m, we should catch e (because professional programmers often do that). Therefore, $\mu_m(e)$ is calculated based on the co-occurrence of m and e in the training data. Assume that n_m is the number of calls to m and $n_{m,e}$ is is the number of calls to m in all try-catch blocks catching e, then:

$$\mu_m(e) = \frac{n_{m,e}}{n_m} \quad (1)$$

Formula 1 implies that the value of $\mu_m(e)$ is between $[0, 1]$. Table 1 lists the top five exception types to catch for several methods and their membership scores estimated from our experiment data. We could see that for Integer.parseInt, NumberFormatException is the top-1 exception type, co-occurring with 32.6% of its method calls. The next one is the general exception type Exception, co-occurring with 14.4%. The remaining exception types have much lower scores.

XRank utilizes the fuzzy sets X_m for two tasks: assessing if a given code snippet needs a try-catch block (i.e.,) and ranking exception types to catch in that block. To do that, for each method m, it first calculates ρ_m, the *exception risk* of m, which is the total possibility that calling m would cause an exception. Because X_m is the fuzzy set representing the possibility that calling m would cause each exception e, the exception risk ρ_m is calculated as the total weight of X_m:

$$\rho_m = |X_m| = \sum_e \mu_m(e) \quad (2)$$

Table 2 lists the exception risk of three methods in Table 1. We can see that among the three of them, Cursor.moveToFirst is the riskiest, e.g., nearly 60% of its calls associated with catching an exception. In contrast, just 14% of calls to MediaPlayer.getDuration associate with catching an exception, thus, its exception risk is much lower.

Fuzzy logic systems often use linguistic concepts and variables to make them easier to use and understand for the domain experts and end-users of those systems. To make FuzzyCatch easier to use for programmers, we classify the methods into different risk classes based on their exception risk scores. Assume that α_1, α_2, and α_3 are respectively the 20%, 30%, and 50% percentile of all exception risk scores. Then, if method m has $\rho_m > \alpha_1$ (i.e., m is the top-20% among all methods), it is classified as "VERY RISKY". If $\alpha_1 \geq \rho_m > \alpha_2$ (m is not in top-20% but in top-30%), it is classified as "RISKY". And if If $\alpha_2 \geq \rho_m > \alpha_3$ (m is not in top-30% but in top-50%) it is classified as "LESS RISKY". The remaining methods are considered as not risky and ignored.

In the IDE (Android Studio), we color-code method calls with their risk levels. For example, calls of "VERY RISKY" methods like Cursor.moveToFirst are colored RED. Calls of "RISKY" methods (like Integer.parseInt) are colored ORANGE. Calls of "LESS RISKY" methods (like IntegerMediaPlayer.getDuration) are colored YELLOW. Calls of other methods are unchanged.

We could see that for a single method call to m, the exception risk ρ_m (which is correspondingly color-coded as visual guide for programmers) and the membership scores in the fuzzy set X_m are

Table 1: Membership Scores

Integer.parseInt (n_m = 38,218)	$n_{m,e}$	$\mu_m(e)$
NumberFormatException	12,469	0.326
Exception	5,496	0.144
Throwable	687	0.018
IllegalArgumentException	470	0.012
JSONException	427	0.011
Cursor.moveToFirst (n_m = 23,406)	$n_{m,e}$	$\mu_m(e)$
SQLiteException	8,159	0.349
Exception	4,288	0.183
SQLException	460	0.020
Throwable	410	0.018
SQLiteFullException	216	0.009
MediaPlayer.getDuration (n_m = 1,152)	$n_{m,e}$	$\mu_m(e)$
IllegalStateException	105	0.091
Exception	44	0.038
Throwable	7	0.006
IllegalArgumentException	5	0.004
SecurityException	2	0.002
Byte.parseByte (n_m = 145)	$n_{m,e}$	$\mu_m(e)$
NumberFormatException	17	0.117
Exception	8	0.055
JSONException	3	0.020
Throwable	3	0.020
NoSuchElementException	2	0.013

Table 2: Exception Risk Scores

Method	n_m	ρ_m
Cursor.moveToFirst	23,406	0.578
Integer.parseInt	38,218	0.512
MediaPlayer.getDuration	1,152	0.141

sufficient to suggest if a try-catch block is needed and to recommend exceptions to catch in such a block. For example, a call to Cursor.moveToFirst should be recommended with a try-catch block (because it is classified as "VERY RISKY" to cause an exception). Then, the top exception to catch when calling Cursor.moveToFirst should be SQLiteException.

However, programmers often do not write try-catch block for each single call. We are more likely to write a try-catch block for a code snippet which might contain several method calls. Thus, to combine exception handling rules of all those method calls, XRank also uses a fuzzy logic approach.

Let us discuss an example first. Assume that a code snippet C contains a call to Cursor.moveToFirst to read data from a database query and a call to Integer.parseInt to parse such data. Then, because both of them have high exception risk, the code snippet also has high risk of causing exceptions. Two most potential exceptions are SQLiteException (from calling Cursor.moveToFirst and NumberFormatException (from calling Integer.parseInt). Thus, a programmer might write a try block around the code snippet and two catch blocks for those two exceptions. However, he/she might also write a simpler solution by using only a catch block catching Exception, which generalizes both NumberFormatException and SQLiteException and also has high membership scores in the fuzzy sets of exceptions of both Cursor.moveToFirst Integer.parseInt. In other words, when two methods Cursor.moveToFirst and Integer.parseInt are called together, the potential exceptions are combined from those for each of that method. In fuzzy logic, this combination is a fuzzy *union*.

In general, assume that C contains k method calls $m_1, m_2..., m_k$. Let X_C be the fuzzy set of exceptions should be caught for C, then X_C is the union of all fuzzy sets of exceptions for $m_1, m_2..., m_k$:

$$X_C = X_{m_1} \cup X_{m_1} \cup ... \cup X_{m_k} \tag{3}$$

Because X_C is a fuzzy set, it has a membership function μ_C. The union operation in fuzzy logic is defined via calculating μ_C from $\mu_{m_1}...\mu_{m_k}$. There are several formula for fuzzy union, we use the following one:

$$\mu_C(e) = 1 - (1 - \mu_{m_1}(e)) \times (1 - \mu_{m_2}(e)) \times ... \times (1 - \mu_{m_k}(e)) \tag{4}$$

Similarly, we also calculate the exception risk of the whole code snippet C as the following:

$$\rho_C = 1 - (1 - \rho_{m_1}) \times (1 - \rho_{m_2}) \times ... \times (1 - \rho_{m_k}) \tag{5}$$

For example, assume that e is Exception, m_1 is Cursor.moveToFirst, and m_2 is Integer.parseInt. From Table 1, $\mu_{m_1}(e) = 0.183$, $\mu_{m_2}(e) = 0.144$. Thus, $\mu_C(e) = 1 - (1 - 0.183)(1 - 0.144) = 0.301$. This membership score is higher suggesting that Exception is more likely to be used in the catch block when both methods are called. Similarly, from Table 1, $\rho_{m_1} = 0.578, \rho_{m_2} = 0.512$. Thus, $\rho_C = 1 - (1 - 0.578)(1 - 0.512) = 0.794$. This exception risk is higher suggesting that when both methods are called, it is higher possible that an exception will occur, which is intuitively plausible.

