D-REX: Static Detection of Relevant Runtime Exceptions with Location Aware Transformer

Farima Farmahinifarahani*§, Yadong Lu*§, Vaibhav Saini†, Pierre Baldi* and Cristina Lopes*
*University of California, Irvine, USA
†Microsoft, USA
*{farimaf, yadongl1, pfbaldi, lopes}@uci.edu, †vaibhav.saini@microsoft.com

***Abstract*—Runtime exceptions are inevitable parts of software systems. While developers often write exception handling code to avoid the severe outcomes of these exceptions, such code is most effective if accompanied by accurate runtime exception types. Predicting the runtime exceptions that may occur in a program, however, is difficult as the situations that lead to these exceptions are complex. We propose D-REX (Deep Runtime EXception detector), as an approach for predicting runtime exceptions of Java methods based on the static properties of code.**

The core of D-REX is a machine learning model that leverages the representation learning ability of neural networks to infer a set of signals from code to predict the related runtime exception types. This model, which we call Location Aware Transformer, adapts a state-of-the-art language model, Transformer, to provide accurate predictions for the exception types, as well as interpretable recommendations for the exception prone elements of code. We curate a benchmark dataset of 200,000 Java projects from GitHub to train and evaluate D-REX. Experiments demonstrate that D-REX predicts runtime exception types with 81% of Top 1 accuracy, outperforming multiple non-Transformer baselines by a margin of at least 12%. Furthermore, it can predict the exception prone elements of code with 75% Top 1 precision.

***Index Terms*—exception prediction, runtime exceptions, software code modeling**

I. Introduction

Exception handling is a crucial part of developing robust and reliable software systems. Exceptions can be defined as events (or errors) that happen during the runtime of a software system and result in the deviation of the execution of the program from its normal and expected behavior [1], [2], which subsequently can result in serious issues such as crashing the system [3]. In order to control the execution of the program in the event of exceptions, developers develop code specific to the occurrence of each exception, also referred to as *exception handling*. For instance, in Java, exception handling can be done by implementing *try-catch* blocks.

Most languages that support exceptions, treat all exceptions the same way. Java, however, has two categories of exceptions: checked and unchecked. The conceptual difference between the two is subtle: checked exceptions are meant to capture errors that can be recovered from, while unchecked exceptions capture unrecoverable errors. Examples of the former include problems with the file system such as file not found; examples of the latter include problems arising from fatally wrong inputs, such as division by a zero value and the dereferencing of null objects [4]. Operationally, Java treats checked and unchecked exceptions differently: the declaration and handling of checked exceptions is enforced by the Java compiler, while unchecked exceptions are left for the runtime to handle.

Predicting the types of exceptions that will happen during the runtime of a program, however, can be difficult and developers may not always come up with an inclusive set. Hence, they may ignore handling such exceptions, or simply resort to handling the most generic type of exception. Studies of the Java ecosystem have shown that the Exception class, which is at the top of the exception hierarchy, has a high frequency of appearance in try-catch blocks [5]. Handling the Exception class is considered a bad practice as it hides the real causes of problems [6], [7], [8]. Previous studies reveal that bad exception handling practices are prevalent [9], and that unchecked exceptions have a heavy toll with respect to bugs [9], [3]. Therefore, it is crucial to handle, or avoid, unchecked exceptions (which are also referred to as runtime exceptions).

In order to prevent such exceptions from happening, developers typically investigate their code, along with the documentation of the APIs it uses. However, rather than solely relying on the developers' knowledge of which exceptions to handle, it will be useful to develop techniques that make recommendations based on the concrete code at hand. A state-of-the-art tool in this context is FuzzyCatch [2], [10], developed for the Android ecosystem. In the absence of the runtime information before executing programs, FuzzyCatch makes predictions based on the co-occurrences of API method calls and runtime exceptions in the Android apps. However, by only considering pairs of method calls and exception types, it fails to account for the complex interplay between multiple code elements (such as method calls, numerical operations, or structural code constructs) and the exception types; this is important because often there are multiple method calls and the use of certain operations and classes in a certain order that result in a runtime exception. Moreover, by only relying on API method calls, methods lacking API method calls (that are possibly exception prone) are totally missed.

In order to be able to estimate the probability of various runtime exception types based on the interplay of different code elements, we need an approach that can infer these relationships from code. A simple statistical model or human developed heuristics may not work well here since the existence of various

§Both authors contributed equally to the paper

code elements can lead to various scenarios of exceptions occurrence, making it infeasible to derive an inclusive set of rules. To address this challenge, we present D-REX (Deep Runtime Exception detector), a deep learning based approach that captures the correlations between certain code elements and a set of Java runtime exception types to predict the possible runtime exceptions. D-REX is also able to make predictions for *the exception prone code elements*. 'Code elements', here, refer to various code constructs, such as method calls, type castings, or numerical operations that are captured by D-REX in the form of special tokens (described in Section IV-A). By being informed about the part of the code that is causing an exception, developers can make more informed decisions on how to handle the exception.

The workflow of D-REX is divided into two stages: 1) building a special token sequence for each method which we call *Action-Context Token Sequence* (ACTS), containing tokens representing different code elements, and 2) joint training of a feature extractor model which produces a semantically meaningful representation of the input ACTS, and two neural network classifiers which take the learned representation as input and predict the relevant runtime exception types, as well as the exception prone ACTS tokens. For the second stage, we propose a deep neural network model, *Location Aware Transformer* (LA-Transformer), which adapts Transformer [11], a state-of-the-art language model, for predicting the possible exception types and exception prone tokens. LA-Transformer is location aware in the sense that given a method's ACTS tokens, it leverages the information of which tokens are located inside the try block in its training phase. This extra piece of information helps it in predicting the exception prone ACTS tokens and also, making more accurate exception type predictions. While during training, we utilize the existing information about the tokens inside the try block, during inference, such information is not needed and the model predicts the tokens which it finds to be exception prone.

We chose Transformer as the feature extractor model of D-REX since its self-attention mechanism allows every element of the input sequence to directly communicate with all other elements in each layer in constant time and space complexity [11], alleviating the issue of forgetting in modeling longer sequence data compared to the recurrent neural networks [12], [13]. We found this particularly useful in learning meaningful representations from code as the lengths of methods can vary both within and across the projects. We demonstrate its effectiveness in our experiments by showing that D-REX outperforms the Bidirectional LSTM [13] in exception type prediction by a considerable margin.

Overall, our contributions can be summarized as follows.

1) We present D-REX, a novel approach for the recommendation of unchecked (runtime) exceptions in Java, which outperforms multiple baselines.
2) We present an application of Transformer model for predicting runtime exceptions. To leverage the information about the parts of the code located inside the try blocks during training, our main contribution, in this respect, is Location Aware Transformer that produces more accurate results and identifies exception prone tokens.
3) We curate a dataset of 200K Java projects with more than ≈442K try-catch blocks handling runtime exceptions available at mondego.ics.uci.edu/projects/d-rex/.

