DeepAnna: Deep Learning based Java Annotation Recommendation and Misuse Detection

Yi Liu[1,2], Yadong Yan[1,2], Chaofeng Sha[*1,2], Xin Peng[1,2], Bihuan Chen[1,2] and Chong Wang[1,2]
[1]School of Computer Science, Fudan University, Shanghai, China
[2]Key Laboratory of Data Science, Fudan University, Shanghai, China
{19212010005, 20212010127, cfsha, pengxin, bhchen, wangchong20}@fudan.edu.cn
[*]Corresponding author

***Abstract*—Annotations have been widely used in Java programs to support additional compile-time, deployment-time, and runtime processing. Developers use annotations to delegate repetitive logics such as object initialization and request forwarding to compilers and runtime frameworks. Therefore, these annotations are important for the correct execution of programs. In practice, however, developers often find it hard to correctly use annotations and the misuse of annotations has led to real bugs in Java programs. In this paper, we conduct an empirical study on Stack Overflow questions to investigate the major development frameworks that are involved in questions about Java annotations and the main problems encountered by developers in the use of Java annotations. Based on the findings of the study, we propose DeepAnna, a deep learning based Java annotation recommendation and misuse detection approach. Based on a corpus of Java programs with intensive use of annotations, DeepAnna trains a deep learning based multi-label classification model by considering both the structural and textual contexts of source code. DeepAnna can recommend annotations at both class level and method level. Our evaluation with a large corpus of open-source Java projects shows that DeepAnna outperforms state-of-the-art text multi-label classification approaches in annotation recommendation and can effectively detect annotation misuses. Based on our analysis, we submit 85 bug-fixing pull requests for annotation misuses in open-source projects and 20 of them have been accepted and merged.**

***Index Terms*—code; Java annotation; deep learning; multi-label classification;**

I. Introduction

Since introduced in JDK 5, annotations have been widely used in Java programs. Annotations provide a convenient way to implement additional compile-time, deployment-time, and runtime processings [1]. Developers can use annotations to delegate repetitive logics such as object initialization and request forwarding to compilers and runtime frameworks. In this way, the complexity of programming can be reduced and the efficiency of development can be improved. Due to these advantages, some popular Java frameworks, such as Spring [2] and Hibernate [3] (two mainstream frameworks for back-end application development), provide a rich set of well-designed annotations to implement various characteristics. For example, `@Autowired` is one of the most important annotations in Spring, which controls where and how object dependencies are injected implicitly. Therefore, correctly understanding and using these annotations has been a premise for using the frameworks.

In practice, however, developers often find it hard to correctly use annotations and the misuse of annotations has led to real defects in Java programs. As a framework may define many annotations, the developers often have difficulties determining when annotations should be used and which one to choose among similar annotations. For example, the Spring framework defines over 300 annotations and just about network request there are 14 annotations. Therefore, developers may misuse annotations and introduce annotation-related defects. For example, Figure 1 shows two commits that fix Spring annotation misuse defects in open-source projects. The commit in Figure 1(a) fixes an annotation defect by replacing `@Service` with `@Repository`. These two annotations are both used to implement the feature of inverse of control (IOC), but they work at different layers (service layer and persistence layer respectively). The commit in Figure 1(b) fixes an annotation defect by adding a `@Service` annotation. This missing annotation defect makes the framework not able to manage the service and implement the feature of IOC. These annotation misuse defects will cause the programs to malfunction. Therefore, it is quite helpful if a tool can automatically recommend suitable annotations for developers based on code context and detect annotation misuse defects in the programs.

