Where to Handle an Exception? Recommending Exception Handling Locations from a Global Perspective

Xiangyang Jia*‡, Songqiang Chen*, Xingqi Zhou*, Xintong Li*, Run Yu†, Xu Chen*‡, Jifeng Xuan*
*School of Computer Science, Wuhan University, China
†Tandon School of Engineering, New York University, USA
jxy@whu.edu.cn, i9schen@gmail.com, 2017302580085@whu.edu.cn, lxtong@whu.edu.cn
run.yu@nyu.edu, xuchen@whu.edu.cn, jxuan@whu.edu.cn

***Abstract*—Exception handling is an effective mechanism to guarantee software reliability in modern programming languages. An exception interrupts the program execution and propagates backwards along the call chain until the exception is caught by an exception handler. In software development practices, developers may be confused in determining where to place the exception handler in the call chain. The reason is that exception handling requires a developer to take a comprehensive consideration from a global perspective of the software project. In this paper, we propose an automatic approach EHAdvisor, which recommends exception handling locations from the global perspective of the project. EHAdvisor first trains a binary classification model based on four types of features, including architectural features, project features, functional features, and exception features. Then, for a new code snippet with exceptions, EHAdvisor predicts the exception catching probability for each method in the call chain based on the classification model and recommends Top-K exception handling locations based on the probability ranking. We conducted experiments on a dataset from 29 high-quality open source projects. Experimental results show that EHAdvisor achieves an average Top-1 recommendation success rate of 70.83% for across-project location recommendation and an average Top-1 accuracy of 86.21% for intra-project recommendation. Experiments on the importance scores show that global features, such as project features and architectural features, are evidently important to the recommendation of exception handling locations.**

***Index Terms*—exception handling, location recommendation, machine learning, global features, classification algorithm**

I. Introduction

Most modern programming languages rely on exception handling mechanisms to deal with abnormal events [1]. Once an error occurs inside a method, the method throws an exception and interrupts the normal flow of program execution. The runtime environment then searches the call stack of the method for exception handlers (i.e., code blocks that catch the thrown exception). The search process starts from the method where the error occurs and iterates through the call chain of the method. If an exception handler in a method is found in the call chain, the search process ends.

To handle an exception, a developer needs to decide where to write the exception handler in the call chain (i.e., a sequence of methods of method invocations) of the exception source. An exception can propagate through multiple methods in the call chain, and sometimes through multiple classes and packages. Therefore, exception handling relies on the *global understanding* of the design in a project [2]. Yet, developers may be confused in determining the location of handling exceptions among methods in the call chain.

Novice developers usually choose to ignore exceptions and not handle them in any method [3]. This leads to system crashes if one or more exceptions are not caught. Besides, if the exception is caught at an improper method which cannot cope with the error, the bug of crashing may also be difficult to be fixed [4].

In fact, improper locations of exception handling is one of the major reasons for software failures. Chen et al. [5] found that about 13% of bugs are caused by exception handling in six widely-deployed cloud-based systems, such as Cassandra, Hbase, and Hadoop MapReduce. Sena et al. [6] found 20.71% of exception handling related bugs in popular Java libraries such as Log4j and Maven-Filtering. Among the exception-handling bugs, 85% to 90% of faults stem from violations in the propagation chains of exceptions [7], [8].

However, it is challenging for developers to choose where to place the exception handlers in a call chain. One survey [9] shows that the developers need to consider handling exceptions in appropriate layers of software projects as well as the propagation of exceptions. They are also required to understand the design of exception flows and ensure that potential exception handling meets the expected design [2]. In other words, choosing locations to handle exceptions requires a comprehensive understanding from the *global perspective* of the software.

In this paper, we propose an automatic approach EHAdvisor, to learn exception handling patterns from high-quality projects and recommend appropriate locations of exception handlers in new projects. EHAdvisor first trains a binary classification model with the architectural features, project features, functional features, and exception features extracted from high-quality source code. Among the features, architectural features, project features, and exception features are designed from a global perspective of the project. Given a code snippet that throws a new exception, EHAdvisor then predicts the exception catching probability for each method in call chain and recommends Top-K exception handling locations based on

‡Xiangyang Jia and Xu Chen are the corresponding authors.

the probability ranking. Experiments on a dataset consist of 29 high-quality open source projects show EHAdvisor achieves an average Top-1 recommendation success rate of 70.83% for across-project location recommendation and average Top-1 accuracy of 86.21% for intra-project recommendation.

The contributions of this work are summarized as follows:

- We propose an automatic approach, EHAdvisor, to recommend locations to handle exceptions for developers. To the best of our knowledge, this is the first work on location recommendation of exception handling.
- We propose four types of features to encode the exception samples, among which architectural features, project features, and exception features are global features. These features encode candidate methods from a global perspective to enhance the machine-learning model in distinguishing the patterns of exception handling.
- We evaluated EHAdvisor on a dataset with 29 high-quality open source projects. Experiments show that EHAdvisor outperforms the baselines and achieves an average Top-1 recommendation success rate of 70.83% for across-project recommendation and 86.21% for intra-project recommendation.

The rest of this paper is organized as follows. Section II presents the problem definition and basic terms used in this paper. Section III describes the overview of the EHAdvisor approach, as well as the technical details of feature engineering, the classification model, and the recommendation algorithm. Section IV and Section V presents experimental setup and evaluation results of EHAdvisor. Section VI discusses the threats to validity of this work. Section VII lists and discusses the related work. Finally, Section VIII concludes this paper.

II. Problem Definition and Motivation

This section gives the formal definition of the problem and the motivation of our work.

A. Problem Definition

Definition 1. (Exception Source) An exception sources is a statement that signals an exception. Two types of statements are regarded as exception sources: (1) the *throw* statement in the source code that explicitly throws an exception, and (2) the call statement which invokes a method in an external class library that throws an exception.

For example, *throw new IOException()* in one Java code snippet is an exception source. Meanwhile, the file writing statement *fileWriter.write(text)* is also an exception source, because it calls the method *write* of the class *FileWriter* in JDK that throws an IOException. Exception sources in this paper are limited to the source code. The statements in third-party libraries that signal exceptions are not considered to be exception sources.

Definition 2. (Call Chain of an Exception Source) Given an exception source *es*, the call chain is represented as a sequence of methods $< m_1, m_2, ..., m_n >$, where m_1 is the method where *es* is located, m_i is called by m_{i+1} $(i = 1...n)$, and m_n is a method not called by any method in the source code.

The difference between a call chain and a call stack is that a call stack is a stack of methods in runtime, while a call chain is a sequence of methods in the source code. In the field of static analysis, a *call graph* is used to represent the invocation relationships of methods in a program. A call chain is obtained by walking along a path in the call graph, starting from the method that signals the exception (for technical details on call chain construction please refer to section IV-B). In a project, a method may be called by multiple methods in the call graph. Therefore, one exception source can be mapped to multiple call chains.

Definition 3. (Exception Handler) An exception handler is a code block that catches and handles specific types of exceptions. It is represented as (m, E, h), where m is the method that catches the exceptions, E is a set of exception types caught by this handler, and h is the code that handles the exception.

A typical exception handling code block usually consists of a *try*$\{$*protected code*$\}$ section and one or more *catch(e)*$\{$*handling code*$\}$ sections. The code in the try statement is the protected code of the exception handlers. The catch section is the exception handler, which catches specific types of exceptions and handles them with the code inside.

Definition 4. (Exception Handling Location) Given an exception source es that signals an exception e and let its call chain to be $S =< m_1, m_2, ..., m_n >$, the exception handling location L refers to the method in S that catches the exception e. Thus, the exception handler for the location L is (L, E, h), where $e \in E$.

The exception handling locations defined in this paper are only at the method level, and are not detailed to code lines. In addition, exception handling locations are only allowed to be selected from the methods in the call chain.