The remainder of this paper is structured as follows. Section II builds the motivation for our task by providing concrete examples and explains our ultimate goal in detail. Section III describes the dataset that we curated for training and evaluation of D-REX, and Section IV discusses our proposed approach by explaining the architecture of D-REX, capturing ACTS and the details of LA-Transformer. Section V presents the results of the experiments conducted to evaluate D-REX's effectiveness in predicting exception types and exception prone tokens on two separate datasets. Finally, we present the related work in Section VI followed by a discussion on threats to validity in Section VII, and conclusions and future work in Section VIII.

II. Motivation and Goal

Runtime exceptions frequently occur in Java programs [9]; a study [3] on 246 Android exception related bugs found the majority of them to be runtime ones. Another study on 656 Java libraries found handling of API runtime exceptions to be more frequent than the checked ones [14]. Runtime exceptions are prevalent and can cause serious issues at the program runtime (e.g., complete crash of program) if not properly tackled or handled. For example, *NullPointerException*, which frequently happens in Java programs, serves as a major cause of crashes in software systems [15] and is typically not caught during testing, making it likely to propagate to the program runtime [16]. These findings demonstrate the importance of addressing runtime exceptions before they cause major issues.

In its simplest form, the problem of *runtime* or *unchecked* exception handling is seen in the *howManyTimes()* method at lines 2 to 8 of Listing 1. In this method, if the second parameter has a value of zero, a division by zero happens, leading to an *ArithmeticException*. This may not catch the developer's attention and they may miss to handle this exception. A possible fix to this issue is shown in the method starting from line 10 and ending at line 16 of Listing 1. In this fix, the developer deals with the *ArithmeticException* by implementing a pre-condition on the argument in their if statement.

Another example of a scenario susceptible to runtime exceptions is depicted in Listing 2 from line 2 to 10. At runtime, this method will throw a *NumberFormatException* on parsing the string values to integer values at lines 4 and 5 if a value in the input array does not represent an integer. Even if all the values in the input array are valid representations of integers, and the parsing is safely done, this method is yet susceptible to *ArrayIndexOutOfBoundsException*. The *for* loop traverses the array to its last element while inside the for loop, accessing the $i + 1_{th}$ index of the array can result in *ArrayIndexOutOfBoundsException* if $i == params.length - 1$. The method starting from line 12 of Listing 2 tries to tackle both problems. *NumberFormatException* is being handled by a try/catch block to alert the user about the proper input,

```
1  // Before fix
2  private int howManyTimes(int a , int b){
3      int result =0;
4      if (a > b) {
5          result = a / b;
6      }
7      return result ;
8  }
9  // After fix
10 private int howManyTimes(int a , int b){
11     int result =0;
12     if (b!=0 && a > b) {
13         result = a / b;
14     }
15     return result ;
16 }
```

Listing 1: Example 1: Method prone to RuntimeException

```
1  // Before fix
2  public static String [] findMax(String [] params){
3    for (int i = 0; i < params.length ; i++) {
4      if (Integer . parseInt (params[i ])>Integer . parseInt (params[i + 1])) {
5        params[i + 1] = params[i ];
6      }
7    }
8    return params;
9  }
10 // After fix
11 public static String [] findMax(String [] params){
12   try {
13     for (int i = 0; i < params.length -1; i++) {
14       if( Integer . parseInt (params[i ])>Integer . parseInt (params[i + 1])){
15         params[i + 1] = params[i ];
16       }
17     }
18   }
19   catch (NumberFormatException e){
20     System.out. println ("Input array may only contain integer values");
21   }
22   return params;
23 }
```

Listing 2: Example 2: Method prone to RuntimeException

and *ArrayIndexOutOfBoundsException* has been controlled by changing the upper bound of the loop counter from $params.length$ to $params.length - 1$.

As the two examples show, knowledge about the relevance of runtime exceptions can help the developer take the necessary actions. These actions need not necessarily be implementing an exception handler; valid actions include the addition of pre-conditions, changing the parameters types, and even rearchitecting the code. A trivial way of reminding developers about possible runtime problems would be to always remind them of all runtime exception types. However, that would have a very high rate of false positives, which would make developers ignore the reminders. Another simple solution can be to only remind about the *RuntimeException* class (the super class of all runtime exceptions in Java) which can result in ignoring important runtime exception types that may happen in different situations and result in unexpected program behaviour.

Our goal is to design and implement an approach that can predict *relevant* runtime exception types and exception prone code elements for each method. Unlike previous approaches [2], [10], [1], [17], [18], [19], our work does not focus on recommending exception handling code. Rather, we aim to inform the developer about the possible incidence of such exceptions and possible causes, but delegate the needed action to the developer, as wrapping runtime exceptions in a try/catch block is not always the best solution; sometimes fixing the code to prevent the exception from happening in the first place can be a better approach (as observed in the above examples).

In the absence of runtime information and user input prior to executing the program, we make use of the static signals from code to find clues pertaining to the relevance of certain runtime exceptions. For example, in Listing 1, the divide operator serves as a clue about the *ArithmeticException*. This is an example where a runtime exception is possible to happen while there is no API method call present in the code related to it. In Listing 2, the invocation of *parseInt()* on the method parameter value gives a clue about *NumberFormatException* and accessing the array indexes by doing an arithmetic operation $(i + 1)$ inside a for loop gives a clue about *ArrayIndexOutOfBoundsException*. This demonstrates how the complex interplay of various code elements can contribute to a runtime exception. Inferring the relationships between the code elements and predicting relevant exception types accordingly using heuristics or statistical methods may not work well here, and a machine learning based method such as D-REX can help us by automatically learning the relevant patterns based on a large dataset of examples.

In addition to recommending relevant exception types, by identifying exception-prone elements of code, D-REX can help the developer find the issues in their program or implement the suitable try/catch block, as most of the time the developer is not aware which part of their code is possibly going to throw an exception. For example, the Top 1 exception prediction of D-REX for the example in Listing 1 is *ArithmeticException* and the *divide* operator on line 5 is predicted as the most probable cause for it. For Listing 2, D-REX predicts both *NumberFormatException* and *ArrayIndexOutOfBoundsException* as its top 2 predicted exception types, and the invocations of *parseInt()* and accessing the array inside the loop as the exception prone parts of code. An interesting observation here is that D-REX predicted *NullPointerException* as its third exception type for this method; a closer look shows that this exception is possible to be thrown if the input array (*params*) has a Null value.

D-REX operates on Java methods and its core is a deep learning model. This model predicts the relevant exception types and exception-prone elements of code by learning from a dataset of over 350k developer-implemented try blocks handling runtime exceptions. For training, we use the information from the try/catch blocks; for evaluation, the try/catch blocks are removed from the methods to provide the model with the raw method source code that it receives from a developer.

III. Dataset

We used the Java dataset used in the GitHub study by Lopes et al. [20] to prepare our dataset. The benefit of this dataset is that it includes the Java projects present in GitHub until the time of paper's publication with forked projects excluded and a mapping of cloning among the projects provided. This is particularly helpful in a machine learning task like ours as we could use it to remove the repetitions among projects to reduce the duplications in training and testing datasets and train a

more generalized model. Their whole Java dataset contains 1,481,468 projects and 72,880,615 files. From these projects, we only considered those that have more than 10 files (to narrow our focus to bigger and meaningful projects), leaving us with 670,429 projects. We then filtered out projects that are 50% or more clones of other projects; giving us 544,513 projects. We then drew a 200,000 random sample of these projects, which we name *200k* dataset. Using the file level mapping of clones [20], we also filtered out the files that are total duplicates of each other; i.e., the Type-1 clones [21].