To investigate the requirements for annotation recommendation and misuse detection, we first conduct an empirical study on Stack Overflow questions about Java annotations. The purposes of the study are twofold, i.e., investigating the major development frameworks that are involved in questions about Java annotations (i.e., RQ1) and collecting the main problems encountered by developers in the use of Java annotations (i.e., RQ2). The results show that the most frequently mentioned frameworks in the questions are Spring, Hibernate, Junit, Jakarta EE, Guice, Jackson, and Lombok. Based on 263 manually labelled Stack Overflow questions, we identify 7 categories of Java annotation problems encountered by developers, including "how to use a specific annotation", "how to define a custom annotation", "which annotation to use", "how to resolve the used annotations", "what is the difference between annotation and XML", "what is annotation", and others. Most of the problems are related to annotation usage, for example "how to use a specific annotation" and "which annotation to use", thus may result in difficulties and even

```
1 - @Service                                                    Deleted Line in this Commit
2 + @Repository                                                 Added Line in this Commit
3   @RequiredArgsConstructor
4   public class PriceComponentRepositoryImpl implements PriceComponentRepository {
5       private final PriceComponentDao priceComponentDao;
6       private final PriceComponentMapper priceComponentMapper;
7       @Override
8       public void createPriceComponent(PriceComponent priceComponentToCreate)
9               throws PriceComponentAlreadyExistsException {
10          if (getPriceComponentEntity(priceComponentToCreate.getName())
11                  .isPresent())  {
12              throw new PriceComponentAlreadyExistsException(String.format("Price Component already exists: %s",
13                      priceComponentToCreate.getName()));
14          }
15          priceComponentDao.save(priceComponentMapper.mapToPriceComponentEntity(priceComponentToCreate));
16      }
17      ...
18  }
```

(a) Commit 11f3d9f343ff0460172e12d8abaaf238a6cca1d3 in TicketService

```
1 + @Service                                                    Added Line in this Commit
2   public class JokeServiceImpl implements JokeService {
3
4       private final ChuckNorrisQuotes quotesGenerator = new ChuckNorrisQuotes();
5       @Override
6       public void getJoke() {
7             return quotesGenerator.getRandomQuote();
8       }
9   }
```

(b) Commit a8c588b8e00f510e63a0c351edf9f59bddeb015b in spring5-jokes-app-v2

Fig. 1. Two Commits that Fix Annotation Misuses in Open-Source Projects

defects in annotation usage.

Based on the findings of the study, we propose a deep learning based approach (called DeepAnna) for Java annotation recommendation and misuse detection. We model the problem of annotation recommendation and misuse detection as a multi-label classification task, which takes source code as input and produces annotations as classification labels. DeepAnna includes a training phase and a prediction phase. In the training phase, DeepAnna constructs a training set based on a corpus of Java programs with intensive use of annotations. Each training sample in the set consists of a code snippet and its corresponding Java annotations. Given a training sample DeepAnna extracts structural and textual contexts for the code snippet from its abstract syntax tree (AST) and identifiers repetitively. These code contexts are then fed into a classification model with two encoding modules, a fusion layer, and a output layer. The output of the model is the probabilities of all the candidate annotations. Based on the probabilities, the model is continuously optimized to maximize the probabilities of the true annotations of the code snippet and minimize the probabilities of the false annotations. In the prediction phase, DeepAnna extracts code contexts from a given code snippet in the same way and uses the trained model to predict the probabilities of all the candidate annotations based on the code contexts. Based on the probabilities DeepAnna selects a set of annotations for the code snippet. It can then recommend the predicted annotations to developers directly or detect annotation misuses by comparing the currently used annotations with those suggested. Note that DeepAnna supports the annotation recommendation and misuse detection at both class level and method level by adopting different structural code context extraction methods.

We conduct a series of experimental studies to evaluate the effectiveness of DeepAnna. We construct a dataset containing 1,000 open-source Java projects using the seven most popular frameworks identified in the empirical study. Based on the dataset, we train an annotation prediction model and use it to evaluate the accuracy of annotation recommendation. The results show that DeepAnna achieves 85.14% of average precision and 83.54% of F1 for annotation recommendation at the class level and 72.22% of average precision and 67.55% of F1 for annotation recommendation at the method level, outperforming the state-of-the-art text multi-label classification approaches on the dataset. Our evaluation also confirms the usefulness of both the structural context and textual context for annotation prediction. Moreover, we use the trained model to detect annotation misuse defects in Java open-source projects. Based on the results, we submit 85 bug-fixing pull requests for annotation misuse defects in open-source projects and 20 of them have been accepted and merged.