Definition 5. (Recommendation of Exception Handling Locations) Given an exception source es that signals an exception e, and let its call chain to be $S =< m_1, m_2, ..., m_n >$, the target of exception handling location recommendation is to identify the Top-K results from $n + 1$ candidate locations, including n methods $< m_1, \dots, m_n >$, and an additional location *"No-Handler"*. *"No-Handler"* indicates that an exception e is not handled by any method in the call chain.

For example, given a call chain $< m_1, m_2, m_3 >$ with three methods, the Tok-3 recommendation is to choose three out of four locations: $\{m_1, m_2, m_3, No - handler\}$, and then rank them according to the recommendation probabilities.

B. Motivation

Senior developers handle exceptions from a global perspective. On one hand, a senior developer considers the exception type, which represents the potential type of errors. In this

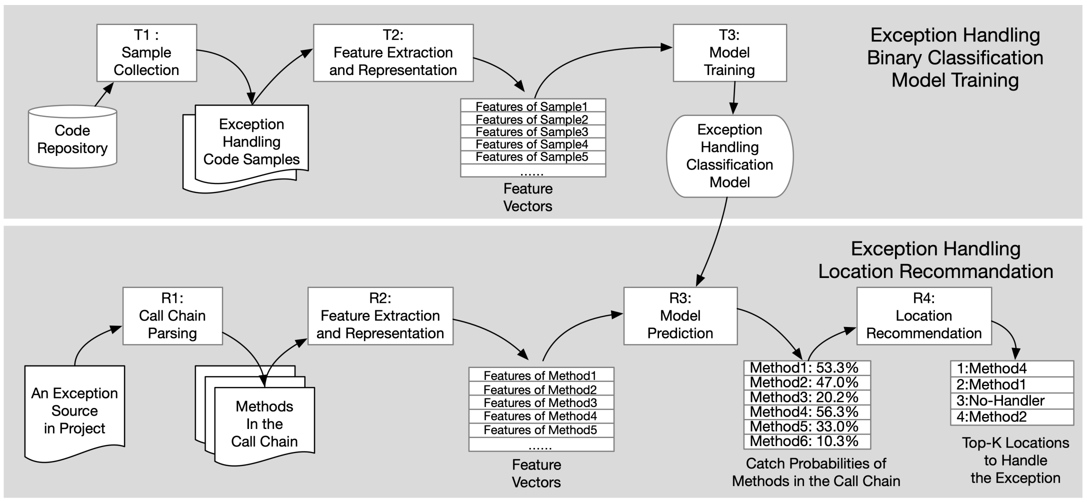

Fig. 1. Overview of EHAdvisor with two stages: exception handling binary classification model training and exception handling location recommendation.

case, different exception types require different strategies of exception handling. On the other hand, the developer considers the layers or locations for all methods in the call chain as well as their functionalities [9]. For example, some exceptions are expected to be caught at a method to restore the system state, some exceptions need to be thrown at the top-level method to notify the client, and some exceptions should be caught at the first method to log for troubleshooting.

In addition, the developer also takes into account the project type because different project types may adopt different exception handling ways. For example, the projects of class libraries should directly return error information to the client and therefore tend to throw exceptions without catching while the midware platforms are expected to avoid any crashes and therefore tend to catch and log exceptions for troubleshooting.

In a word, senior developers solve the problem of exception handling by comprehensively analyzing the system from a global perspective, including the architectural features and functional features of methods in the call chain, as well as project features and exception features. Based on these features, the senior developers finally choose a method from the call chain to place the exception handler.

Motivated by the use of the global features during exception handling in real-world development, we propose an approach that learns patterns of exception handling locations from high-quality projects. This approach extracts the architectural features, project features, functional features, and exception features to train a classification model. Developers can then use the trained model to predict the probability of exception handling for each method in the call chain. The method with the highest probability is recommended to the developers.

III. Proposed Approach

In this section, we present the overview, the feature engineering, the classification model, and the recommendation method in our work.

A. *Overview of EHAdvisor*

EHAdvisor is a machine learning based approach that learns general exception handling patterns from the code of high quality projects, and recommends Top-K best handling locations in the call chain for exception sources in new projects. Fig. 1 shows the overview of EHAdvisor with two phases: the binary classification model training phase and the location recommendation phase.

In the binary classification model training phase, EHAdvisor learns the exception handling patterns from high-quality code repositories. This phase consists of three major steps, namely T1 to T3. In T1 (sample collection), EHAdvisor searches code repositories, identifies the exception source, constructs the call chains of the exception source, and extracts code snippets of methods in the call chains. Each method with an exception is considered an *sample*. If the method of the sample contains an exception handler for the exception, the sample is labeled “catch”, otherwise is labeled “throw”.

In T2 (feature extraction and representation), EHAdvisor extracts architectural features and functional features from code snippet of methods, project features from project documents and dependency files, and exception features from class hierarchies of exceptions. These features are then encoded into feature vectors for training. In T3 (model training), EHAdvisor trains a binary classification model. The feature vectors and the labels (“catch”/“throw”) of the samples are fed into model as input. If a feature vector encodes a method m and an exception e, and is labeled “catch”, it means method m has an exception handler for e. After that, a trained classification model that can predict the exception catching probability for each method in the call chain is obtained.

In the phase of exception handling location recommendation, given an exception source in the source code, EHAdvisor recommends the methods in the call chain according to their exception catching probabilities. This phase consists of four major steps, R1 to R4. In R1 (call chain parsing), the source code is parsed and a corresponding call graph from the exception source is constructed, whose paths are converted into a set of call chains. In R2 (feature extraction and representation), features of all methods in the call chain as well as features of the project and the exception are extracted and encoded in the same way as in the model training phase, where each sample is represented as a feature vector. In R3 (model prediction), the feature vectors of methods are fed into the binary classification model to predict the exception catching probabilities of all methods in call chains. In R4 (location recommendation), the exception catching probabilities of the methods are ranked to give the Top-K recommendations for each call chain.

B. Feature Engineering

EHAdvisor extracts four types of features for each sample (a method with an exception). The features encode the context of samples from the global perspective of the project. Specifically, the global features for each sample contain its relative locations in the architecture, the characteristics of the project, and the inheritance hierarchy of the exception class.

1) Architectural Features: It is important to take method locations in software architecture into account when developers decide whether a method should handle the exception or not. For example, in the design of exception handling, database exceptions should be dealt with in the business layer of the software, while the business exceptions are required to be caught in the user interface layer to notify users of error messages [9].

Therefore, we design the architectural feature for each method in the call chain to represent the global location of the method in software architecture. If multiple methods in different projects are in a similar location, the distance of their architectural feature vectors should be close.

Specifically, EHAdvisor uses three relative locations as architectural features: the location of the method in the class, the location of the class in the package, and the location of the package in the project. Each relative location is represented as two distances: the distance to the top layer and the distance to the bottom layer. Fig. 2 shows an example of architectural features of methods in a call chain. There are five methods in the call chain. The architectural feature of method Package1.Class1.Method1 is represented as (0,1,0,0,0,2). The first two dimensions represent the method location (in terms of call relationship) in the class. Since Method1 is on the top in Class1, the distance to the top layer is 0 and the distance to the bottom method Method2 is 1. Therefore, these two dimensions are represented as (0,1). The third and fourth dimensions represent the location of class in package. Since Class1 is both the bottom layer and the top layer in Package1, it is represented as (0,0). The fifth and sixth dimensions represent the package location in project. Package1 is the top package in the call chain, and the distance to the bottom package Package3 is 2. Then these two dimensions are (0,2).

We construct a call graph for the exception source and calculate the features for each method in the call graph to synthesize the architectural features. If a method is included by multiple paths (i.e., call chains) in the call graph, the shortest path is chosen to calculate the architectural feature.