Prior to parsing the projects, we collected the list of Java SDK runtime exceptions from its documentation[1]. We collected all subclasses of the *RuntimeException* (including this class itself), a set of 169 exceptions. To curate a labeled dataset of methods that may have runtime exceptions, we looked for methods that have a *try-catch* block where one of the 169 collected runtime exceptions is caught in their catch statement.

We modified JavaParser [22] to parse the projects and fetch methods and the try-catch blocks within them and the tokens needed for the Action-Context Token Sequence. In this step, we found 577,067 examples of try-catch blocks catching a Java runtime exception. From these, we removed the ones that were catching the *RuntimeException* itself since it is the superclass of all of the target exceptions and including it as a label can confuse the model. This left us with 470,376 examples. By analyzing the exceptions' frequencies, we found some rare exception types: for example, 35 exceptions appear less than 10 times. We decided to exclude the examples with very rare exceptions (the ones appearing <100 times) to focus on the most frequent exception types. We also removed methods with inner classes in their bodies since their structure is different from conventional methods and can introduce noise to the dataset. After these two steps, we were left with 442,446 samples and 52 exception types. Figure 1 shows the frequencies for the top 10 frequent runtime exceptions. The three most frequent exceptions are the *NumberFormatException* (142,800 times), the *IllegalArgumentException* (89,193 times) and the *NullPointerException* (46,575 times). For the rest of the exceptions not shown here, twelve appeared between 10,000 and 1,000 times, and the rest appeared less than 1,000 times.

We split the dataset into train, validation and test sets by dedicating 80% of samples to the train set, 10% to the validation set and 10% to the test set. As a result, the train set has 353,956 samples and test and validation each have 44,245 samples. We perform the training on the train set, determine all the hyperparameters and early stopping steps using the validation set, and benchmark the accuracy of D-REX and other baselines using the hold out test set.

IV. Proposed Approach

Figure 2 shows an overview of the D-REX prediction pipeline. The prediction starts by receiving a Java method and parsing it to retrieve a token sequence which we call *Action-Context Token Sequence* (ACTS). The goal here is to capture the key elements of the code that represent the main actions carried out in the method while decreasing the language vocabulary. This sequence is then fed to a deep learning model, called LA-Transformer, that applies transformations on its input to build a high-level semantic representation, and predicts two outputs: (i) top K runtime exception types possible to happen ranked by their occurrence probabilities, and (ii) top N exception prone tokens, which are tokens from the ACTS that are most likely responsible for the predicted exceptions. The two main components of this pipeline, *Action-Context Token Sequence* and *LA-Transformer*, are discussed in this section.

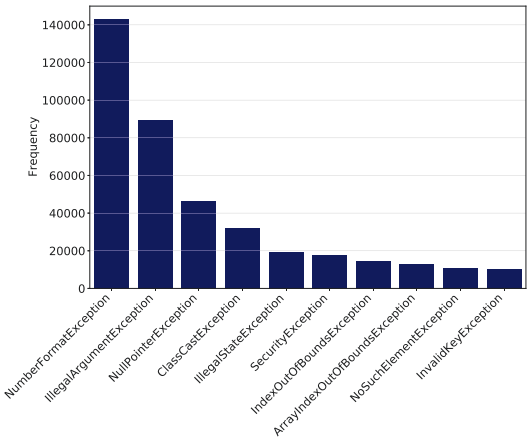


Fig. 1: Distribution of the 10 Most Frequent Exceptions

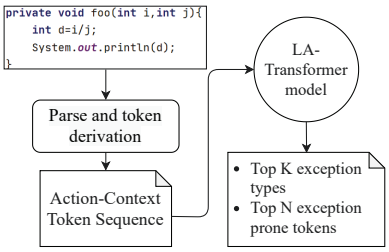


Fig. 2: Overview of the D-REX Prediction Pipeline

A. Action-Context Token Sequence

In order to provide the input method to the prediction pipeline, we derive a token sequence from it providing the key information helpful in predicting the runtime exceptions. Previous research has explored the correlations of API method calls with the occurrence of runtime exceptions [2], [10], however, method calls alone may not be sufficient to estimate the probability of runtime exceptions. For instance, in the *Before fix* situation of the example depicted in Listing 2, the occurrence of *ArrayIndexOutOfBoundsException* is not recognizable if we only look at the method calls; it is accessing the indexes of the input array that causes this exception. Moreover, a single access to an array may not introduce a high probability of exception occurrence compared to when it is accessed inside a loop. Therefore, one may find it useful to feed all method tokens to the deep learning model and let the model perform

[1]https://docs.oracle.com/javase/7/docs/api/java/lang/RuntimeException.html

its inference and make the predictions. However, source code vocabulary has been shown to be unlimited as new identifier names get introduced constantly [23], [24]; such increase in vocabulary can make it difficult for the deep learning model to infer meaningful representations from the input and hampers the model convergence. Therefore, we need to derive a set of tokens from the methods that can decrease the vocabulary by removing unnecessary elements while capturing the key elements of code that can contribute to the occurrence of runtime exceptions. As the examples in Section II showed, it is often an interplay of different code elements that ultimately result in a runtime exception; in other words, it is a specific action (such as a method call or a mathematical operation) in a specific context (for example a for loop or a special condition in a nested if statement) that causes a specific runtime exception. The names of the variables, for example, may not introduce relevant and useful information here.

To address these needs, we build a special token sequence for each method, called *Action-Context Token Sequence* (ACTS), capturing two important aspects from each method: *actions* carried out in it and the *context* in which these actions are performed. Hence, we have two categories of tokens: tokens capturing actions and tokens capturing contextual information. These tokens may have identical correspondence with the tokens in code (e.g., method calls), or may be derived from code and presented by a literal value (e.g., 'Cast' to show type casting). Details of ACTS are as follows:

1) Tokens capturing method's actions: One set of captured tokens are the method calls invoked in the method body. As the example in Listing 2 showed, method calls (e.g., *parseInt()*) can serve as causes for throwing a runtime exception (*NumberFormatException*). FuzzyCatch [2], [10] heavily relies on API method calls and Barbosa et. al [1] use method calls to recommend exception handling code. Saini et al. [25] refer to method calls as *'Action Tokens'* and introduce two other action tokens as well: *ArrayAccess* for accessing the index of an array and *ArrayAccessBinary* for accessing the index using an arithmetic operation (such as $i + 1$). We consider these two tokens in the ACTS as well, since as we saw in Listing 2, such tokens can signal for runtime exceptions. Another set of tokens that convey actions are the Binary and Unary operations. A simple example is Listing 1 where *divide* operator signals for an *ArithmeticException*. Finally, we capture a set of special tokens to cover the actions that were not covered by the tokens explained so far. These tokens capture variable declarations, type-casts, object instantiations, using the *Null* literal, accessing objects' fields and returning methods' results.