This paper makes the following contributions.

- We conduct an empirical study on the major development frameworks involved in annotation related questions and the main problems encountered by developers in the use of Java annotations.
- We propose a deep learning based annotation recommendation and misuse detection approach based on both the structural and textual context of code.
- We evaluate the effectiveness of the approach based on a set of Java open-source projects by analyzing the average precision and F1 of annotation recommendation and detecting real annotation misuse defects in open-source projects.

II. Empirical Study

To investigate the requirements for annotation recommendation and misuse detection, we conduct an empirical study to answer the following two research questions.

- **RQ1**: What are the major development frameworks that are involved in questions about Java annotations?
- **RQ2**: What are the main problems encountered by developers in the use of Java annotations?

A. Dataset

We select a set of annotation related questions from Stack Overflow as the subjects of the study in the following way. We select the questions with both “java” and “annotations” tags from Stack Overflow. To ensure the quality of Stack Overflow posts to be analyzed, we eliminate questions based on two criteria, the first is that the post must have an answer accepted by the questioner, and the second is that the post has no less than 20 likes. A question with a high number of votes reflects that it is common and helpful to programmers. The process results in 263 Stack Overflow questions as the dataset.

B. RQ1: Major Frameworks involved in Annotation Related Questions

By extracting and analyzing the framework names that appear in the question tags, we identify seven major Java frameworks that are most frequently mentioned in the 263 annotation related Stack Overflow questions as shown in

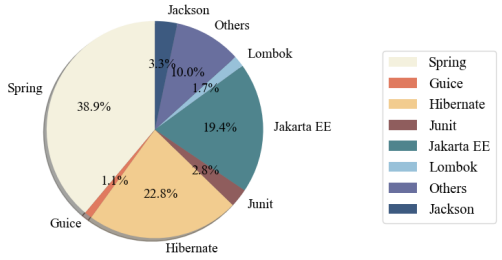

Fig. 2. Major Frameworks Involved in Annotation Related Questions

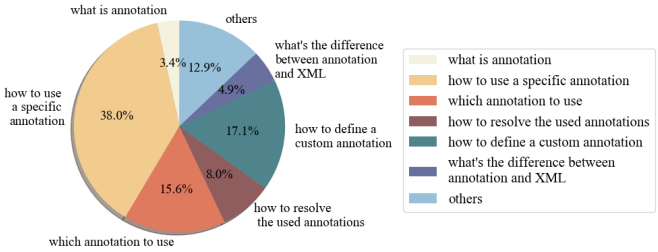

Fig. 3. Main Problems Encountered by Developers

Figure 2. These frameworks account for 90% of the frameworks discussed in the questions. The most popular framework discussed in these questions is Spring, which includes basic Spring Framework, Spring Boot, Spring MVC, and Spring Data. This result is consistent with the popularity of Spring in Java development. In addition, Hibernate, Junit, Jakarta EE, Lombok, Guice, and Jackson are also frequently mentioned in the questions. We consider the seven Java frameworks as the targets of annotation recommendation and misuse detection in our current implementation and evaluation.