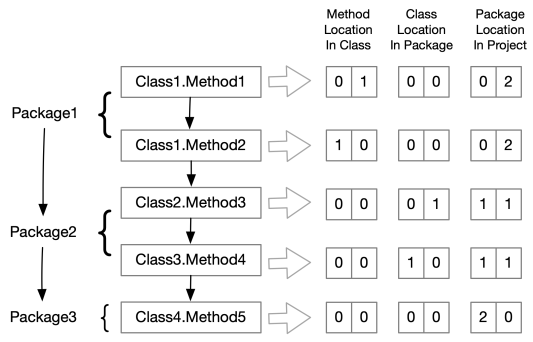

Fig. 2. An example of architectural features of methods in a call chain.

2) Project Features: As mentioned in Section II-B, the design of exception handling flows also relies on the project types. Therefore, we extract and encode project information into project features. The project features are represented based on functional and technical characteristics of the project.

The functional characteristics of the project are represented by the description of the project, which is encoded into feature vectors using the Doc2Vec algorithm [10]. In this way, if two projects have similarity functionality, the number of semantically-similar words will be high in the description. As a result, the distance between the feature vectors obtained by Doc2Vec algorithm is close.

The technical characteristics of the projects are extracted based on libraries and frameworks that the projects depend on. To some extent, the framework used by the project represents the technical architecture of the project. For example, if both of the two Java projects use the Spring Boot framework, the projects may be at similar architectural layers. This may lead to common practices in exception handling: both projects are likely to use global exception handling in the Controller layer.

Libraries or frameworks of a project are vectorized with multi-hot encoding [11], where each bit in the vector represents the usage of a library or a framework. For example, suppose we use a 128-bit feature vector whose 8th, 13th and 15th bit represent Spring framework, Guava library, and Apache Common IO library, respectively, a project that depends on Spring and Guava but does not depend on Apache Common IO will be encoded with a vector whose 8th and 13th bit are set to 1 and the 15th bit is set to 0. However, not all the libraries contribute to help EHAdvisor recognize project characteristics. Specifically, on one hand, some libraries (e.g., JUnit) are used in almost all projects, which make them useless to distinguish the project characteristic. On the other hand, some libraries are rarely used and hence do not have generalization capability as features. To reduce the number of dimensions, we ignore such libraries in the embedding.

3) Functional Features: The functionality of a method affects the decision of whether a method handles exceptions or not. For example, a query method prefers to catch exceptions propagated from low layer methods while a file manipulation method tends to throw exceptions to the client. Therefore, we encode the functionality of the method into features. Functional features are encoded based on the textual information of the method, including the method name, method arguments, method comments, and its class name. These text data are put together and encoded with the Doc2Vec algorithm to obtain the functional feature vector of the method. Many method names, arguments and class names are composed of multiple words, which are concatenated together according to the naming convention like the Hungarian notation or the Camel notation [12]. Therefore, we break them into multiple words according to the naming convention in the Doc2Vec embedding.

4) Exception Features: The exception type is an important factor that affects the way of exception handling. For example, an IllegalArgumentException is caused by invalid arguments coming from the client method and hence it is usually tend to be thrown to the client.

To make exception feature generalizable for across-projects recommendation, the exception feature encodes the class inheritance hierarchy of the exception. For example, if ExceptionA inherits ExceptionB and ExceptionB inherits ExceptionC, then the feature of ExceptionA contains the information of ExceptionA, ExceptionB, and ExceptionC. Specifically, the exception feature encodes the exception class and its ancestor exception classes with multi-hot encoding [11], where each bit in the feature vector represents an exception type. In this manner, even though two exceptions are customized exception classes in different projects, their feature vectors can be close to each other if they inherit the same parent class.

C. Classification Model

As introduced in Section III-A, to predict the probability to catch the exception for each method in the call chain, EHAdvisor trains a classification model. The model is a binary classification model whose input is a vector representing the architectural and functional features of the sampled method, the feature of the project where the method belongs to, and the feature of the exception to be handled.

This model predicts the probabilities of “catch”, and classifies the sample into one of two groups “catch” and “throw”. If the catch probability is higher than 50%, the prediction result is considered as “catch”.

Theoretically, any classification algorithm can be used to train a binary classification model of exception handling, such as classification algorithms Decision Tree [13], Random Forest [14], SVM [15], KNN [16], DNN [17].

The performance of classification algorithms also depends on the hyper-parameter optimization. To reduce the impact of model selection and hyper-parameter tuning on the experimental results, AutoML (Automated Machine Learning) tools can be used in the training the classification model.

In our experiments, we adopt AutoGluon [18] to automatically construct classification models and tune the hyper-parameters for training. AutoGluon is an open source AutoML tool released by Amazon. It enables easy-to-use AutoML with automatic hyper-parameter tuning, model selection, architecture search, and data processing. In general, AutoGluon automatically builds an integrated learning model that contains several sub-models such as random forest and SVM.

We have tried various classical classification algorithms in our experiment, and it shows that AutoGluon preforms better than others in the most cases. For example, the average F1 values of cross-project classification are 0.5430 for AutoGluon, 0.5358 for Decision Tree, 0.4086 for Random Forest, 0.3766 for SVM, 0.4293 for KNN, and 0.5017 for DNN.

D. Exception Handling Location Recommendation

With the trained binary classification model of exception handling, we recommend the location for an exception source. Algorithm 1 shows the whole process of exception handling location recommendation. The input of the algorithm consists of an exception source es, an exception e, and an exception handling classification model M. The output of the algorithm is the Top-K recommendation results for all call chains of es.

Algorithm 1 Exception handling location recommendation

Input es: exception source, e: exception, M: classification model
Output R: Top-K results for call chains

```
1)  Map<Chain,TopK> R= $\varnothing$;
2)  CG = constructCallGraph($es$);
3)  *for* chain $\in$ CG.Paths{
4)    ProbMap = M.predict(chain.methods);
5)    ResultArray = SortByProbability(ProbMap);
6)    *for* i from 0 to ResultArray.length-1
7)        if ResultArray[i] $<$ 0.5 break;
8)    Insert “No-Handler” into MethodArray as the ith element;
9)    TopK = ResultArray.SubArray(0,K);
10)   R.put(chain,TopK);
11) }
```

The first line of the algorithm initializes an empty map to store the recommended results of each call chain. The second line parses the source code starting from the exception source es and constructs a call graph CG for es. In the call graph, each path from the top-level method to the exception source is a call chain. Lines 3-11 process the call chains in the call graph and recommend the k most suitable exception handling locations for each call chain.

To recommend the exception handling location in a call chain, Line 4 predicts the probability of catching an exception for each method in the call chain using the classification model M. Line 5 ranks the methods based on the probability of each method to get an array of methods. This array is not the final result. An additional option No-Handler should be inserted into the array. Therefore, Lines 6 and 7 of the algorithm get the first element whose catch probability is smaller than 0.5 from the array. Then the *No-Handler* option is inserted into the array at this location (Line 8). Finally, the algorithm outputs the first K elements from the array as the Top-K results of the call chain (Line 9 and 10).

As an example shown in Fig. 1, there are six methods in the call chain. The prediction results are {Method1: 53.3%, Method2: 47.0%, Method3: 20.2%, Method4: 56.3%, Method5: 33.0%, Method6: 10.3%}. To get Top-4 recommendation results, we rank these methods according to their probabilities. Afterwards, sorted results are as follows: {Method4, Method1, Method2, Method5, Method3, Method6}.

Since the probabilities of Method2, Method5, Method3, and Method6 are all less than 50%, the classification model predicts that these methods are more likely to throw the exception. Therefore, the third best option of exception handling after Method4 and Method1 should be "No-Handler", i.e., all methods do not handle the exception and let it be thrown by the top-level method. Finally, we get the Top-4 recommendation result $\{1: \text{Method4}, 2: \text{Method1}, 3: \text{No-Handler}, 4: \text{Method2}\}$ for this call chain.