In summary, XRank recommends for a given code snippet C as the following. First, it calculates the exception risk ρ_C. If ρ_C exceeds the user preset risk level (e.g., VERY RISKY with $\rho_C > \alpha_1$), then it calculates $\mu_C(e)$ for every available exception type e and ranks them descendingly. FuzzyCatch presents the top exceptions in the ranked list to the programmer. When he/she selects one exception, FuzzyCatch will generate a try-catch block around code snippet C.

3.1.1 Improvements. To improve the prediction accuracy, we incorporate several programming language features to the XRank model. First, we observe that exceptions have hierarchical structure. For instance, in Java, all Exceptions and Errors are sub-classes of Throwable, all runtime exceptions are sub-classes of RuntimeException, and RuntimeException is a sub-class of Exception. With the hierarchical structure, programmers can use a super-class exception in any catch block of its sub-class exception. For example, in Figure 9, the programmer could use Exception in the catch block instead of its subclass RuntimeException. In XRank, we incorporate this hierarchical property by modifying the value of $n_{m,e}$ in the Equation 1. Anytime m appears in a try block while e appears in the corresponding catch block, in addition to increase the value of $n_{m,e}$ by 1, we also increase the value of $n_{m,e'}$ by a weighted value $w_{e'}$ if e' is a super-class of e. $w_{e'}$ could be calculated as the frequency of e' in all catch blocks:

$$w_{e'} = \frac{n_{e'}}{\sum n_e} \tag{6}$$

Most modern object-oriented programming languages support polymorphism. Overloading methods is one form of polymorphism in which methods within a class can have the same name if they have different parameter lists. For example, the Integer class has two methods for parsing numbers that have the same name parseInt(String s) and parseInt(String s, int radix). The later method has an additional parameter for radix. These overloading methods often perform similar tasks, thus, they are likely to throw similar exceptions. Furthermore, the later method in the example appears much less than the method parseInt(String s). Thus, learning rules for parseInt(String s, int radix) could be challenging due to lack of data. We improve the prediction accuracy of XRank by grouping overloading methods together and consider each group as one method when learning the model.

Finally, Table 1 also shows the fuzzy set of the API method Byte.parseByte. We can see that although parseByte is similar to parseInt and parseLong, the statistics of parseByte are small because it appears much less than the other methods. Thus, learning rules for parseByte could be challenging due to lack of data. To solve this problem, we group methods with the same aspect and learn the usage rules for the group. For example, we group the three methods as *"number parsing"* methods. The exception handling rule for the group are described as catching NumberFormatException when parsing number.

3.2 Recommending Exception Reaction

In addition to recommending what exception to catch, FuzzyCatch also recommends code to react when such an exception occurs by written in the catch block. Figures 1 to 3 illustrate code examples with different ways to write the code to react when an exception occurs. We consider the non-reaction in Figure 2 as a bad practice. The other code examples suggest that the reaction code often i) try to recover after the error (e.g., by making a workable object as in Figure 2) and ii) log the error for future investigations (e.g., by calling Log.i("WordPress", e.getMessage()) as in Figure 3), and iii) clean up the system (e.g., by releasing resources as in Figure 3).

```
//Usage 1
try {
  httpConnection.openConnection();
  httpConnection.setRequestProperty();
  ...
  httpConnection.connect();
} catch (IOException e){
  in = httpConnection.getErrorStream();
  ...
}
```

```
//Usage 2
try {
  httpConnection.getInputStream();
  httpConnection.getHeaderField();
  ...
} catch (IOException e){
  int responseCode =  httpConnection.getResponseCode();
  httpConnection.disconnect();
  ...
}
```

Figure 3: Handling Exceptions for HttpURLConnection

The examples suggest that reaction code written in the catch block can be large and complicated. However, our study (Section 2) suggests that in most cases, programmers often call only one method for each object such as printStackTrace for an Exception object, close for an SQLiteDatabase object, or disconnect for an HttpURLConnection object. We consider creating a new object (such as Date now = new Date()) as calling a special method new. Similarly, assigning a value to an object (such as in = httpConnection.getErrorStream()) is like calling a special method assign.

Therefore, FuzzyCatch recommends code in the catch block by considering all objects appearing in the try block and providing a ranked list of methods to call for that object. XHand is the fuzzy logic system responsible to generate that ranked list.

One observation when developing XHand is that the method r to call for reaction depends on the method called and potentially cause the exception in the try block. For example, consider two usages of a HttpURLConnection in Figure 3. In the first usage, the exception may occur while using the HttpURLConnection for connecting to a server via connect method. In this situation, we would want to get the error stream by calling getErrorStream to get information about the error for further processing. In the second usage, the object is already connected to the server but the read timeout expires causing an exception. In this situation, the exception handler, we would want to get the response code and disconnect from the server.

Thus, the key idea of XHand is to learn the fuzzy rule *"if call m in the try block then call r in the catch block to react or recover"*. For example, as in Figure 3, SQLiteDatabase.delete is called in the try block, thus, SQLiteDatabase.close is called in the catch block to release resources.

Thus, similar to XRank, we also define in XHand for each method m a fuzzy set R_m of methods to call for reaction to exception possibly caused by calling m. Its membership function ν_m is defined as:

$$\nu_m(r) = \frac{n_m(r)}{n_m} \tag{7}$$

In this formula, n_m is again the number of calls of m and $n_m(r)$ is the number of times m is called in a try block and r is called in the corresponding catch block.

Sometimes the method called for reaction depends on the occurring exception. For example, if the exception is Exception, then programmers usually call its method printStackTrace or getMessage. In other words, XHand also needs to learn the fuzzy rule *"if exception e occurs then call r to react or recover"*. Thus, for each exception e, we also define a fuzzy set R_e of methods to call for reaction if e occurs. Its membership function ν_e is defined as:

$$\nu_e(r) = \frac{n_e(r)}{n_e} \tag{8}$$

In this formula, n_e is the number of times e appears in a catch block m and $n_e(r)$ is the number of times r is called in that catch block.