C. RQ2: Main Problems Encountered by Developers

To answer RQ2, we manually identify a set of question categories from the questions in the dataset using open coding and then classify the questions into different categories. Three authors of the paper conduct the coding together, starting from a seed code "how to use an annotation". For each question, they decide which code the question can be classified into by discussion. If a question cannot be classified into an existing code, they create a new code or modify the name and definition of an existing code to accommodate the question. If a new code is created or an existing code is modified, they re-annotate all the questions that have been annotated. The open coding process ends when all the questions have been classified into a question type (i.e., code). Based on the question categories, we invite two master students who are familiar with Java annotations (not involved in the open coding process) to classify the 263 questions in the dataset into the categories independently. If their classifications of a question are different, a third master student familiar with Java annotations is assigned to resolve the conflict using the majority-win policy. We calculate the Cohen's Kappa coefficient [4] of the classifications of the two students before conflict resolution and the result is 0.858 (i.e., almost perfect agreement).

The resulting question categories and the percentages of different categories are shown in Figure 3. The categories include "how to use a specific annotation", "how to define a custom annotation", "which annotation to use", "how to resolve the used annotations", "what is the difference between annotation and XML", "what is annotation", and others. Among them, two categories "which annotation to use" and "how to use a specific annotation" are related to the correct usage of annotations in Java programs, accounting for 53.6% of the questions. For example, the questions in the category "which annotation to use" often ask about how to select a suitable annotation from a set of relevant annotations (e.g., `@Service` and `@Repository`); those in the category "how to use a specific annotation" ask the correct way of using a given annotation (e.g., the position where a `@Component` annotation needs to appear).

Based on the above results, we can conclude that more than half of the StackOverflow questions related to Java Annotations ask how to use annotations properly. Therefore, there is a need for recommending proper annotations for developers based on a given code snippet.

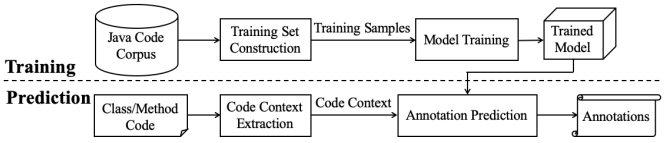

Fig. 4. An Overview of DeepAnna

III. Approach

DeepAnna treats the problem of annotation recommendation and misuse detection as a multi-label classification task. It takes the source code of a class or method as input and produces class- or method-level annotations as output (i.e., recommended annotations). When used for annotation misuse detection, DeepAnna compares the recommended annotations with the annotations currently used by the developers. In this section, we first present an overview of DeepAnna and then detail the steps involved in it.

A. Overview

An overview of DeepAnna is shown in Figure 4. It consists of two phases, i.e., training and prediction.

The training phase trains an annotation prediction model based on a given code corpus. It includes two steps, i.e., training set construction and model training. Training set construction constructs a set of training samples based on the code corpus. It extracts code snippets (i.e., classes/methods) and their annotations from the code corpus and treats each code snippet together with its annotations as a training sample. For each training sample it further extracts structural and textual contexts from the code snippet. Model training trains an annotation prediction model based on the training samples. The model is a deep learning based classification model which predicts the probabilities of all the candidate annotations. Based on the training loss, the model is continuously optimized to maximize the probabilities of the true annotations of

```java
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Service {

    /**
     * The value may indicate a suggestion for a logical component name,
     * to be turned into a Spring bean in case of an autodetected component.
     * @return the suggested component name, if any (or empty String otherwise)
     */
    @AliasFor(annotation = Component.class)
    String value() default "";

}
```