This algorithm returns Top-K recommendation for each call chain in the call graph. As a reminder, since one method can be included in multiple call chains, the recommendation result of a method in multiple chains is possible to be conflicted. The developers need to choose one from the K results for each call chain and guarantee the chosen handling locations in all call chains are not conflicted.

IV. Experimental Setup

A. Research Questions

EHAdvisor aims to help developers select locations for exception handling in daily development. Therefore, we propose the following three research questions in our experiments to evaluate the performance of EHAdvisor.

- **RQ1:** How effective is EHAdvisor in handler location recommendation?
- **RQ2:** How effective is EHAdvisor in catching probability prediction on each method?
- **RQ3:** How useful is the four types of features extracted in feature engineering?

First, RQ1 is designed to empirically measure the effectiveness of the location recommendation. Since the recommendation in EHAdvisor relies on the prediction of positive samples, RQ2 is then designed to evaluate the effectiveness of such prediction. Besides, RQ3 is designed to evaluate the impacts of features on the performance of EHAdvisor.

B. Dataset

Samples in the experiment comes from 29 high-quality Java projects in two open-source communities, namely Apache and GitHub. We chose Java programs in our experiments because there is a well-designed mechanism of exception handling in Java language and the open source community of Java projects provides many popular and high-quality projects.

Table I shows the details of the projects in the experiment. All chosen projects are popular projects in Apache and GitHub, including server applications, libraries, and frameworks. We downloaded the source code of 29 projects as well as the project description files and their Maven build files. The source code was parsed and processed using Eclipse AST Parser tool, JDT [19].

We construct a call graph for each projects firstly. In our work, recall is the most important metric in call graph construction. However, literature [20] reports that existing call graph construction approaches have an average recall of 0.8840 on 31 real-world Java programs. The fairly low recall will cause broken exception propagation chains, which in turn lead to imprecise exception handling location.

TABLE I
SUMMARY OF 29 PROJECTS IN THE EXPERIMENT

Category	Project	Abbreviation	#Sample
Library	C3P0	C3P0	762
	Apache Commons Collections	Collections	755
	Apache Commons DBCP	DBCP	415
	Fastjson	FastJson	361
	Google Gson	Gson	366
	Google Guava	Guava	2196
	Apache HttpComponents Client	HttpClient	442
	Jackson	Jackson	968
	Apache Commons Logging	Logging	54
Framework	Apache Dubbo	Dubbo	2061
	Apache Flink	Flink	10863
	Grizzly	Grizzly	2004
	HikariCP	HikariCP	230
	Jersey	Jersey	1888
	Spring Data JPA	JPA	60
	Apache Log4J	Log4J	1351
	MyBatis	MyBatis	609
	Apache Shiro	Shiro	486
	SLF4J	SLF4J	116
	Apache Storm	Storm	4524
	Apache Struts	Struts	1858
	XNIO	XNIO	1137
Midware/ Platform	Apache ActiveMQ	ActiveMQ	9982
	Alibaba Druid	Druid	4002
	Apache HBase	HBase	11074
	Netty	Netty	4879
	Apache RocketMQ	RocketMQ	1308
	Apache Tomcat	Tomcat	5925
	Apache ZooKeeper	ZooKeeper	1388

Therefore, we adopt an approximation method to construct call graph to improve its recall by scarifying its precision. For method calls involving abstract methods and interfaces (this situation is prevalent in projects adopting dependency injection frameworks), we added call relations for all their sub-classes. In addition, the edges of recursive calls in the call graph are removed to guarantee that the same method does not appear multiple times in the call chain.

This approximation approach guarantees the connectivity of exception propagation chains, but produces false call chains that may not exist. However, in terms of exception handling, the methods in false chains have similar global context with the methods in real chains. Therefore, we think they can be considered as up-sampled samples during training.

Call chains are constructed by walking along each edges in the call graph starting from exception sources. For each method in a call chain, we extracted source code, architectural locations of the method, project documents, and exception type hierarchy. The label (i.e., throw or catch) of this method was also collected. Finally, we obtained a dataset that contains 72,064 samples for training.

C. Experiment Scenarios

We set up three experiment scenarios to mimic the use cases of EHAdvisor in software development.

- **Across-project recommendation**. This scenario simulates the development process of a new project with little exception handling code in the project. EHAdvisor can help developers determine the locations of exception handling according to classification models trained from other projects. The experimental design of this scenario is as follows: for each project, the classification model is tested on the samples of one project and is trained using the other 28 projects.
- **Intra-project recommendation**. This scenario simulates the development process of a project that contains adequate and well-designed exception handling code. A new developer in the development team can use EHAdvisor to recommend exception handling locations that conform with existing exception handling patterns. For a given target project, we randomly select $9/10$ of samples from this project as the training set and the remaining $1/10$ of samples are used as the test set.
- **Intra-project recommendation with a pre-trained model (abbreviated as *Intra-project+*)**. This scenario simulates the development process of a project that does not contain sufficient high-quality exception handling code. EHAdvisor is used to help developers both keep conformity to existing code patterns and provide exception handling patterns from other projects. For a given target project, we fine-tune the model that is pre-trained on the other 28 projects. We choose $9/10$ of samples from this project as the training set to fine-tune a classification model and the remaining $1/10$ of the samples are used as the test set to evaluate the final model.

D. Classification Model Training

As introduced in Section III-C, we use an AutoML tool, AutoGluon, to train binary classification models of exception handling. AutoGluon can automatically construct a classification model for our dataset and then train the model with automatically tuned hyper-parameters to achieve effective performance [18]. During the training, AutoGluon automatically constructs a classification model that composed with 23 sub-models based on ensemble learning framework. The types of the sub-model include RandomForestClassifier, K-NeighborsClassifier, LightGBMClassifier, CatboostClassifier, and NeuralNetClassifier.

We perform all the training and testing on a computer with an Intel Core i7-6850K CPU and 64GB RAM.

E. Performance Metrics

1) Recommendation Metric: In our experiment, the performance metric of exception handling location recommendation is Top-K Success Rate ($SuccRate@K$) [21], which is widely-used to evaluate recommending systems. The Top-K Success Rate is calculated as:

$$SuccRate@K = \frac{NumberOfSucc@K}{NumberOfSamples}$$

The $NumberOfSamples$ in the above formula is the total number of samples of the call chain in testing while $NumberOfSucc@K$ is the number of samples whose first K recommendation contains the correct exception handling location.

In our experiments, we set the K from 1 to 3. The reason is that the average length of call chains is between 3 and 4 in our dataset. For example, if K is set to 3, the recommended success rate has already been higher than 90% in 28 projects under the scenarios of intra-project recommendation and intra-project+ recommendation.

2) Classification Metric: We used the accuracy metric to measure the overall effectiveness of classification. We adopted precision, recall, and F1-score as the performance metrics of the positive samples, i.e., the "catch" sample. The four metrics in the evaluation are calculated as follows:

$$Accuracy = (TP + TN)/(TP + FP + TN + FN)$$

$$Precision = TP/(TP + FP)$$

$$Recall = TP/(TP + FN)$$

$$F1 = 2 \times Precision \times Recall/(Precision + Recall)$$

where TP, FP, TN, FN are the number of True Positives (correct prediction for "catch" samples), False Positives (incorrect prediction for "catch" samples), True Negatives (correct prediction for "throw" samples), and False Negatives (incorrect prediction for "throw" samples) in the prediction results, respectively.

V. Evaluation Results

A. Recommendation Performance (RQ1)

To evaluate the recommendation performance of the model, we used EHAdvisor to recommend exception handling locations for 29 projects in 3 scenarios. In each scenario, we trained 29 models for the projects respectively and conducted cross-validation. Thus, we performed 87 (29 $\times$ 3) times of training and testing. In across-project recommendation, the test set of each project contains all call chains. In intra-project/intra-project+ recommendation, the test set of each project contains 1/10 call chains of the project.