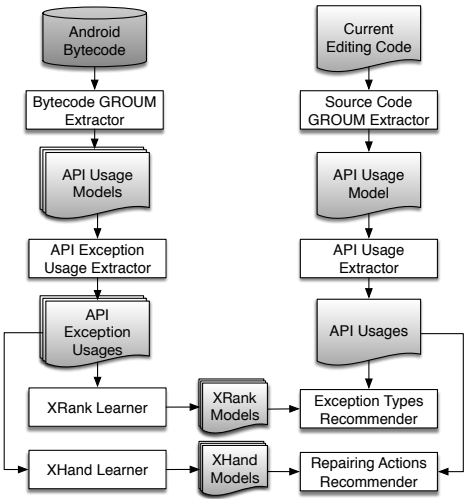

Figure 4: Design Overview of FuzzyCatch

Then, similar to XRank when a code snippet C calls k methods $m_1, m_2..., m_k$ and catches exception e. Let R_C be the fuzzy set of methods should be called for reaction, then R_C is the union of all fuzzy sets $R_{m_1}, R_{m_2}..., R_{m_k}$ and R_e. Its membership function is defined as:

$$\nu_C(r) = 1 - (1 - \nu_{m_1}(r)) \times (1 - \mu_{m_2}(r)) \times ... \times (1 - \nu_e(r)) \tag{9}$$

Finally, for each object o in C, if $r \in R_C$ is a method callable for o (e.g., r is a method of o, or r can accept o as an argument), it will be added to a list. The list is ranked descendingly by the membership scores and presented to the users of FuzzyCatch.

4 SYSTEM IMPLEMENTATION

In this section, we discuss the implementation and usage of FuzzyCatch. Currently available at rebrand.ly/exassist, it is released as a plugin of IntelliJ IDEA and Android Studio, two popular IDEs for Java programs and Android mobile apps.

4.1 Design Overview

Figure 4 shows the design overview of FuzzyCatch. It has a) two modules to extract API usage models from source code and bytecode, b) one module to extract exception handling information from those usage models, c) two modules to train XRank and XHand from those extracted data, and d) two modules using the trained fuzzy logic systems to recommend code for catching and reacting to exceptions. Let us describe those modules in more detail.

4.1.1 GROUM Extractor. FuzzyCatch employs GROUM (Graph-based Object Usage Model) to represent the raw API usages. It has a module to extract GROUMs from the bytecode of Android apps in the training dataset. It also has a similar module to extract GROUMs from the code being written, which are used for its two tasks of recommending exception types and repairing actions. The extracting algorithms could be found at [30–32].

4.1.2 API Exception Usage Extractor. Because XRank and XHand are trained from usages of API objects and method calls in exception handling code, FuzzyCatch has a module named API Exception

Usage Extractor to extract from those usages from GROUMs. This module traverses each sub-graph of a GROUM representing a try-catch block and extracts all API method calls and the catching exception. The temporal order and data dependency between those API method calls are also extracted. The module then stores that information as an API exception usage.

4.1.3 XRank and XHand Learners. These modules are responsible for training XRank and XHand systems from the extracted API exception usages. They count the raw occurrences n_m, $n_m(e)$, $n_m(r)$..., compute membership scores $\mu_m(e)$, $\nu_m(r)$... and exception risk scores ρ_m. (See Section 3 for details).

4.1.4 Exception Types Recommender. This module provides the recommendations on unchecked exception types for a selected code snippet C. It first extracts the set of API method calls that appeared in the under-editing code. Then, it utilizes XRank to compute exception risk scores for each call and make them color-coded in the IDE (e.g., red for VERY RISKY method calls). The programmer can use the color-code to select the code snippet of his/her interest for exception handling. Once the code is selected, XRank is used to calculate the membership scores of exceptions towards method calls appearing in the selected code. Top-ranked exceptions are added to the recommendation list.

4.1.5 Repairing Actions Recommender. This module provides the recommendations for repairing actions in exception handling code. It extracts the set of API method calls in the try block the catching exception e. It then groups API method calls by objects. All API method calls in a group of an object have data dependency with that object. It uses the XHand system to recommend repairing actions for each object. Finally, it combines all the predict repairing actions to produce the final recommendation.

4.2 Usage

In this section, we present how to use FuzzyCatch in practice. Figure 5 shows a code example. Assume a programmer is writing code to open and get data from a database. She first opens a SQLiteDatabase and assigns it to variable sqliteDB. She then runs some SQL commands to update the database. Next, she creates a query that returns a Cursor object and uses that Cursor object to retrieve data to a list named bookTitles.

While the programmer is writing the code, FuzzyCatch utilizes XRank to access the exception risks and color-code its method calls. For example, the calls cursor.moveToFirst and cursor.moveToNext are red-flagged, bookTitles.add is orange-flagged. That makes the programmer aware that the code is dealing with database and calling Cursor's methods might cause *unchecked exceptions* (runtime exceptions) at runtime. (Without FuzzyCatch, the built-in exception checker in Android Studio only supports adding *checked exceptions*, thus, does not help her to make appropriate action in this case).

Now, the programmer invokes FuzzyCatch by selecting the code snippet that she wants to check for exception then pressing Ctrl + Alt + R. In Figure 6, FuzzyCatch is invoked for a code snippet reading data from database with the Cursor object. It suggests a ranked list of unchecked exceptions with SQLiteException at the top. If the programmer chooses this exception, FuzzyCatch will wrap the currently selected code with a try-catch block with SQLiteException in the catch expression.

```
// create Cursor in order to parse our sqlite results
Cursor cursor = sqliteDB.rawQuery( sql: "SELECT bookTitle FROM " + tableName,  selectionArgs: null);
// if Cursor is contains results
if (cursor != null) {
        // move cursor to first row
    if (cursor.moveToFirst()) {
        do {
            // Get version from Cursor
            String bookName = cursor.getString(cursor.getColumnIndex( columnName: "bookTitle"));
            // add the bookName into the bookTitles ArrayList
            bookTitles.add(bookName);
            // move to next row
        } while (cursor.moveToNext());
    }
}
Collections.sort(bookTitles);
```

Figure 5: Exception Warnings by FuzzyCatch

```
if (cursor != null) {
    if (cursor.moveToFirst()) {
        do {
            // Get version from Cursor
            String bookName = cursor.getString(cursor.getColumnIndex( columnName: "bookTitle"));
            // add the bookName into the bookTitles ArrayList
            bookTitles.add(bookName);
            // move to next row
        } while (cursor.moveToNext());
    }
}
Collections.sort(bookTitles);
```