2) Tokens capturing contextual information: One aspect of the contextual information are the code blocks that define the sequence of actions and therefore, can contribute to the occurrences of exceptions; for example, consider invoking the *subString()* method on a string value or accessing an array indexes; both of these actions may be more exception prone if done inside a loop. We define a set of special tokens to capture these blocks in a high-level manner. For example, the beginning of a loop is captured with *BeginLoop* and its ending is captured by *EndLoop*. Similarly, the beginning and ending of if-else blocks, try-catch-finally blocks (for checked exceptions) and switch-case blocks are captured. The other set of relevant tokens are the data types. Barbosa et al. [1] use variable types for recommending exception handling code and discuss that methods using variables of the same types are likely to implement more similar tasks. Other than this, certain data types can correlate with certain runtime exceptions occurrence (such as the use of numerical data types correlating with mathematical runtime exceptions). Class types also give context to actions and can correlate with exceptions relevance; for example, class *Integer* can correlate with the occurrence of *NullPointerException* while the primitive type *int* will not. Hence, we capture all primitive types and class names as tokens. The *Array* token is used to capture array types.

TABLE I: Tokens in Action-Context Token Sequence

Tokens Capturing Actions	Tokens Capturing Context
1. *Method calls*	10. BeginLoop, EndLoop (any loop block)
2. ArrayAccess, ArrayAccessBinary	11. BeginIf, Else, EndIf (if block)
3. *Binary* and *unary operations*	12. BeginSwitch, EndSwitch (switch block)
4. VarDec (variable declaration)	13. BeginTry, EndTry (try block)
5. Cast (act of type casting)	14. BeginCatch (catch in a try block)
6. New (object instantiation)	15. BeginFinally (finally in a try block)
7. Null (referencing "Null" literal)	16. Array
8. FieldAccess (access an object field)	17. *Primitive data types*
9. Return (method's return)	18. *Class names*

Table I provides a summary of the ACTS tokens, where literal tokens (e.g. Null) are expressed in regular font, and tokens that take values from the code are expressed in *italics*.

B. LA-Transformer Model

Figure 3 shows the high-level architecture of LA-Transformer, the deep learning model of D-REX. First, the trained Transformer layers extract a high-level representation of the given ACTS tokens. This representation is fed into two networks at the same time: a location prediction network (to predict the exception prone tokens) and an exception prediction network (to predict the exception types), both of which consist of several fully connected layers. The Transformer layers and the two prediction networks are trained jointly in an end-to-end fashion. In this section, we first give an introduction to Transformer [11], the feature extractor of D-REX, and discuss its difference with other deep neural network models that can be used for this purpose. Then we explain the location awareness concept in LA-Transformer and its training paradigm.

1) Transformer vs. Recurrent Neural Network: Recent advances in natural language modeling [26], [27], [11] have seen great success in modeling sequence data. Among them, Transformer is the state-of-the-art attention-based deep neural network model, specially designed for extracting high-level representations from sequence data. In contrast to other neural network models designed for modeling sequence data, such as recurrent neural networks (RNN) [12], Transformer does not rely on a recurrent structure which suffers from known issues such as catastrophic forgetting [28], [29] when modeling long sequences, where the network fails to model the representation

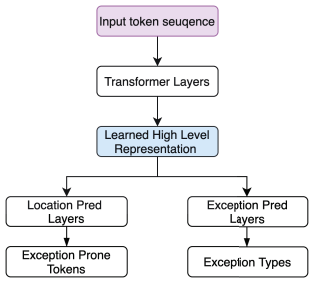


Fig. 3: Architecture of LA-Transformer Model

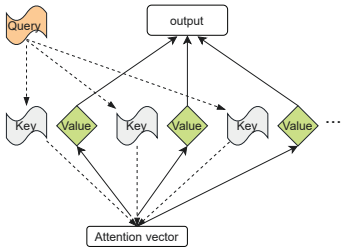


Fig. 4: An Illustration of Self-attention Mechanism

of the whole sequence due to forgetting. This is due to the fact that in Transformer, between every two layers, every output unit is directly connected with every input unit through the self-attention mechanism. This is useful when learning long method token sequences.

The structure of the self-attention mechanism is shown in Figure 4. Every input element (input token in the first layer, or the output of previous layer in the other layers) possesses three different feature vectors: a key, a value and a query vector, all learned during the training. The i-th attention vector is produced by the dot product between the i-th query vector and all key vectors. This attention vector is used as a weight to get a weighted average of the value vectors of all input elements, producing the i-th output element (shown as output in the figure). Therefore the i-th output element is directly connected with every input element in the self-attention layer. This connection is restricted in RNN layers as the i-th output unit can only connect with the j-th input unit if $j \leq i$. Further, the connection between the i-th output unit and the j-th input unit becomes weaker and weaker as the number of units between them increases in RNN layers.

2) Location Awareness: In the problem of exception type prediction, there is an extra piece of information in training data that we can exploit to provide more accurate recommendations: tokens located inside the associated try block. The exception types handled in a try-catch block of a method are strongly correlated with the tokens appearing in the try block. Such information is present in our training data; hence, given a try-catch block handling a runtime exception in a method, in the training phase, we augment the input ACTS tokens with location indices of tokens belonging to the try block, and use it as label information for location prediction layer during joint training. Note that this information is not necessary for the location prediction layers during inference.

Since training is done with an objective function to optimize these two prediction networks as well as the Transformer layers simultaneously, it forces the learned high-level representation to be location aware, hence, more expressive than a model without the location prediction network, which we refer to as the *plain Transformer* model. The plain Transformer model can only predict exception types (and not the exception prone tokens) and we compare D-REX using LA-Transformer with a version of it using plain Transformer in Section V-A

3) Training: During training, the loss function takes into account both the error from predicting the exception type and the error from predicting the tokens that belong to the try block. L, the joint objective function of exception prediction and location prediction, is formulated as follows:

$$L = L_{\text{excep}} + \lambda L_{\text{tryloc}} \tag{1}$$

In this equation, L_{excep} denotes the cross entropy loss resulting from the exception type prediction, and L_{tryloc} denotes the mean value of the loss of every token in the input, where each individual token uses a binary cross entropy loss [30]. λ is a tuning parameter between $[0, 1]$ balancing the magnitude of the two losses. In practice, we treated λ as a hyper parameter and conducted a grid search in a list of values: $[10^{-2}, 5 \cdot 10^{-2}, 10^{-1}, 5 \cdot 10^{-1}]$, where we found that 10^{-2} yields the best performance. Note that the weight parameters in Transformer layers and in the two prediction layers are trained jointly using equation (1).

The number of Transformer layers and other layer specifications such as the number of units in each layer were treated as hyperparameters, tuned through grid search. The best resulting architecture consists of 10 Transformer layers with a hidden dimension of 512 and an embedding size of 512. Both try location prediction and exception type prediction networks are parameterized by a two-layer fully connected neural network of width 100. With the output of these two networks, equation 1 is used to calculate the overall loss. Adaptive moment estimation (Adam) [31] optimizer with learning rate 3×10^{-3} was used to minimize equation 1. Both plain Transformer and LA-Transformer models were trained for 100 epochs and the top k (1, 2, 3, 5, 10) accuracies converged (stopped improving).