We first compared the micro-average result of EHAdvisor and the baseline. The micro-average success rate is calculated with the number of successful samples and the total number of samples in 29 projects. It indicates the overall performance of one method over all the projects. The baseline in comparison is designed based on the random strategy. It calculates the micro-average of the success rate of all call chains, where each option in one call chain is randomly selected.

Table II shows the micro-average success rate in three scenarios. As shown in Table II, EHAdvisor outperforms the baseline in all three scenarios. This indicates that EHAdvisor is reliable for developers to recommend reasonable exception handling locations. We also found the performance differences under three scenarios. We first noticed that the micro-average success rate of Top-1 in across-project recommendation scenario is 0.7083. This means that patterns learnt from other projects are useful for the new project. We then found

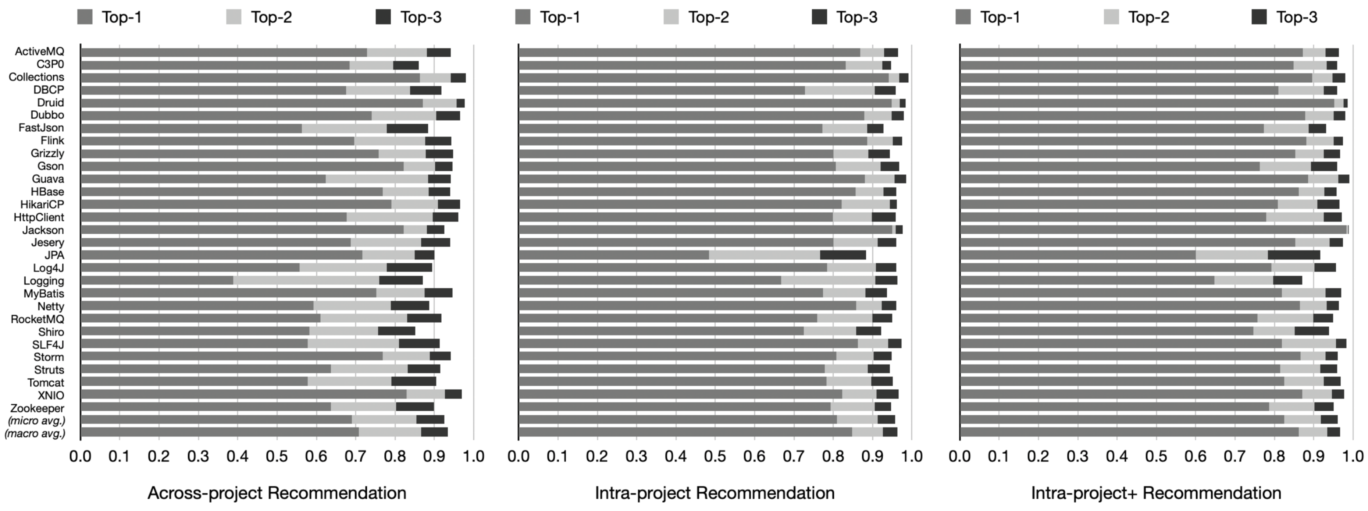

Fig. 3. Recommendation Result on Each Project in Three Experiment Scenarios in Terms of Top-K Success Rate (K=1, 2, 3).

TABLE II
RESULT COMPARISON BETWEEN EHADVISOR IN THREE SCENARIOS AND THE BASELINE

Scenario	Top-1 Success Rate	Top-2 Success Rate	Top-3 Success Rate
Across-Project	0.7083	0.8655	0.9341
Intra-Project	0.8481	0.9260	0.9627
Intra-Project+	0.8621	0.9348	0.9671
Random*	0.3813	0.7626	0.8943

*: Performance of the random baseline is the same under the three scenarios.

that intra-project recommendation with pre-trained models (abbreviated as "Intra-Project +") obtains a higher micro-average success rate (0.8621) than the original intra-project recommendation (0.8481). This indicates that intra-project recommendation with pre-trained model can both ensure the conformity to exception handling polices of the team and benefit from the knowledge from other projects.

Fig. 3 shows the recommendation result of each project in three scenarios. The result indicates that EHAdvisor performs well in both across-project and intra-project recommendation. In the across-project recommendation scenario, the Top-1 success rates of 24 projects are over 60% and the Top-1 success rates of 14 projects are over 70%. The Top-2 success rates of all projects are over 80% and the Top-3 success rates of 24 projects are over 90%. In the intra-project recommendation scenario, the Top-1 success rates of 20 projects are over 80% and the Top-3 success rates of 23 projects are over 90%. In the scenario of intra-project recommendation with pre-trained models, the Top-1 success rates of 22 projects are over 80%; the Top-2 success rates of 25 projects are over 90% Top-2; the Top-3 success rates of 27 projects are over 95%.

B. Binary Classification Performance (RQ2)

RQ2 requires a further evaluation of the performance of the classification model. We introduce three baselines based on three simple exception handling strategies. Specifically, "All-Throw" classifies all samples as "throw", "All-Catch" classifies all samples as "catch", and "Random" selects "throw" and "catch" with a 50-50 chance.

Among four metrics in Section IV-E2, accuracy is the overall metrics for classification. At the meantime, Precision, Recall, and F1-score are metrics for samples with the positive samples, i.e., labeling as "catch". F1-score is the overall metric via weighting Precision and Recall. An effective classification requires both a high accuracy value and a high F1-score.

Table III shows the experimental results of each model trained in three scenarios for 29 projects.

The last three rows show the results of three baselines, including All-Throw, All-Catch, and Random. The row of micro-average values indicates the overall performance of 29 projects, which are calculated based on the total numbers of TPs, TNs, FPs, and FNs of all samples.

As shown in Table III, we found that all the micro-average F1-scores in three scenarios greatly exceed three baselines. Although the micro-average accuracy values are close to the accuracy values in the baseline "All-Throw", the micro-average F1-scores are much higher than this baseline. This indicates that the classification model achieves high accuracy on all samples and high ability of classifying samples with "catch" labels.

Additionally, we noticed that the micro-average values of accuracy and F1-score reach 0.7935 and 0.5688 in the across-project recommendation scenario, respectively. They reach 0.8641 and 0.7134 in the across-project recommendation scenario; and 0.8743 and 0.7626 in the across-project+ recommendation scenario. The classification performance in intra-project+ scenario outperforms the original intra-project scenario. This also explains why EHAdvisor achieves the best recommendation performance in intra-project+ recommendation among all three scenarios.

C. Feature Contribution (RQ3)

RQ3 aims to evaluate how well the four types of features contribute to the classification and recommendation. The AutoML (Automated Machine Learning) tool AutoGluon [18] was used in the experiments as the binary classification model.