Exception Types
1. android.database.sqlite.SQLiteException (0.80)
2. android.database.sqlite.SQLiteFullException (0.14)
3. android.database.sqlite.SQLiteFullException (0.06)

Figure 6: Recommending Exception Types

```
                } while (cursor.moveToNext());
            }
        }
        Collections.sort(bookTitles);
    } catch (SQLiteException e) {
        if (cursor != null) {
            cursor.close();
        }
        bookTitles = null;
    }
```

Handling Actions
1. cursor.close(); bookTitles = null;

Figure 7: Recommending Repairing Actions

Figure 7 demonstrates the usage of FuzzyCatch in recommending exception repairing actions. After adding a try-catch block with SQLiteException for the code in the previous scenario, the programmer wants to perform recovery actions. To invoke FuzzyCatch for this task, she moves the cursor to the first line of the catch and presses Ctrl + Alt + H. In the example, FuzzyCatch detects that the Cursor object should be closed to release all of its resources and making it invalid for further usages. It also suggests to set bookTitles equals null to indicate the error while collecting data from cursor. If the programmer chooses the recommended actions, FuzzyCatch will generate the code in the catch block as in the figure. To save space, we show both the recommendation and generated code in a same window.

5 EMPIRICAL EVALUATION

We conducted several experiments to evaluate the effectiveness of FuzzyCatch on recommending exception handling code for Android and Java APIs. All experiments are executed on a computer running Windows 10 with Intel Core i7 3.6Ghz CPU, 8GB RAM, and 1TB HDD storage.

Table 3: Data Collection

Number of apps	13,463
Number of classes	21,527,731
Number of methods	26,235,142
Number of bytecode instructions	741,912,624
Space for storing .dex files	165.0 GB
Number of exception types	261
Number of API methods	64,685

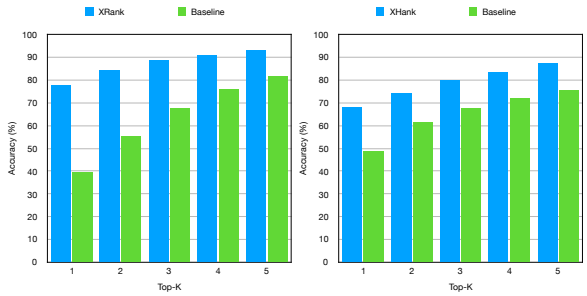

Figure 8: Recommendation Accuracy

5.1 Data Collection

The source code of most Android apps is not publicly available. With few apps with source code available, training FuzzyCatch from existing mobile app projects would be difficult and insufficient. Thus, we decided to train our models with Android bytecode. Table 3 summaries our dataset for training the models. In total, we downloaded and analyzed 13,463 top free apps from the Google Play Store. We fetched a list of top free apps from all the categories of the app store. Then, we only downloaded apps with an overall rating of at least 3 (our of 5), based on the assumption that the high-rating apps would have high code quality, and thus, would have better exception handling code. Since Android mobile apps are distributed as .apk files, our crawler unpacked each .apk file and kept only its .dex file, which contains the bytecode of the app. The total storage space for the .dex files of the downloaded apps are around 165 GB. After parsing those .dex files, we obtained about 21 million classes.

Next, we developed a bytecode analyzer that analyzed each class and looked for all methods in the class to build GROUM models. Since an Android mobile app is self-contained, its .dex files contain the bytecode of all external libraries it uses. That leads to the duplication of the bytecode of shared libraries. To remove that duplication, our bytecode analyzer maintains a dictionary of the analyzed methods and analyzed each of them only once. In the end, we analyzed over 26 million *unique* methods which have in total nearly 740 million bytecode instructions. They are used to train XRank and XHand, producing fuzzy logic rules for 261 exception types and 64,685 methods in Java and Android APIs.

5.2 Experiments on Android Bytecode

We consider the bytecode of the downloaded apps as the evaluation dataset and use 10-fold cross-validation to evaluate the accuracy of FuzzyCatch on them. We divide them into 10 folds, each contains roughly 1400 apps. At each iteration, we choose a fold for testing and the remaining for training. The final results are averaged from 10 iterations.

For each try-catch block in the code of the testing apps, we evaluated if FuzzyCatch can recommend the actual exception and reaction method calls in the catch block given the method calls in the try block. For example, if the code actually catches e and e is also in the top-k list recommended by FuzzyCatch then it is an accurate recommendation for catching exceptions. Similarly, if for an object o, the code calls method r for recovery, and r is also in the top-k recommended by FuzzyCatch then it is an accurate recommendation for reaction to exceptions.

To measure the effectiveness of FuzzyCatch, we compared it with a simple baseline. Because no other approaches have been proposed for this recommendation problem, we design the baseline as a simple approach based on the frequency of catching exception types and reaction calls. That is, we count the number of times an exception (or a repair call) appear in all try-catch blocks in the training data. Then, the exceptions (or repair calls) are ranked descendingly by their frequencies and recommended as such. For example, Exception is the top-1 exception recommended by this approach. And close is the top-1 method recommended to call for repairing (or recover) a Cursor object.

Figure 8 shows the experiment results (the left chart for XRank and the right chart for XHand). From the figure, we can see that both XRank and XHand have consistently high levels of accuracy. For example, XRank has top-1 accuracy of 77% and top-5 accuracy of 92%. For XHand, they are 69% and 87%. Also, both models significantly outperform the baselines.

5.3 Experiments on real exception bugs

We conducted two more experiments to evaluate on the effectiveness of FuzzyCatch on detecting and fixing real exception bugs.

5.3.1 Data Collection. In this section, we describe briefly our dataset of exception handling bugs and fixes. We focus on bugs that are caused by not catching exceptions in the code. For convenience, we defined those bugs as **exception bugs**. The fix for those types of bugs is called **exception bug fixes**. To build the dataset, we applied the similar steps described in [15, 33]. We first collected a dataset consists of several large open-source Android projects. For each project, we checked out its source code repository to retrieve all the code and commits. We developed an extraction tool to identify the bug fixes from the commits and issues of those projects. Next, we manually inspected all the bug fixes to identify exception bug fixes. Finally, we were able to collect a dataset of 1,000 exception bug fixes. Figure 9 shows an exception bug fix in our dataset. The dataset is available at rebrand.ly/ExDataset.

5.3.2 Detecting Exception Bugs. By calculating exception risk scores of method calls and code snippets, FuzzyCatch could warn programmers of missing exception handling code, i.e., it could detect exception related bugs. For example, if a method call having a "VERY RISKY" exception risk score is not covered by a try catch block, FuzzyCatch can warn the programmer of a potential exception bug and recommends an exception to catch.