V. Evaluation

In this section, we present the results of evaluating D-REX's exception type and exception prone token predictions. The ground truth exception types in these experiments are the runtime exceptions handled by developers in try blocks. For the exception prone token predictions, we present quantitative and qualitative analyses by looking at the top-k predictions by D-REX. In quantitative analysis, the ground truth for the exception prone tokens are the tokens inside each try block. In all experiments, for each try block of interest in each method, we removed the keyword *try* and the exception handling code. This way we ensured that the input to our model does not contain any tokens related to the exception handling code.

A. Exception Type Prediction

1) Baselines: In order to compare D-REX with the existing methods of exception type prediction, we searched for available tools in this area. Although there has been much work in recommending exception handling strategies, the only approach we found that predicts runtime exceptions is FuzzyCatch [10], [2]. The core of FuzzyCatch is a statistical model based on the co-occurrences of the API method calls and exception types, trained and tested using a set of Android projects. Hence, there are two issues that make a direct comparison between D-REX and FuzzyCatch not valid: (1) The FuzzyCatch model was trained exclusively on Android code, while the dataset used for training our D-REX model is based on randomly selected Java projects and does not focus on Android projects. This matters, because there is considerable divergence of vocabulary between Android APIs and regular Java APIs: a model trained on one vocabulary may not do well on the other. (2) FuzzyCatch is offered as IntelliJIdea and AndroidStudio plugins, and expects a human in the loop; unlike D-REX, it is not possible to run FuzzyCatch headless on a large source code dataset. Since a direct comparison with this tool is not feasible, we developed a baseline inspired by their technique that works based on the co-occurrences of method calls and the exception types and we trained it on the 200k train set. We name this baseline *Method-Exception-Frequency (M-E-F)*. For each method, M-E-F collects all method calls in it and then queries the train dataset for the co-occurrences of each method call and all exception types. Using a fuzzy union formula (presented in [10], [2]), it then aggregates the results for all method calls and recommends the possible exceptions sorted by their predicted probability.

As we discussed in Section III, the proportion of the runtime exceptions in our dataset is imbalanced. While *NumberFormatException*, for example, has appeared more than 140,000 times, there are exceptions that have appeared only a few hundred times. As a result, one may wonder how a model that recommends runtime exceptions randomly with a chance proportional to their occurrences would work. To answer this question, we developed a model, which we call *Random* baseline, that recommends runtime exceptions randomly according to the frequency of the exception types in the train dataset. In particular, an exception that has appeared N times in the train dataset gets N times the chance of being predicted, compared to the exception that has appeared only once. In order to have reproducible results and gain a broader perspective, for this baseline, each time a set of five seeds are generated randomly and using these seeds, the Random baseline generates five different results. The accuracy numbers represent the average across all these runs.

Moreover, to compare the LA-Transformer component of D-REX with other potentially suitable deep learning models, we trained two other models: (1) a bidirectional LSTM model [13] (Bi-LSTM), a variant of RNN models, and (2) a plain Transformer model (P-Trans) (described in Section IV-B). For each of these, LA-Transformer in D-REX is replaced with the corresponding model and the top k predictions are collected.

All baselines are trained on the 200k training dataset, and their accuracy is compared with D-REX's accuracy on 200k test dataset as well as a separate set of unseen java projects.

2) Accuracy Comparison on 200k Test Set: We first evaluate D-REX and other baselines on the test portion of the 200k dataset (44,245 samples). Table II shows the accuracy of each approach on its Top 1, Top 2, Top 3, Top 5 and Top 10 recommendations. Top 1 accuracy, for example, presents the proportion of the Top 1 recommendations by each approach that matches the set of true runtime exceptions present in the dataset. More formally, if m is a method and E_m is the set of runtime exceptions caught by a try block in m, and $\hat{E}_{K,m}$ is the list of top K runtime exceptions predicted by an approach for m, sorted based on the descending order of the predicted scores, then $TopK$ accuracy over N samples is defined as:

$$\frac{1}{N}\sum_{m=1}^{N} 1_{(E_m \cap \hat{E}_{K,m} \neq \emptyset)} \tag{2}$$

where $1_{(E_m \cap \hat{E}_{K,m} \neq \emptyset)}$ is an indicator function that equals 1 when $E_m \cap \hat{E}_{K,m} \neq \emptyset$ and 0 otherwise. This measurement is similar to the accuracy measurement used by FuzzyCatch and as they have also pointed out, we are not able to evaluate the results using Precision due to the lack of negative examples: given a piece of code, it is almost impossible to determine that a specific runtime exception will never happen for it. As for the recall number, the top k recall depends on the number of the true exceptions in a method, which varies method by method. However, if there is only one exception in the method, then our top 1 accuracy is equivalent to recall.

As the table shows, D-REX has been able to achieve the highest accuracy across all categories of Top K predictions. The difference, however, is mostly notable in the Top 1 results which is of utmost importance as the Top 1 prediction is the one that developers pay more attention to. In particular, D-REX has ≈81% accuracy in its Top 1 predictions, compared to Plain Transformer which is the second best one with ≈79% Top 1 accuracy. This indicates that the extra information about the tokens inside the try block helps D-REX to achieve better results in the exception type prediction. It also implies that the exception prone token prediction loss, L_{tryloc} in equation 1, has a regularization effect for the LA-Transformer model to alleviate the over-fitting during training, leading to an improvement in exception type prediction on the test set. M-E-F has the third rank in Top 1 accuracy with ≈12% of difference with D-REX. The Bi-LSTM model was not able to achieve comparable results which can attest the effectiveness of LA-Transformer model over it. The Random model achieved poor results compared to other approaches showing that a random approach purely based on the observed frequency of exceptions cannot serve as a viable solution to our problem.

3) Accuracy on Unseen Projects: To understand the effectiveness of D-REX on a variety of projects and those that were not included in the 200k dataset, we downloaded a separate set of Java projects from GitHub, shown in Table III. For these projects, we downloaded their default branch's version. In the

TABLE II: Exception Prediction Accuracy on 200k Test Set

	Top 1	Top 2	Top 3	Top 5	Top 10
M-E-F	69.67%	81.37%	86.30%	92.38%	96.46%
Random	16.75%	31.36%	43.39%	61.08%	82.33%
Bi-LSTM	59.32%	71.71%	78.56%	85.76%	93.12%
Plain Transformer	79.39%	87.49%	90.67%	93.74%	96.68%
D-REX	**81.09%**	**88.91%**	**91.70%**	**94.52%**	**97.08%**