TABLE III
CLASSIFICATION RESULTS IN THREE SCENARIOS

	Across-project Classification					**Intra-project Classification**					**Intra-project+ Classification**				
Project	#Sample	Accuracy	Precision	Recall	F1	#Samples	Accuracy	Precision	Recall	F1	#Samples	Accuracy	Precision	Recall	F1
ActiveMQ	9982	0.8200	0.3851	0.7457	0.5079	999	0.8818	0.5188	0.8730	0.6508	999	0.8868	0.5320	0.8571	0.6565
C3P0	762	0.8202	0.5495	0.8260	0.6600	77	0.8831	0.7857	0.6470	0.7096	77	0.9090	0.8571	0.7058	0.7741
Collections	755	0.9072	0.2656	0.4250	0.3269	76	0.9078	0.3750	0.6000	0.4615	76	0.8552	0.2000	0.4000	0.2666
DBCP	415	0.7927	0.6515	0.6825	0.6666	42	0.7380	0.6428	0.6000	0.6206	42	0.8333	0.7222	0.8666	0.7878
Druid	4002	0.9382	0.7621	0.8571	0.8068	401	0.9625	0.8571	0.9230	0.8888	401	0.9551	0.8133	0.9384	0.8714
Dubbo	2061	0.8549	0.7044	0.8407	0.7665	207	0.8937	0.8085	0.7450	0.7755	207	0.8888	0.8181	0.7058	0.7578
FastJson	361	0.7950	0.7245	0.8120	0.7658	37	0.8918	0.8750	0.8750	0.8750	37	0.8378	0.8125	0.8125	0.8125
Flink	10863	0.7838	0.4104	0.6500	0.5031	1087	0.8776	0.6409	0.7230	0.6795	1087	0.8712	0.6190	0.7333	0.6713
Grizzly	2004	0.8388	0.6062	0.4531	0.5186	201	0.8208	0.5151	0.8947	0.6538	201	0.8557	0.6750	0.6279	0.6506
Gson	366	0.8715	0.6122	0.5172	0.5607	37	0.5945	0.4166	0.3846	0.4000	37	0.7567	0.6428	0.6923	0.6666
Guava	2196	0.7176	0.6164	0.4079	0.4909	220	0.9136	0.9538	0.7948	0.8671	220	0.9000	0.9242	0.7820	0.8472
HBase	11074	0.7944	0.4338	0.6350	0.5155	1108	0.8718	0.5812	0.8609	0.6939	1108	0.8772	0.5977	0.8342	0.6964
HikariCP	230	0.8434	0.5925	0.6956	0.6399	23	0.7826	1.0000	0.6153	0.7619	23	0.7391	0.8888	0.6153	0.7272
HttpClient	442	0.7850	0.6888	0.4806	0.5662	45	0.7555	0.6000	0.8000	0.6857	45	0.8666	0.8750	0.5833	0.7000
Jackson	968	0.8336	0.0370	0.5454	0.0693	97	0.9793	0.5000	1.0000	0.6666	97	0.9793	0.5000	1.000	0.6666
Jersey	1888	0.7981	0.6762	0.5019	0.5761	189	0.8042	0.5797	0.8333	0.6837	189	0.8359	0.7272	0.6274	0.6736
JPA	60	0.7833	0.8000	0.4210	0.5517	6	0.6666	0.5000	0.5000	0.5000	6	0.3333	0.0000	0.0000	0.0000
Log4J	1351	0.7283	0.7142	0.6589	0.6855	136	0.8161	0.7727	0.8360	0.8031	136	0.8308	0.7968	0.8360	0.8160
Logging	54	0.7222	0.7777	0.8750	0.8235	6	1.0000	1.0000	1.0000	1.0000	6	0.8333	1.0000	0.8333	0.9090
MyBatis	609	0.8390	0.6984	0.5945	0.6423	61	0.8032	0.5625	0.6428	0.6000	61	0.8524	0.7500	0.7058	0.7272
Netty	4879	0.6612	0.2837	0.6180	0.3889	488	0.8913	0.6666	0.7954	0.7253	488	0.9262	0.8023	0.7840	0.7931
RocketMQ	1308	0.7981	0.7962	0.7363	0.7651	131	0.8854	0.8360	0.9107	0.8717	131	0.8854	0.8596	0.8750	0.8672
Shiro	486	0.7386	0.5252	0.7591	0.6208	49	0.7755	0.6086	0.8750	0.7179	49	0.8367	0.5454	0.6666	0.6000
SLF4J	116	0.7931	0.7380	0.7045	0.7209	12	0.6666	0.6000	0.6000	0.6000	12	0.9160	0.0000	0.0000	0.0000
Storm	4524	0.8408	0.5478	0.6975	0.6137	453	0.8189	0.4728	0.8133	0.5980	453	0.8300	0.4916	0.7866	0.6051
Struts	1858	0.7868	0.6464	0.7000	0.6721	186	0.8118	0.7205	0.7538	0.7368	186	0.8494	0.7777	0.7241	0.7500
Tomcat	5925	0.7049	0.5366	0.6633	0.5932	593	0.8128	0.6902	0.7918	0.7375	593	0.8347	0.7756	0.6576	0.7117
XNIO	1137	0.8619	0.5000	0.6114	0.5501	114	0.8596	0.5172	0.8823	0.6521	114	0.8859	0.5217	0.8571	0.6486
ZooKeeper	1388	0.7860	0.6889	0.6329	0.6597	139	0.7913	0.6578	0.6097	0.6329	139	0.8201	0.6904	0.7073	0.6987
(micro avg.)	*72064*	*0.7935*	*0.5001*	*0.6593*	*0.5688*	*7220*	*0.8641*	*0.6439*	*0.7996*	*0.7134*	*7220*	*0.8743*	*0.6758*	*0.7626*	*0.7166*
All-Throw	72064	0.7934	0.0000	0.0000	0.0000	7220	0.7887	0.0000	0.0000	0.0000	7220	0.7887	0.0000	0.0000	0.0000
All-Catch	72064	0.2065	0.2065	1.0000	0.3423	7220	0.2112	0.2112	1.0000	0.3487	7220	0.2112	0.2112	1.0000	0.3487
Random	72064	0.5000	0.2065	0.5000	0.2922	7220	0.5000	0.2112	0.5000	0.2969	7220	0.5000	0.2112	0.5000	0.2969

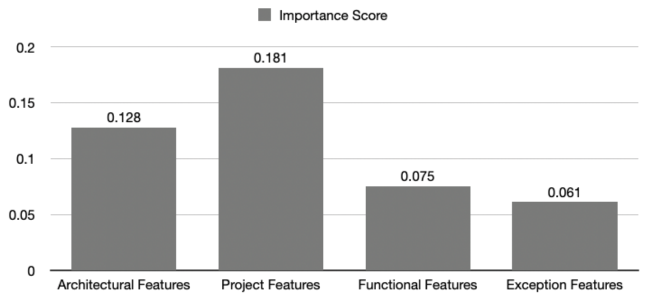

Fig. 4. Importance Scores of Four Types of Features.

It provides the functionality of calculating the importance of each individual features in the model.

In AutoGluon, the importance score of a feature is calculated as follows: AutoGluon first makes a perturbed copy of the dataset, where values of this feature are randomly shuffled across rows. Then the performance drop of the model prediction result is considered as the importance score of the feature [18]. For example, an importance score of 0.01 of a feature indicates that the predictive performance dropped by 0.01 on the perturbed dataset. The higher the score a feature has, the more important the feature is to the model performance. A feature with a negative importance score means that the feature is likely harmful to the final model.

We evaluated the importance scores with AutoGluon on a dataset with all samples to find the overall importance of four features. The result is shown in Fig. 4. We found that all the four types of features contribute positively. Specifically, the importance scores of two global features, namely the project features and the architectural features, are evidently higher than the other two features. This indicates that global features play much more important roles in exception handling prediction and recommendation.

VI. THREATS TO VALIDITY

We present three threats to validity of our work as follows.

The first threat to validity is that we only evaluate EHAdvisor on Java projects. Yet, EHAdvisor is designed as a language-independent approach, and the features, classification model, and recommendation approach could be adapted to the projects in other languages.

The second threat to validity comes from the quality and diversity of the dataset. Although all 29 open-source projects are popular projects, their code quality are different: there may be inappropriate exception handling operations in the ground truth. Meanwhile, we do not include Android and desktop applications in our dataset. This is because exception handling in these projects is very different from Java libraries and frameworks [22]. Besides, some projects may use event-driven frameworks, task scheduling frameworks, or other frameworks, which makes their call graphs not easy to construct. Therefore, these projects are also not included in the dataset. We will add more types of projects into the project list and improve sample quality through automatically or manually cleaning the samples in future work.

The third threat to validity comes from the various samples distribution of projects. Several projects contain a small sample set. This makes the result unstable. In addition, the distribution of positive and negative samples in different projects varies widely. This may result in a non-linear correlation of metrics between classification and recommendation. For example, the project Jackson contains more "throw" samples than "catch" samples. Though it has a relatively low F1-score of classification, the recommendation success rate on it is still high.