COMMIT MESSAGE: "fix #2695: re-introduce a workaround we were using in previous versions"

```
- postContent = new SpannableStringBuilder(
-                                mEditorFragment.getSpannedContent());
+ try {
+     postContent = new SpannableStringBuilder(
+                                mEditorFragment.getSpannedContent());
+ } catch (RuntimeException e) {
+     // A core android bug might cause an out of
+     // bounds exception, if so we'll just use the current editable
+     // See https://code.google.com/p/android/issues/detail?id=5164
+     postContent = new SpannableStringBuilder(
+         StringUtils.notNullStr((String) mEditorFragment.getContent()));
+ }
```

Figure 9: An Exception Bug Fix

To evaluate FuzzyCatch on such warnings, we manually applied FuzzyCatch for the 1,000 exception bugs that we collected. We run FuzzyCatch on the buggy code version and check if FuzzyCatch produces warnings. Then, we compare the exception recommended by FuzzyCatch and the actual exception caught in the fixed version.

Table 4 lists the results. If we set FuzzyCatch to the strictest mode, i.e., only warning on method calls with 'VERY RISKY' exception risk level then it produced warnings for 734 out of 1,000 bugs (73.4%). In the loosest mode, i.e., warning on all method calls with 'LESS RISKY' exception risk level and higher, then it produced warnings for 821 out of 1,000 bugs (82.1%).

Prior experiments show that FuzzyCatch does not always recommend the exception actual used by the programmers. For example, the top-1 recommendation for Integer.parseInt is NumberFormatException. However, in many cases the programmers actually use the more general Exception. Thus, it is similarly in this experiment. In 734 cases that FuzzyCatch warns of potential exception bugs in its strictest mode, the top-1 recommended exception is actually used in 547 cases (74.5%).

When using FuzzyCatch, there could be a case in which the tool flags a warning on a method that the programmer does not want to add try-catch block or catch an exception. We defined those cases as "unwanted warnings". To evaluate if FuzzyCatch produces unwanted warnings, we selected 100 code snippets from the latest version of our subject systems. We only selected code snippets of at least 10 lines of code and having no try catch block added through the whole project history. We assumed that such code snippets will not need exception handling code. Thus, if FuzzyCatch produces a warning (of missing exception handling) for such a code snippet, it will be an unwanted warning. In a real-world, large-scale system, it is nearly impossible to prove or verify that a piece of code C is free of bugs. So we did not have a reliable method to determine if C does not need a try-catch statement. We instead looked at its revision history. Because the subject systems have a long history (some have up to 28,000 revisions), if a piece of code has not been enclosed by a try-catch statement throughout such long history then we could reasonably assume that it does not need.

In the strictest mode, FuzzyCatch produced 8 unwanted warnings (out of 100 code snippets). In the loosest mode, FuzzyCatch produced 35 unwanted warnings. Thus, this is a balanced trade-off. If the programmer wants to higher code quality (i.e., fewer exception bugs), he needs to spend more time on FuzzyCatch warnings to determine if its warnings and recommendations are necessary.

In addition, adding a try-catch statement is never wrong as it just makes the code safer. The expected loss for an "unwanted warning" is much smaller than a bug caused by a missing try-catch statement. Thus, we conservatively call recommendations of FuzzyCatch "unwanted warnings" because programmers have not yet wanted to use them rather than "false positives" meaning those recommendations are wrong.

Table 4: Recommendation Results for 1,000 Exception Bugs

Risk level	VERY RISKY	RISKY	LESS RISKY
Correct warning	734	783	821
Top-1 Accuracy	547 (75%)	587 (75%)	601 (73%)
Top-3 Accuracy	593 (81%)	618 (79%)	646 (80%)

Table 5: Results in recommending repairing actions

	FuzzyCatch	Barbosa *et al.*	CarMiner
# of fixes	437	437	437
# of matches	287 (65%)	126 (28%)	196 (44%)

5.3.3 Repairing Exception Bugs. In this experiment, we evaluate the effectiveness of FuzzyCatch in recommending repairing actions for real exception bug fixes. Not all of the exception bug fixes contains handling actions, for example, the fixer could add code to swallow exceptions, add logging messages, or re-throw another exception. We select a subset of the 1,000 exception bug fixes which includes all the exception bug fixes that contain handling actions by programmers in the catch block. In each bug fix of the subset, developers performed at least one repairing action (i.e. a method call, an assignment, etc.) in the exception handling code. Figure 10 shows an example of a bug fix in the subset. In the bug fix, the developer performed three method calls and one assignment to set value for postContent. In total, there are 437 out of 1,000 exception bug fixes have handling actions, thus, we evaluate FuzzyCatch on this subset.

For each bug fix in the dataset, we invoked FuzzyCatch to recommend repairing actions and compare the recommendation result with the corresponding fix. If the recovery actions recommended by FuzzyCatch exactly match with the fix, we count as a match. Otherwise, we consider the case as a miss.

To further evaluate the effectiveness of FuzzyCatch in this task, we compare our tool with two existing approaches presented in [5, 41]. Barbosa *et al.* [5] proposed a technique that uses exception types, method calls, and object types as heuristic strategies to identify and recommend relevant code examples with current editing handling code. Although the approach does not provide actual recommendations, the examples that it suggests could be used as a guide for programmers to fix an exception bugs. Thus, we evaluate the baseline as follows: if at least one of the top-5 relevant code examples recommended by the baseline matches with the fix, we count as a match, otherwise, we consider the case as a miss. We re-implemented the technique using the same configurations presented in the work. As the approach requires a corpus of real code examples, not bytecode, we implemented it with a dataset contains the source code of 1,514 Android projects listed on the FDroid app repository [1]. The list of all the projects in the dataset could be

found at rebrand.ly/ExDataset. Rahman *et al.* [35] proposed another approach that recommends exception handling code examples from a number of GitHub projects. We contacted the author but we could not compile their code or collect their training and experiment data. Thus, we did not compare the result with their work.

We also compare our method with CAR-miner [41], an approach that uses association mining techniques to mine association rules between method calls of try and catch blocks in exception handling code. The model was used to detect bugs related to exceptions. The mined rules have a form (S_n, S_e) in which S_n is the set of all methods in a try block while S_e is the set of repairing methods in a catch block. S_e could be used as the recommendation on handling actions. We re-implement CAR-miner with the same settings that were used in the original paper on the same Android bytecode dataset as FuzzyCatch.