TABLE III: Downloaded Java Projects Summary

Project	URL	Last commit	LOC	#Samp.
Batik	apache/xmlgraphics-batik	Jan 12, 2021	190,597	172
Lucene	apache/lucene-solr	Jan 29, 2021	1,350,727	411
Xalan	apache/xalan-j	May 23, 2019	170,574	218
Cassandra	apache/cassandra	Jan 29, 2021	464,594	211
SonarQube	SonarSource/sonarqube	Feb 17, 2021	516,101	203
Camel	apache/camel	Feb 17, 2021	1,539,888	213
IntelliJ IDEA CE	JetBrains/intellij-community	Feb 17, 2021	4,095,728	789
Hibernate ORM	hibernate/hibernate-orm	Feb 17, 2021	779,538	333
Object Teams	eclipse/objectteams	Jan 14, 2021	1,868,449	371
MPS	JetBrains/MPS	Feb 17, 2021	1,872,299	162
Ignite	apache/ignite	Feb 16, 2021	1,170,332	333
CloudStack	apache/cloudstack	Feb 12, 2021	661,128	283

table, we show the GitHub URL, the last commit date when we downloaded the repository, number of source code lines (LOC) calculated with IntellijIdea's Statistic plugin [2] and the number of samples (try-catch blocks with runtime exceptions) fetched from the project. To select these projects, we first downloaded and processed the set of Java projects used by Hindle et al. [32], and included the ones with 100 or more samples to make sure that the evaluation is done on large enough datasets (first 4 rows of Table III). Then, to expand the comparison to more projects, we queried the database provided by Lopes et al. [20] for projects that are not included in 200k dataset and have more than 5,000 files, and selected 8 more projects. The rest of the projects listed in Table III were selected in this step.

Table IV shows the the exception prediction accuracy of D-REX and other baselines on each of these projects, and overall on all of them, for the Top 1 to 3 predictions. In general, we see that D-REX is achieving the highest accuracy compared to all other baselines in Top 1 predictions, except for two projects: Hibernate ORM and MPS. For MPS, M-E-F achieves a better accuracy, with a difference of ≈3%, and for Hibernate ORM, M-E-F performs better by a margin of ≈6%. On other projects, D-REX has an accuracy improvement over P-Trans in a range of ≈0.2% to ≈8%, and an improvement over M-E-F in a range of ≈3% to ≈30%. We see a similar trend of D-REX performing better than other baselines in Top-2 and Top-3 predictions for the majority of projects. In the overall results, D-REX is performing the best in Top 1 and Top 2 predictions, while in Top 3 predictions, P-Trans has slightly better accuracy by a margin of less than 0.5%.

B. Exception Prone Tokens Prediction

The other output of D-REX is the set of tokens that can be responsible for the predicted exceptions, in the order of their predicted probability. We evaluate the correctness of these predictions in two ways: a quantitative analysis to measure the precision of these predictions with respect to the tokens in the ground truth try blocks, and a qualitative analysis aimed at investigating the relevance of predictions with the possible exceptions through a manual analysis.

1) Quantitative Analysis: For each try-catch block present in the dataset that handles a runtime exception, we consider the method's ACTS tokens that are inside the try block as the true tokens responsible for the exception. The reason is that when developers implement try-catch blocks, they form the try block around the portion of code that they believe can cause the exception being handled. We evaluate the correctness of theses predictions with precision: for each sample, we measure the percentage of the top k predictions that are included in the ground truth try block. We do this evaluation for $k = 1, 2, 3$ and report the mean value precision ($Mean_{Precision}$) across all of the samples in the 200k test set as well as the samples from the unseen projects dataset explained in Section V-A3. For each value of k, we evaluate D-REX on the samples that have a try block consisting of more than k tokens and a method body length of more than than $2 * k$ tokens. The former is done since it is not meaningful to look at top k predictions if the true try block has less than k tokens. The latter is done to eliminate the cases where the number of tokens in method is close to k and hence, D-REX has a high a chance of having its top k predictions to be true predictions.

Table V shows the results of evaluating the top k predictions (where $k = 1, 2, 3$) on the two datasets. On the 200k test set, $Mean_{Precision}$ of top 1 predictions ($k = 1$) is 75%, suggesting that 75% of the times, the top 1 predicted token has actually been located in the try block. This value is 57% for the unseen projects dataset. The $Mean_{Precision}$ values for $k = 2$ and $k = 3$ are also very close to the values reported for $k = 1$. This is promising as it can help developers to identify the source of exceptions by looking at the top 3 predictions.

It is worth mentioning that we did not measure recall here since while the length of try blocks can vary, D-REX always produces top k predictions. Therefore, when a method has a long try block, recall will be dominated by the length of try block no matter how accurate the prediction is. Also, in predicting the exception prone tokens, it is the precision of top K predictions that matters most to the developer, not the fraction of all tokens in the true try block that are captured. There can be tokens in the true try block that are not directly related to the exception but has been placed in the try block because they form a code construct.

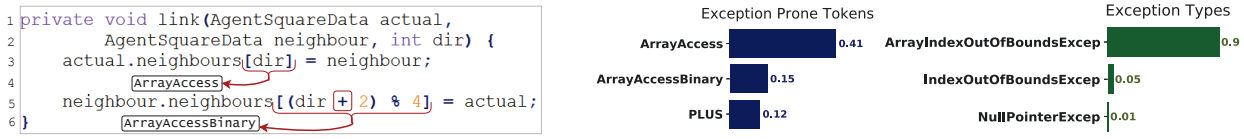
2) Qualitative Analysis: To gain a deeper insight on the usability of D-REX's predicted exception prone tokens in real coding scenarios, we manually investigated its predictions for a number of methods from the 200k test dataset. Figure 5 shows three of these methods, where in each example, the left part shows the method with its predicted exception prone ACTS tokens circled, and the right half shows the predicted exception types and exception prone tokens with their probability scores.

The example in Figure 5(a) shows a method that can throw *ArrayIndexOutOfBoundsException*. The top 3 token predictions

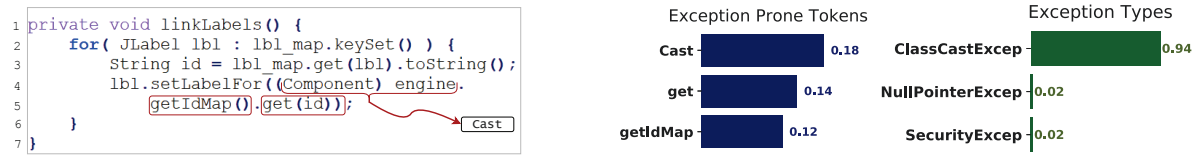
[2]https://plugins.jetbrains.com/plugin/4509-statistic