VII. Related Work

We list the related work in following two categories:

A. Empirical Studies on Exception Handling

Many empirical studies have revealed the problems in exception handling. Empirical studies [6], [23]–[27] analyzed the exception handling strategies among different programming languages. User surveys [28] studied the developer behaviors of exception handling. Their work shows that the inherent complexity in exception handling is one of reasons that many developers end up with oversimplifying or even neglecting exception handling. Hugo et al. [9] identified 48 exception handling guidelines related to seven different categories through interviews and surveys.

Existing studies [5], [6], [24], [29]–[32] investigated the impact of the exception handling related code over the system robustness. They found exception handling bugs are prevalent in cloud-based systems [5] and popular Java libraries [6]. Among exception handling bugs, 85% to 90% faults have related to global exception flows [7], [8], [33].

Exception flow analysis is a classical program analysis technique to study the exception flows and help understand the global design of exception flows in software architectures [34]–[38]. Chang and Choi [39] provided a comprehensive review on exception analysis, including exception usage analysis and exception flow analysis.

B. Code Mining for Exception Handling

Thummalapenta and Xie [40] presented an approach based on closed frequent sequential pattern mining to find exception-handling rules. The rules can partially represent specifications for exception handling. Some studies also propose methods or tools to detect inaccurate exception handling code based on pre-defined exception handling policies or anti-patterns and provide some recommendations for fix the bugs [5], [41]–[44]. Barbosa and Garcia [2] proposed a heuristic strategy that recommends repairs to global exception policy violations.

However, these exception handling rules are often project-specific and hence are not easy to be reused across projects. The quality of the rules heavily relies on the developer's experience as well. In contrast, our EHAdvisor automatically learns general exception handling patterns across multiple projects to provide the broad applicability.

Many studies also recommend code for exception handling. Rahman and Roy [45] and Barbosa et al. [46] proposed methods to search for similar exception handling snippets in open source code repositories based on code context similarity. Li et al. [47] presented a automatic method that recommends exception handling strategies based on program context, including method names and descriptions. Nguyen et al. [48] presented a recommendation tool for handling exceptions based on fuzzy logic. Their work can predict whether a runtime exception would occur in a given code snippet and recommend code to handle that exception. However, these recommendation approaches cannot recommend the exception handling locations in call chains from a global view.

VIII. Conclusions

Developers are often confused about where to handle the exceptions in the call chains of methods since exception handling requires developers to understand the global exception flow of the project. In this paper, we propose an automatic approach EHAdvisor, to recommend the exception handling locations. EHAdvisor conduct recommendation based on four types of features, namely, architectural features, project features, functional feature, and exception features from a global perspective. EHAdvisor first trains a binary classification model based on the samples from high-quality code repositories and then recommends the most appropriate method to handle exceptions based on the ranking of the exception catch probability of each method in the chain that is predicted by the classification model prediction. Experimental results show that EHAdvisor can achieve an average Top-1 accuracy of 70.83% for the across-project location recommendation and an average Top-1 accuracy of 86.21% for the intra-project recommendation. The two global features, namely architectural features and project features, are shown to contribute the most to the recommendation.

In future work, we plan to integrate EHAdvisor into an IDE as a plugin to help developers cope with exceptions. We also plan to introduce the exception handling policies and anti-patterns as knowledge into EHAdvisor to further improve its performance. In addition, we are also interested in exploring methods to make the implicit exception handling knowledge learned from the code repository explainable.

Data Availability

All the data and code for this work are available online at https://github.com/jiaxy/EHAdvisor/.

Acknowledgments

We sincerely appreciate reviewers' valuable advice for this work and thank Prof. Xiaoyuan Xie for her support. This work is supported by the National Key R&D Program of China (No. 2018YFB1003901), the National Natural Science Foundation of China (No. 61872273), the Advance Research Projects of Civil Aerospace Technology, Intelligent Distribution Technology of Domestic Satellite Information (No. B0301) and Communications, Navigation and Remote Sensing Integrated Applications and Multi-source Spatial Data Fusion Technology.

References

[1] B. Cabral and P. Marques, "Exception handling: A field study in java and .net," in *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, ser. Lecture Notes in Computer Science, E. Ernst, Ed., vol. 4609. Springer, 2007, pp. 151–175. [Online]. Available: https://doi.org/10.1007/978-3-540-73589-2_8

[2] E. A. Barbosa and A. Garcia, "Global-aware recommendations for repairing violations in exception handling," *IEEE Trans. Software Eng.*, vol. 44, no. 9, pp. 855–873, 2018. [Online]. Available: https://doi.org/10.1109/TSE.2017.2716925

[3] H. B. Shah, C. Görg, and M. J. Harrold, "Understanding exception handling: Viewpoints of novices and experts," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 150–161, 2010.

[4] R. Coelho, L. Almeida, G. Gousios, A. v. Deursen, and C. Treude, "Exception handling bug hazards in android," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1264–1304, Jun 2017. [Online]. Available: https://doi.org/10.1007/s10664-016-9443-7

[5] H. Chen, W. Dou, Y. Jiang, and F. Qin, "Understanding exception-related bugs in large-scale cloud systems," *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, pp. 339–351, 2019.

[6] D. Sena, R. Coelho, U. Kulesza, and R. Bonifácio, "Understanding the exception handling strategies of java libraries: an empirical study," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, M. Kim, R. Robbes, and C. Bird, Eds. ACM, 2016, pp. 212–222. [Online]. Available: https://doi.org/10.1145/2901739.2901757

[7] N. Cacho, T. César, T. Filipe, E. Soares, A. Cassio, R. Souza, I. García, E. A. Barbosa, and A. Garcia, "Trading robustness for maintainability: an empirical study of evolving c# programs," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. ACM, 2014, pp. 584–595. [Online]. Available: https://doi.org/10.1145/2568225.2568308

[8] N. Cacho, E. A. Barbosa, J. Araujo, F. Pranto, A. Garcia, T. Cesar, E. Soares, A. Cassio, T. Filipe, and I. Garcia, "How does exception handling behavior evolve? An exploratory study in Java and C# applications," *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014*, no. September, pp. 31–40, 2014.

[9] H. Melo, R. Coelho, and C. Treude, "Unveiling Exception Handling Guidelines Adopted by Java Developers," *SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering*, pp. 128–139, 2019.

[10] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, ser. JMLR Workshop and Conference Proceedings, vol. 32. JMLR.org, 2014, pp. 1188–1196. [Online]. Available: http://proceedings.mlr.press/v32/le14.html

[11] Google, "Machine learning crash course," https://developers.google.com/machine-learning/crash-course/representation/feature-engineering.

[12] Wikipedia, "Naming convention (programming)," https://en.wikipedia.org/wiki/Naming_convention_(programming).

[13] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE transactions on systems, man, and cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.

[14] V. Svetnik, A. Liaw, C. Tong, J. C. Culberson, R. P. Sheridan, and B. P. Feuston, "Random forest: a classification and regression tool for compound classification and qsar modeling," *Journal of chemical information and computer sciences*, vol. 43, no. 6, pp. 1947–1958, 2003.

[15] W. S. Noble, "What is a support vector machine?" *Nature biotechnology*, vol. 24, no. 12, pp. 1565–1567, 2006.

[16] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE transactions on information theory*, vol. 13, no. 1, pp. 21–27, 1967.

[17] H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin, "Exploring strategies for training deep neural networks," *J. Mach. Learn. Res.*, vol. 10, pp. 1–40, 2009. [Online]. Available: https://dl.acm.org/citation.cfm?id=1577070

[18] N. Erickson, J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li, and A. Smola, "Autogluon-tabular: Robust and accurate automl for structured data," *arXiv preprint arXiv:2003.06505*, 2020.

[19] Eclipse, "Eclipse jdt," https://www.eclipse.org/jdt/.