Table 5 shows the evaluation result of FuzzyCatch and the baseline for the task. In a total of 437 exception bug fixes, the recommendations of FuzzyCatch match the fixes of developers in 287 cases. Overall, FuzzyCatch could provide meaningful recommendations in roughly 65% of the bug fixes. The first baseline only matches the fixes of developers in 126 (28%) cases. The association mining technique CAR-miner performs better with 196 cases (44%). Overall, the result of FuzzyCatch outperforms the baseline methods substantially.

COMMIT MESSAGE: "Merge pull request #937 from garvankeeley/bug-network"

```
- InputStream in = new BufferedInputStream(
-                       httpURLConnection.getInputStream());
  InputStream in = null;
+ try {
+     in = new BufferedInputStream(
+                       httpURLConnection.getInputStream());
+ } catch (Exception ex) {
+     in = httpURLConnection.getErrorStream();
+ }
```

Figure 10: An Example of Handling an Exception

For demonstration, Figure 10 shows an exception bug fix in which the three techniques have different recommendation results. In this example, FuzzyCatch recommends calling getErrorStream on httpURLConnection object and an assignment for the InputStream object. The result exactly match the code of the programmer as shown in the figure. CAR-Miner mined (getInputStream, disconnect) as a association rule, thus, it suggests calling the method disconnect instead of getErrorStream, which is incorrect. While, the code examples provided by the first baseline did not provide any meaningful recommendations.

6 DISCUSSIONS

In this section, we discuss several aspects of FuzzyCatch in more detail. From machine learning perspective, FuzzyCatch is a simple ensemble approach to three classification problems: Given code snippet C, (1) predict if it needs an enclosing try-catch, which is a binary classification problem; (2) predict what exception to catch, which is a multi-class prediction problem with classes are all exception types; (3) predict what method to call in the catch block, which is a multi-class predict problem with classes are all API methods. FuzzyCatch learns and predicts like Naive Bayes [18] but more flexibly. Because fuzzy membership functions are not probability distribution functions, FuzzyCatch does not need to assume the independence of predictors or normalize $\sum_e \mu_C(e) = 1$.

We could consider FuzzyCatch is a big code powered fuzzy logic system specially designed for the software engineering domain. It represents exception handling knowledge as fuzzy logic rules. It uses fuzzy set theory to model and apply those rules, and uses fuzzy union operations to combine those rules. In the traditional fuzzy logic system, variables are often continuous such as Temperature, Density or linguistic like LOW, VERY LOW. The membership functions are often manually defined by domain experts with functions such as triangular or trapezoidal. In FuzzyCatch, variables are discrete, i.e. methods, exceptions, and the membership functions are estimated automatically from millions of code examples.

FuzzyCatch is trained from a dataset contains over 13,000 highly rated Android apps. We assume that FuzzyCatch can learn common, statistically significant programming patterns from its huge collection of code. Also, it is reasonable to believe that the dataset contains high-quality code, thus, better exception handling code, because it contains apps that are developed and frequently updated by top software companies, e.g. Google, Facebook, which employs the best software engineers in the world. They are used daily by millions to billions of users worldwide, any errors are likely discovered and reported quickly. Our empirical evaluation also confirms that FuzzyCatch can be used effectively to detect and fix exception-handling bugs.

FuzzyCatch is a fully configurable, automated system. In our evaluation, we limit the scope to Android and Java APIs but the system can learn rules for methods in third-party libraries if we provide code examples using them. We plan to extend FuzzyCatch to learn exception handling patterns from the own history of software systems and programmers. If it can learn project or programmer specific rules, it could reduce the risk of "poor exception handling practices" learned from external sources.

7 THREATS TO VALIDITY

The threat to internal validity includes errors when we identified and evaluated exception bug fixes. Firstly, we might incorrectly identify bug fixes. To reduce this threat, we could apply more patterns (e.g. adding more filter keywords) when identifying bug fixes from commits. Secondly, we might incorrectly identify exception bug fixes from the bug fixes. As the evaluation process is manually carried out by three researchers, there are might be errors when reporting results due to human errors. The threat could be reduced by adding more people to our labelling and evaluation process.

The threat of external validity includes our selected projects for labeling exception bugs and fixes. Although we analyzed medium to large Android projects, the selected projects may still be limited to Android platforms. It is likely that most our results still hold for other platforms. To reduce this threat, our evaluation should be replicated using projects from other platforms or programming languages. Also, the number of exception of bug fixes should be bigger for the result more convincing. We plan to extend the exception bug fixes in our dataset in our future work.

8 RELATED WORK

Exception handling recommendation has been studied in several researches [3, 5, 6, 16, 28, 35]. Barbosa *et al.* [5] proposed a set of three

heuristic strategies used to recommend exception handling code. They also proposed RAVEN, a heuristic strategy aware of the global context of exceptions that produces recommendations of how violations in exception handling may be repaired [3]. Rahman *et al.* [35] proposed a context-aware approach that recommends exception handling code examples from a number of GitHub projects. Filho *et al.* [16] proposed ArCatch, an architectural conformance checking solution to deal with the exception handling design erosion. Lie et al. [25] proposed an approach, named EXPSOL, which recommends online threads as solutions for a newly reported exception-related bug. Kistner et al. [22] built a tool that reveals possible sources of exceptions in the code. This tool also provides project-specific recommendations and detects common bad exception handling practices. Montenegro et al. [29] proposed an "exception policy expert" tool which alerts developers about policy violations and can suggest possible handlers for the exceptions. Li et al. [23] devised a method that automatically recommends exception handling strategies, based on program context. To the best of our knowledge, FuzzyCatch is the first code recommendation tool available that actually recommends and generates exception catching and handling code as a plugin of IDEs.

There exist several methods for mining exception-handling rules. WN-miner [43] and CAR-miner [41] are approaches that use association mining techniques to mine association rules between method calls of try and catch blocks in exception handling code. Both models are used to detect bugs related to exceptions. Zhong *et al.* [44] proposed an approach named MiMo, that mines repair models for exception-related bugs. Approaches based on association rule mining mines only most frequent rules, i.e. rules that satisfy certain support and confidence. Thus, rules involving rare exceptions or methods with lower support and confidence are undiscoverable. FuzzyCatch also aims to mine programming patterns for exception handling. However, FuzzyCatch represents those patterns using fuzzy sets and combines them using fuzzy logic, making it highly robust and scalable. For example, FuzzyCatch can learn fuzzy logic rules for rare methods (shown in Section 3.1) and mine a very large code repository for patterns of 64 thousand API methods and 260 exception types.