TABLE IV: Exception Prediction Accuracy Comparison on Unseen Java Projects

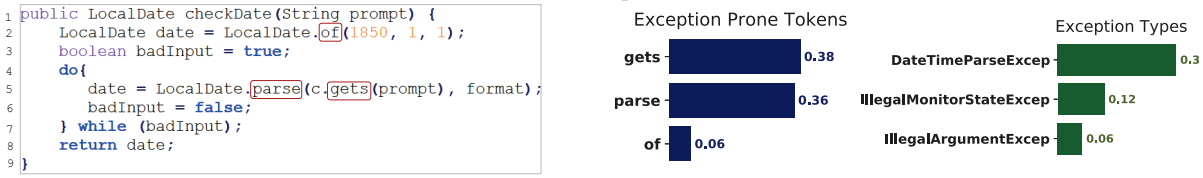
Project	Top 1					Top 2					Top 3				
	D-REX	P-Trans	Bi-LSTM	M-E-F	Rand	D-REX	P-Trans	Bi-LSTM	M-E-F	Rand	D-REX	P-Trans	Bi-LSTM	M-E-F	Rand
Batik	**81.40%**	73.26%	51.74%	51.79%	16.51%	**84.88%**	82.56%	58.72%	54.17%	32.33%	**87.21%**	84.88%	65.70%	57.14%	41.98%
Lucene	**64.08%**	60.44%	45.50%	52.27%	16.06%	**73.06%**	73.03%	59.12%	70.20%	30.41%	76.70%	**77.43%**	69.83%	75.00%	41.75%
Xalan	**88.53%**	88.07%	28.90%	76.70%	6.79%	**92.20%**	91.74%	45.87%	80.10%	13.03%	**94.04%**	93.58%	87.16%	81.55%	19.08%
Cassandra	**60.38%**	54.72%	54.03%	48.82%	15.45%	**71.70%**	71.23%	66.35%	59.24%	29.76%	**77.36%**	75.00%	74.41%	65.88%	43.03%
SonarQube	**69.46%**	62.56%	45.81%	53.20%	14.19%	**76.85%**	71.43%	70.44%	70.44%	26.90%	**83.74%**	79.31%	82.27%	82.76%	39.11%
Camel	**61.03%**	53.05%	47.89%	49.29%	13.05%	**74.18%**	73.71%	68.54%	72.99%	25.44%	**84.04%**	78.87%	75.12%	77.73%	36.53%
IntelliJ IDEA CE	**60.71%**	60.58%	49.68%	51.39%	18.80%	71.23%	**71.48%**	60.58%	69.25%	33.41%	75.41%	**76.30%**	69.33%	73.47%	44.92%
Hibernate ORM	36.94%	39.34%	38.44%	**43.37%**	12.01%	**68.47%**	61.86%	63.66%	59.04%	23.66%	74.77%	78.38%	**81.68%**	71.69%	34.17%
Object Teams	**59.30%**	58.76%	41.51%	56.76%	11.81%	74.12%	**77.63%**	49.87%	72.37%	22.47%	79.25%	**82.21%**	56.87%	78.08%	32.72%
MPS	46.91%	43.83%	45.06%	**49.04%**	18.52%	63.58%	51.85%	62.35%	**68.15%**	31.60%	66.05%	70.37%	**72.22%**	71.34%	45.93%
Ignite	**63.06%**	61.56%	37.54%	52.58%	13.09%	**71.47%**	68.17%	52.85%	65.96%	24.44%	**75.68%**	72.37%	68.47%	**75.68%**	33.21%
CloudStack	**87.63%**	86.57%	65.37%	71.53%	20.42%	93.29%	**94.35%**	76.68%	91.24%	37.60%	95.05%	**96.47%**	81.98%	95.62%	52.30%
Overall	**62.75%**	61.34%	46.09%	55.33%	15.11%	**74.78%**	73.91%	60.61%	65.96%	28.26%	79.29%	**79.78%**	72.48%	75.68%	39.17%



(a) Example 1



(b) Example 2



(c) Example 3

Fig. 5: Qualitative Analysis Examples

TABLE V: Exception Prone Tokens Prediction: $Mean_{Precision}$

Dataset	$k=1$	$k=2$	$k=3$
200k test set	75%	71%	69%
Unseen projects	57%	56%	56%

are *ArrayAccess*, *ArrayAccessBinary* and *PLUS* (pointing to plus operator). We see that *dir*, used as index for accessing the array, is a parameter whose value is determined at runtime, and hence, using it to access the array index at line 3 can throw the predicted exception. Even if the exception does not happen at this line, it is possible to happen at line 5 when the array index is accessed by doing an addition operation with *dir*. Thus, the *ArrayAccessBinary* is also a true exception prone token.

Figure 5(b) presents a method susceptible to *ClassCastException*. The top 1 predicted token is *Cast* (a derived token corresponding to the type casting implemented at line 4) which is a true prediction. The next two predicted exception prone tokens are *getIdMap()* and *get()*. At first look, they both seem to be false positive predictions, however, a close look shows that the second predicted exception type is *NullPointerException* (although with low probability) and these tokens can throw such exception. This case shows that although the focus is often on the top 1 predicted exception prone token, the next predictions give insights to the developer as well.

We also looked at examples where the top 1 predicted exception prone token is incorrect, as in Figure 5(c). This method is prone to *DateTimeParseException* with the top 2 exception prone tokens predicted as: *gets()* and *parse()*. The second prediction, *parse()*, is directly related to the exception and *gets()* is not. A reason can be that *DateTimeParseException* is a rare exception, with only 357 appearances in the 200k dataset. As a result, D-REX has not seen many examples of the tokens accompanied by this exception. Although it predicted the true token responsible for the exception, this token is in the second rank. An important note here is that the top 1 and top 2 predicted tokens have a close predicted probability showing that they were close choices for D-REX.

The above examples suggest that D-REX is able to produce meaningful predictions favoring the potentially exception prone tokens. Therefore, it can help the developer in making better judgements on the causes for exceptions in their code. During inference, on average, it takes 0.1s to produce both exception type and exception prone token predictions for each sample tested on an 8-core Intel(R) Xeon(R) E5-2620 v4 CPU.

VI. Related Work

Java exceptions and exception handling. A very recent related work is FuzzyCatch [2], [10], which focuses on recommending both exception types and exception handling code in the Android ecosystem. The recommendation of exception types is based on the co-occurrences of API method calls and exception types in their dataset. D-REX is different than FuzzyCatch as D-REX works for Java (in general) and can predict exception prone tokens as well. Also, we do not focus on recommending exception handling code. Barbosa et al. [1], [33] propose heuristic strategies based on the code context (structural information of code) to search for exception handling code. Rahman et al.'s approach [17] recommends exception handling code by doing a code search on a number of popular GitHub open source repositories. EH-Recommender [18] leverages program context in recommending exception handling code, where program context can be exceptional, architectural or functional. Maestro [34] recommends the StackOverflow post mostly related to a runtime exception using a special code representation, called Abstract Program Graph. Our work is different than these as we recommend relevant runtime exception types and exception prone code elements, not the exception handling code or post. Nakshatri et al. [5] studied practices and patterns of handling checked exceptions revealing exception types higher in the class hierarchy being handled more than the concrete ones. Sena et al. [14] studied exception handling practices on a set of 656 Java libraries, finding a large portion of undocumented runtime exceptions in the studied libraries, and finding API runtime exception handling to be more frequent than API checked exception handling. Nguyen et al. [3] studied 246 exception related bugs from Android apps, finding 51% of them to be thrown by Android API method calls and that runtime exception are more prevalent than checked ones. Nexgen [19] is a neural network-based approach for predicting the location of try blocks and generating exception handling code. NexGen's try block location prediction is done at the statement level, while D-REX produces more fine-grained predictions at token level, which can be more helpful in finding the root cause of an exception. Another difference is that D-REX only focuses on predicting harder to predict runtime exceptions and delegates the task of dealing with the exceptions to the developer (to catch the exception or to prevent it).