[20] L. Sui, J. Dietrich, A. Tahir, and G. Fourtounis, "On the recall of static call graph construction in practice," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 1049–1060. [Online]. Available: https://doi.org/10.1145/3377811.3380441

[21] C. Liu, X. Xia, D. Lo, C. Gao, X. Yang, and J. Grundy, "Opportunities and challenges in code search tools," *arXiv preprint arXiv:2011.02297*, 2020.

[22] J. Oliveira, D. Borges, T. Silva, N. Cacho, and F. Castor, "Do android developers neglect error handling? a maintenance-centric study on the relationship between android abstractions and uncaught exceptions," *J. Syst. Softw.*, vol. 136, pp. 1–18, 2018. [Online]. Available: https://doi.org/10.1016/j.jss.2017.10.032

[23] D. Reimer and H. Srinivasan, "Analyzing exception usage in large java applications," in *Proceedings of ECOOP 2003 Workshop on Exception Handling in Object-Oriented Systems*, 2003, pp. 10–18.

[24] C. Marinescu, "Should we beware the exceptions? an empirical study on the eclipse project," in *15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2013, Timisoara, Romania, September 23-26, 2013*, 2013, pp. 250–257.

[25] M. Monperrus, M. Germain De Montauzan, B. Cornu, R. Marvie, and R. Rouvoy, "Challenging Analytical Knowledge On Exception-Handling: An Empirical Study of 32 Java Software Packages," Laboratoire d'Informatique Fondamentale de Lille, Technical Report hal-01093908, 2014. [Online]. Available: https://hal.inria.fr/hal-01093908

[26] S. Nakshatri, M. Hegde, and S. Thandra, "Analysis of exception handling patterns in java projects: an empirical study," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 500–503.

[27] M. Asaduzzaman, M. Ahasanuzzaman, C. K. Roy, and K. A. Schneider, "How developers use exception handling in java?" in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 516–519. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903500

[28] H. Shah, C. Görg, and M. J. Harrold, "Understanding exception handling: Viewpoints of novices and experts," *IEEE Trans. Software Eng.*, vol. 36, no. 2, pp. 150–161, 2010. [Online]. Available: https://doi.org/10.1109/TSE.2010.7

[29] P. Sawadpong, E. B. Allen, and B. J. Williams, "Exception handling defects: An empirical study," in *14th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2012, Omaha, NE, USA, October 25-27, 2012*, 2012, pp. 90–97.

[30] N. Cacho, T. César, T. Filipe, E. Soares, A. Cassio, R. Souza, I. García, E. A. Barbosa, and A. Garcia, "Trading robustness for maintainability: an empirical study of evolving c# programs," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, 2014, pp. 584–595.

[31] F. Ebert, F. Castor, and A. Serebrenik, "An exploratory study on exception handling bugs in java programs," *Journal of Systems and Software*, vol. 106, pp. 82–101, 2015. [Online]. Available: https://doi.org/10.1016/j.jss.2015.04.066

[32] H. Osman, A. Chis, C. Corrodi, M. Ghafari, and O. Nierstrasz, "Exception evolution in long-lived java systems," in *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 302–311.

[33] Y. Gu, J. Xuan, H. Zhang, L. Zhang, Q. Fan, X. Xie, and T. Qian, "Does the fault reside in a stack trace? assisting crash localization by predicting crashing fault residence," *J. Syst. Softw.*, vol. 148, pp. 88–104, 2019. [Online]. Available: https://doi.org/10.1016/j.jss.2018.11.004

[34] B. Chang, J. Jo, and S. H. Her, "Visualization of exception propagation for java using static analysis," in *2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002), 1 October 2002, Montreal, Canada*, 2002, p. 173. [Online]. Available: https://doi.org/10.1109/SCAM.2002.1134117

[35] M. P. Robillard and G. C. Murphy, "Static analysis to support the evolution of exception structure in object-oriented systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 12, no. 2, pp. 191–221, 2003. [Online]. Available: http://doi.acm.org/10.1145/941566.941569

[36] C. Fu and B. G. Ryder, "Exception-chain analysis: Revealing exception handling architecture in java server applications," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, 2007, pp. 230–239. [Online]. Available: https://doi.org/10.1109/ICSE.2007.35

[37] H. Shah, C. Görg, and M. J. Harrold, "Visualization of exception handling constructs to support program understanding," in *Proceedings of the ACM 2008 Symposium on Software Visualization, Ammersee, Germany, September 16-17, 2008*, 2008, pp. 19–28. [Online]. Available: http://doi.acm.org/10.1145/1409720.1409724

[38] P. Ma, H. Cheng, J. Zhang, and J. Xuan, "Can this fault be detected: A study on fault detection via automated test generation," *J. Syst. Softw.*, vol. 170, p. 110769, 2020. [Online]. Available: https://doi.org/10.1016/j.jss.2020.110769

[39] B. Chang and K. Choi, "A review on exception analysis," *Information & Software Technology*, vol. 77, pp. 1–16, 2016. [Online]. Available: https://doi.org/10.1016/j.infsof.2016.05.003

[40] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, 2009, pp. 496–506.

[41] F. Kistner, M. B. Kery, M. Puskas, S. Moore, and B. A. Myers, "Moonstone: Support for understanding and writing exception handling code," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2017, Raleigh, NC, USA, October 11-14, 2017*, A. Z. Henley, P. Rogers, and A. Sarma, Eds. IEEE Computer Society, 2017, pp. 63–71. [Online]. Available: https://doi.org/10.1109/VLHCC.2017.8103451

[42] T. Montenegro, H. Melo, R. Coelho, and E. Barbosa, "Improving developers awareness of the exception handling policy," in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, R. Oliveto, M. D. Penta, and D. C. Shepherd, Eds. IEEE Computer Society, 2018, pp. 413–422. [Online]. Available: https://doi.org/10.1109/SANER.2018.8330228

[43] T. T. Nguyen, P. M. Vu, and T. T. Nguyen, "Recommendation of exception handling code in mobile app development," *CoRR*, vol. abs/1908.06567, 2019. [Online]. Available: http://arxiv.org/abs/1908.06567

[44] J. L. M. Filho, L. S. Rocha, R. M. C. Andrade, and R. Britto, "Preventing erosion in exception handling design using static-architecture conformance checking," in *Software Architecture - 11th European Conference, ECSA 2017, Canterbury, UK, September 11-15, 2017, Proceedings*, ser. Lecture Notes in Computer Science, A. Lopes and R. de Lemos, Eds., vol. 10475. Springer, 2017, pp. 67–83. [Online]. Available: https://doi.org/10.1007/978-3-319-65831-5_5

[45] M. M. Rahman and C. K. Roy, "On the use of context in recommending exception handling code examples," in *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*, 2014, pp. 285–294. [Online]. Available: https://doi.org/10.1109/SCAM.2014.15

[46] E. A. Barbosa, A. Garcia, and M. Mezini, "Heuristic strategies for recommendation of exception handling code," in *26th Brazilian Symposium on Software Engineering, SBES 2012, Natal, Brazil, September 23-28, 2012*, 2012, pp. 171–180. [Online]. Available: https://doi.org/10.1109/SBES.2012.22

[47] Y. Li, S. Ying, X. Jia, Y. Xu, L. Zhao, G. Cheng, B. Wang, and J. Xuan, "Eh-recommender: Recommending exception handling strategies based on program context," in *23rd International Conference on Engineering of Complex Computer Systems, ICECCS 2018, Melbourne, Australia, December 12-14, 2018*. IEEE Computer Society, 2018, pp. 104–114. [Online]. Available: https://doi.org/10.1109/ICECCS2018.2018.00019

[48] T. Nguyen, P. Vu, and T. Nguyen, "Code recommendation for exception handling," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 1027–1038. [Online]. Available: https://doi.org/10.1145/3368089.3409690