There are several studies on exception handling. Sena *et al.* [37] presents an empirical study to investigate the exception handling strategies adopted by Java libraries and their potential impact on the client applications. Ebert *et al.* [15] presented an exploratory study on exception handling bugs by surveying of 154 developers and an analysis of 220 exception handling bugs from two Java programs, Eclipse and Tomcat. Similarly, Nguyen *et al.* [33] performed an empirical study on nearly 300 exception-related bugs and fixes from 10 mobile apps. Coelho *et al.* [11] performed a detailed empirical study on exception-related issues of over 6,000 Java exception stack traces extracted from over 600 open-source Android projects. Chen *et al.* [10] performed a comprehensive study on 210 exception related bugs from six widely-deployed cloud systems to understanding exception related bugs in large-scale cloud systems. Barbosa *et al.* [4] presented a categorization of the causes of exceptional faults observed in two mainstream open-source projects. Oliveira *et al.* [34] presents an empirical study to understand the relationship between changes in Android programs and their robustness and comparing the evolution of the exception handling code in Android and standard Java applications. Marinescu [26] showed the classes that use exceptions are more defect prone than the other classes. Padua *et al.* [14] investigated the relationship between software quality measured by the probability of having post-release defects with exception flow characteristics and exception handling anti-patterns. In [13], they studied exception handling practices with exception flow analysis. Kechagia *et al.* [19] investigated the exception handling mechanisms of the Android APIs to understand when and how developers use exceptions. In [20], they examined Java exceptions and propose a new exception class hierarchy and compile-time mechanisms that take into account the context in which exceptions can arise. In [21], they showed that a significant number of crashes could have been caused by insufficient documentation concerning exceptional cases of Android API. Bruntink *et al.* [7] provided empirical data about the use of an exception handling mechanism based on the return code idiom in an industrial setting. Coelho *et al.* [12] studied exception handling bug hazards in Android based on GitHub and Google code issues. In [27], they studied exception handling guidelines adopted by Java developers.

There are several researches focus on the global impact of exceptions [8, 17, 36, 38]. Robillard *et al.* [36] studied the current Java exception handling mechanism. They argued that exceptions are a global design problem, and exceptions source are often difficult to predict in advance. Fu *et al.* [17] presents a new static analysis that, when combined with previous exception-flow analyses, computes multiple-link exception propagation paths. Cacho *et al.* [8] presented an aspect-oriented model for exception handling implementation. Shah *et al.* [38] presents a new visualization technique for supporting the understanding of exception-handling constructs in Java programs. Different from these approaches, FuzzyCatch focuses on handling exceptions "locally" by recommending programmers to add try-catch blocks and handling actions. Previous studies showed that in many cases, programmers did not perform those actions correctly, which leads to serious bugs. FuzzyCatch is designed to assist programmers in such situations.

Fuzzy-based approaches have been proposed to solve problems in software engineering, such as bug triaging problem [39, 40], automatic tagging [2], bug categorization [9]. However, they focus on modeling textual software artifacts. To the best of our knowledge, FuzzyCatch is the first fuzzy-based approach applied to source code and bytecode.

9 CONCLUSIONS

Exceptions are unexpected errors occurring while an application is running. Learning to handle exceptions correctly is often challenging due to the fast-changing nature of API libraries and frameworks, and the insufficiency of API documentation. Prior studies show that there is a considerable number of exception-related bugs occur in software developments, and programmers still use bad practices while handling exceptions. We propose two techniques based on fuzzy logic for learning and recommending exception types and repairing actions for exception handling code. Based on the proposed techniques, we have developed FuzzyCatch, a code recommendation tool for exception handling. Our evaluation shows that FuzzyCatch can effectively learn exception handling patterns from a large repository of mobile apps and provide highly accurate recommendations.

REFERENCES

[1] [n.d.]. *https://f-droid.org/.*

[2] J. M. Al-Kofahi, A. Tamrawi, Tung Thanh Nguyen, Hoan Anh Nguyen, and T. N. Nguyen. 2010. Fuzzy set approach for automatic tagging in evolving software. In *ICSM.*

[3] E. A. Barbosa and A. Garcia. 2017. Global-Aware Recommendations for Repairing Violations in Exception Handling. *TSE* (2017).

[4] E. A. Barbosa, A. Garcia, and S. D. J. Barbosa. 2014. Categorizing Faults in Exception Handling: A Study of Open Source Projects. In *2014 Brazilian Symposium on Software Engineering*. 11–20. https://doi.org/10.1109/SBES.2014.19

[5] E. A. Barbosa, A. Garcia, and M. Mezini. 2012. Heuristic Strategies for Recommendation of Exception Handling Code. In *Brazilian Symposium on Software Engineering.*

[6] E. A. Barbosa, A. Garcia, M. P. Robillard, and B. Jakobus. 2016. Enforcing Exception Handling Policies with a Domain-Specific Language. *TSE* (2016).

[7] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. 2006. Discovering Faults in Idiom-based Exception Handling. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. ACM, New York, NY, USA, 242–251. https://doi.org/10.1145/1134285.1134320

[8] Nelio Cacho, Fernando Castor Filho, Alessandro Garcia, and Eduardo Figueiredo. 2008. EJFlow: Taming Exceptional Control Flows in Aspect-Oriented Programming. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development* (Brussels, Belgium) *(AOSD '08)*. Association for Computing Machinery, New York, NY, USA, 72–83. https://doi.org/10.1145/1353482.1353492

[9] Indu Chawla and Sandeep K. Singh. 2015. An Automated Approach for Bug Categorization Using Fuzzy Logic. In *ISEC.*

[10] H. Chen, W. Dou, Y. Jiang, and F. Qin. 2019. Understanding Exception-Related Bugs in Large-Scale Cloud Systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 339–351. https://doi.org/10.1109/ASE.2019.00040

[11] Roberta Coelho, Lucas Almeida, Georgios Gousios, and Arie van Deursen. 2015. Unveiling Exception Handling Bug Hazards in Android Based on GitHub and Google Code Issues. In *MSR.*

[12] Roberta Coelho, Lucas Almeida, Georgios Gousios, and Arie van Deursen. 2015. Unveiling Exception Handling Bug Hazards in Android Based on GitHub and Google Code Issues. In *Proceedings of the 12th Working Conference on Mining Software Repositories* (Florence, Italy) *(MSR '15)*. IEEE Press, Piscataway, NJ, USA, 134–145. http://dl.acm.org/citation.cfm?id=2820518.2820536

[13] G. B. d. Pádua and W. Shang. 2017. Revisiting Exception Handling Practices with Exception Flow Analysis. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 11–20. https://doi.org/10.1109/SCAM.2017.16