Source code representation learning. Another line of related work is source code representation learning. As discussed by Chen et al. [35], these approaches can largely be divided into learning token level and function (or method) level embeddings. In Code2Vec [36] and Code2Seq [37], authors propose to learn the 'code embedding' as continuous distributed vectors, which is similar to our goal. However, they rely on decomposing the method into a collection of paths in its abstract syntax tree (AST), and learn the representation vector for each path, as well as the the amount of attention to put on each paths. D-REX simplifies this process and only needs to pre-process the code into a single sequence, then it leverages the representation learning ability of the Transformer model to summarize the context of the source code. Moreover, since one of our goals is to predict the exception prone likelihood of each token in the input sequence, a finer grained embedding at the token level is needed. Therefore the AST path level embedding may be less effective to infer whether each token is exception prone. Another recent work in building contextual embeddings of source code is CuBERT [38], which directly uses the architecture of Transformer [11] and hence, is similar to our plain Transformer model. However, CuBERT is not specifically adapted for exception type recommendation in Java. Moreover, it does not leverage the information about the tokens in the try block and does not recommend exception prone tokens.

VII. Threats to Validity

Runtime exceptions and try blocks used in training and evaluation of this work are extracted from a set of Java open source repositories from GitHub. We did not validate these exceptions and their associated try blocks with respect to their correctness, however, we used enough projects (200k) to address for any inaccuracies in the projects. We used the JavaParser to parse these projects and any errors in this tool may affect the results. We aimed for comparing D-REX with the state-of-the-art tool, FuzzyCatch, however, since FuzzyCatch is exclusively trained on Android ecosystem, we were not able to perform a direct comparison. We implemented a baseline inspired by FuzzyCatch's approach to mitigate this issue.

VIII. Conclusions and Future Work

Runtime exception handling is an important and challenging task that requires understanding the runtime state of programs. We proposed D-REX to ease this task by simultaneously achieving two goals: 1) prediction of the relevant exception types given un-labeled code, and 2) prediction of the exception prone parts of the code. These two objectives are optimized at the same time by training D-REX from end to end. For training and evaluation of D-REX, we created a benchmark dataset which will be made publicly available. We demonstrated through experiments that D-REX significantly outperforms multiple non-Transformer baselines on two separate datasets.

Our next step is to package D-REX as an IDE plugin: developers will be able to ask for possible exceptions as they code and the corresponding exception prone code elements will be highlighted. Further, it is interesting to explore other tasks that can utilize the location awareness of LA-Transformer. One example is code refactoring where the location awareness can be used to find parts of code that need refactoring. More generally, D-REX provides a way to leverage any kind of extra information in the training data, and its application is not limited to using the try block information from training data.

REFERENCES

[1] E. A. Barbosa, A. Garcia, and M. Mezini, "Heuristic strategies for recommendation of exception handling code," in *2012 26th Brazilian Symposium on Software Engineering*, pp. 171–180, IEEE, 2012.

[2] T. Nguyen, P. Vu, and T. Nguyen, "Recommending exception handling code," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 390–393, IEEE, 2019.

[3] T. T. Nguyen, P. M. Vu, and T. T. Nguyen, "An empirical study of exception handling bugs and fixes," in *Proceedings of the 2019 ACM Southeast Conference*, pp. 257–260, 2019.

[4] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold, "Fault localization and repair for java runtime exceptions," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, pp. 153–164, 2009.

[5] S. Nakshatri, M. Hegde, and S. Thandra, "Analysis of exception handling patterns in java projects: An empirical study," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pp. 500–503, IEEE, 2016.

[6] M. Asaduzzaman, M. Ahasanuzzaman, C. K. Roy, and K. A. Schneider, "How developers use exception handling in java?," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pp. 516–519, IEEE, 2016.

[7] "Best (and worst) java exception handling practices." https://able.bio/DavidLandup/best-and-worst-java-exception-handling-practices--18h55kh. Accessed: 2020-08-15.

[8] "Beware the dangers of generic exceptions." https://www.infoworld.com/article/2073800/beware-the-dangers-of-generic-exceptions.html. Accessed: 2020-08-15.

[9] S. Jiang, H. Zhang, Q. Wang, and Y. Zhang, "A debugging approach for java runtime exceptions based on program slicing and stack traces," in *2010 10th International Conference on Quality Software*, pp. 393–398, IEEE, 2010.

[10] T. Nguyen, P. Vu, and T. Nguyen, "Code recommendation for exception handling," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1027–1038, 2020.

[11] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), vol. 30, Curran Associates, Inc., 2017.

[12] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[13] M. Schuster and K. Paliwal, "Bidirectional recurrent neural networks," *Trans. Sig. Proc.*, vol. 45, p. 2673–2681, Nov. 1997.

[14] 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*, pp. 212–222, 2016.

[15] S. Banerjee, L. Clapp, and M. Sridharan, "Nullaway: Practical type-based null safety for java," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 740–750, 2019.

[16] B. Gaudin, E. I. Vassev, P. Nixon, and M. Hinchey, "A control theory based approach for self-healing of un-handled runtime exceptions," in *Proceedings of the 8th ACM international conference on Autonomic computing*, pp. 217–220, 2011.

[17] M. M. Rahman and C. K. Roy, "On the use of context in recommending exception handling code examples," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pp. 285–294, IEEE, 2014.

[18] 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 *2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS)*, pp. 104–114, IEEE, 2018.

[19] J. Zhang, X. Wang, H. Zhang, H. Sun, Y. Pu, and X. Liu, "Learning to handle exceptions," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 29–41, IEEE, 2020.

[20] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "Déjàvu: a map of code duplicates on github," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[21] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[22] "Java parser." https://javaparser.org/. Accessed: 2020-06-03.

[23] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, pp. 207–216, IEEE, 2013.

[24] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 763–773, 2017.

[25] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: Detection of clones in the twilight zone," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 354–365, 2018.

[26] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018.

[27] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26* (C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, eds.), pp. 3111–3119, Curran Associates, Inc., 2013.

[28] M. McCloskey and N. J. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," *Psychology of Learning and Motivation*, vol. 24, pp. 109–165, 1989.

[29] R. Ratcliff, "Connectionist models of recognition memory: constraints imposed by learning and forgetting functions.," *Psychological review*, vol. 97 2, pp. 285–308, 1990.

[30] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. Springer Series in Statistics, New York, NY, USA: Springer New York Inc., 2001.

[31] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[32] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 837–847, IEEE, 2012.

[33] E. A. Barbosa, A. Garcia, and M. Mezini, "A recommendation system for exception handling code," in *2012 5th International Workshop on Exception Handling (WEH)*, pp. 52–54, IEEE, 2012.

[34] S. Mahajan, N. Abolhassani, and M. R. Prasad, "Recommending stack overflow posts for fixing runtime exceptions using failure scenario matching," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1052–1064, 2020.

[35] Z. Chen and M. Monperrus, "A literature study of embeddings on source code," *CoRR*, vol. abs/1904.03061, 2019.

[36] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *CoRR*, vol. abs/1803.09473, 2018.

[37] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[38] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," (Vienna, Austria), 2020.