[14] Guilherme B. de Pádua and Weiyi Shang. 2018. Studying the Relationship Between Exception Handling Practices and Post-release Defects. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. ACM, New York, NY, USA, 564–575. https://doi.org/10.1145/3196398.3196435

[15] Felipe Ebert, Fernando Castor, and Alexander Serebrenik. 2015. An Exploratory Study on Exception Handling Bugs in Java Programs. *J. Syst. Softw.* 106, C (Aug. 2015), 82–101. https://doi.org/10.1016/j.jss.2015.04.066

[16] Juarez L. M. Filho, Lincoln Rocha, Rossana Andrade, and Ricardo Britto. 2017. Preventing Erosion in Exception Handling Design Using Static-Architecture Conformance Checking. In *Software Architecture.*

[17] C. Fu and B. G. Ryder. 2007. Exception-Chain Analysis: Revealing Exception Handling Architecture in Java Server Applications. In *29th International Conference on Software Engineering (ICSE'07)*. 230–239. https://doi.org/10.1109/ICSE.2007.35

[18] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2001. *The Elements of Statistical Learning.* Springer New York Inc.

[19] Maria Kechagia, Marios Fragkoulis, Panos Louridas, and Diomidis Spinellis. 2018. The Exception Handling Riddle: An Empirical Study on the Android API. *Journal of Systems and Software* 142 (04 2018). https://doi.org/10.1016/j.jss.2018.04.034

[20] Maria Kechagia, Tushar Sharma, and Diomidis Spinellis. 2017. Towards a Context Dependent Java Exceptions Hierarchy. In *Proceedings of the 39th International Conference on Software Engineering Companion* (Buenos Aires, Argentina) *(ICSE-C '17)*. IEEE Press, Piscataway, NJ, USA, 347–349. https://doi.org/10.1109/ICSE-C.2017.134

[21] 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)*. ACM, New York, NY, USA, 312–315. https://doi.org/10.1145/2597073.2597089

[22] F. Kistner, M. Beth Kery, M. Puskas, S. Moore, and B. A. Myers. 2017. Moonstone: Support for understanding and writing exception handling code. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 63–71. https://doi.org/10.1109/VLHCC.2017.8103451

[23] Y. Li, S. Ying, X. Jia, Y. Xu, L. Zhao, G. Cheng, B. Wang, and J. Xuan. 2018. EH-Recommender: Recommending Exception Handling Strategies Based on Program Context. In *2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS)*. 104–114. https://doi.org/10.1109/ICECCS2018.2018.00019

[24] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. API Change and Fault Proneness: A Threat to the Success of Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. ACM, New York, NY, USA, 477–487. https://doi.org/10.1145/2491411.2491428

[25] X. Liu, B. Shen, H. Zhong, and J. Zhu. 2016. EXPSOL: Recommending Online Threads for Exception-Related Bug Reports. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. 25–32. https://doi.org/10.1109/APSEC.2016.015

[26] Cristina Marinescu. 2011. Are the Classes That Use Exceptions Defect Prone?. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution* (Szeged, Hungary) *(IWPSE-EVOL '11)*. Association for Computing Machinery, New York, NY, USA, 56–60. https://doi.org/10.1145/2024445.2024456

[27] H. Melo, R. Coelho, and C. Treude. 2019. Unveiling Exception Handling Guidelines Adopted by Java Developers. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 128–139. https://doi.org/10.1109/SANER.2019.8668001

[28] T. Montenegro, H. Melo, R. Coelho, and E. Barbosa. 2018. Improving developers awareness of the exception handling policy. In *SANER.*

[29] T. Montenegro, H. Melo, R. Coelho, and E. Barbosa. 2018. Improving developers awareness of the exception handling policy. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 413–422. https://doi.org/10.1109/SANER.2018.8330228

[30] 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 the 7th 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, New York, NY, USA, 383–392. https://doi.org/10.1145/1595696.1595767

[31] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. [n.d.]. Learning API Usages from Bytecode: A Statistical Approach. In *the 38th International Conference on Software Engineering (ICSE '16).*

[32] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. 2015. Recommending API Usages for Mobile Apps with Hidden Markov Model. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on.*

[33] Tam The Nguyen, Phong Minh Vu, and Tung Thanh Nguyen. 2019. An Empirical Study of Exception Handling Bugs and Fixes. In *Proceedings of the 2019 ACM Southeast Conference* (Kennesaw, GA, USA) *(ACM SE '19)*. Association for Computing Machinery, New York, NY, USA, 257–260. https://doi.org/10.1145/3299815.3314472

[34] Juliana Oliveira, Nelio Cacho, Deise Borges, Thaisa Silva, and Fernando Castor. 2016. An Exploratory Study of Exception Handling Behavior in Evolving Android and Java Applications. In *Proceedings of the 30th Brazilian Symposium on Software Engineering* (Maringá, Brazil) *(SBES '16)*. Association for Computing Machinery, New York, NY, USA, 23–32. https://doi.org/10.1145/2973839.2973843

[35] M. M. Rahman and C. K. Roy. 2014. On the Use of Context in Recommending Exception Handling Code Examples. In *SCAM.*

[36] Martin P. Robillard and Gail C. Murphy. 2000. Designing Robust Java Programs with Exceptions. *SIGSOFT Softw. Eng. Notes* 25, 6 (Nov. 2000), 2–10. https://doi.org/10.1145/357474.355046

[37] D. Sena, R. Coelho, U. Kulesza, and R. Bonifácio. 2016. Understanding the Exception Handling Strategies of Java Libraries: An Empirical Study. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 212–222.

[38] Hina Shah, Carsten Görg, and Mary Jean Harrold. 2008. Visualization of Exception Handling Constructs to Support Program Understanding. In *Proceedings of the 4th ACM Symposium on Software Visualization* (Ammersee, Germany) *(SoftVis '08)*. Association for Computing Machinery, New York, NY, USA, 19–28. https://doi.org/10.1145/1409720.1409724

[39] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen. 2011. Fuzzy set-based automatic bug triaging: NIER track. In *ICSE.*

[40] Ahmed Tamrawi, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2011. Fuzzy Set and Cache-based Approach for Bug Triaging. In *ESEC/FSE.*

[41] 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)*. Washington, DC, USA.

[42] Westley Weimer and George C. Necula. 2004. Finding and Preventing Run-time Error Handling Mistakes. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (Vancouver, BC, Canada) *(OOPSLA '04)*. ACM, New York, NY, USA, 419–431. https://doi.org/10.1145/1028976.1029011

[43] Westley Weimer and George C. Necula. 2005. Mining Temporal Specifications for Error Detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*. Berlin, Heidelberg.

[44] 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