Fig. 5. Annotation Declaration

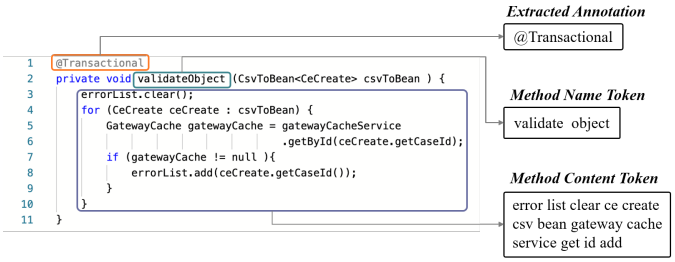

Fig. 6. Extracting Textual Contexts from Code

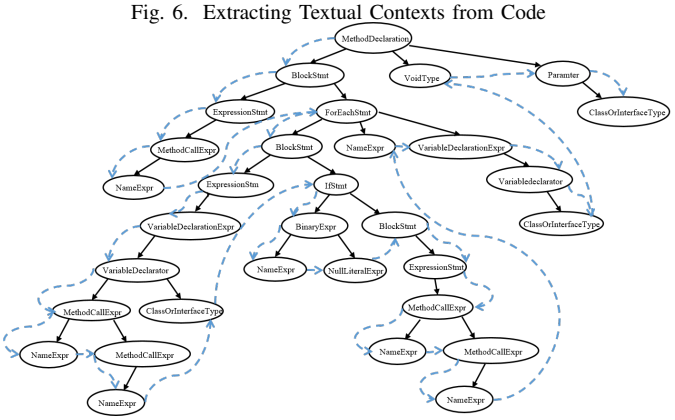

Fig. 7. Extracting Structural Contexts from Code

the code snippet and minimize the probabilities of the false annotations.

The prediction phase recommends a set of annotations for a given class or method. It first extracts structural and textual contexts from the code of the class or method in the same way of model training. It then uses the trained model to predict the annotations for the class or method based on the extracted code contexts. It can recommend the predicted annotations to the developers directly or detect annotation misuses by comparing the predicted annotations with the annotations currently used by the developers.

B. Training Set Construction

Given a Java code corpus, DeepAnna constructs a training set in three steps.

1. Annotation Definition Extraction

DeepAnna supports the annotation recommendation and misuse detection of a given set of Java frameworks. For example, our current implementation of DeepAnna supports the seven major Java frameworks identified in the empirical study. For each framework, DeepAnna extracts all the annotations that are defined by it. The annotations of a framework are usually defined in its source code using the `@interface` keyword. For example, Figure 5 shows the declaration of the `@Service` annotation in the source code of the Spring framework. Therefore, DeepAnna analyzes the source code of each framework and identifies the defined annotations by parsing the annotation declarations using the `@interface` keyword.

2. Training Sample Extraction

DeepAnna parses the Java code corpus and extracts training samples from it. For class-level annotation prediction, DeepAnna extracts classes and their annotations and treats each class together with its annotations as a training sample. For method-level annotation prediction, DeepAnna extracts methods and their annotations and treats each method together with its annotations as a training sample.

3. Code Context Extraction

For each training sample, DeepAnna further extracts code contexts from the code snippet, which will be used to train the model. For each code snippet (class or method), DeepAnna extracts its structural contexts and textual contexts repetitively.

1) Structural Context Extraction.

The structural contexts of a code snippet reflect its control flow and syntactic features. They can be extracted from the abstract syntax tree (AST) of the code snippet. Given a code snippet (class or method), DeepAnna parses it into an AST and removes the subtrees of annotations. It then generates an AST node sequence by traversing the AST by pre-order. For example, for the method shown in Figure 6 DeepAnna extracts an AST and generates a node sequence as shown in Figure 7.

2) Textual Context Extraction.

The textual contexts of a code snippet reflect the latent semantics implied in the identifies of the code. Given a code snippet (class or method), DeepAnna parses the code identifiers in it and extracts a token sequence from the code identifiers. It first extracts a sequence of identifiers from the code by the order of appearance. It then splits each code identifier into tokens by camel case and underscore to generate a token sequence. After that, it further filters out meaningless tokens that are not included in GloVe vocabulary* [5] (which contains more than 400,000 English words from Wikipedia and Gigaword) from the sequence. The resulting token sequence is used as textual contexts of the code snippet. For example, the token sequence extracted for the method in Figure 6 is shown on the right of the figure.

C. Model Training

DeepAnna uses a deep learning based multi-label classification model to predict the annotations of a class or method. The architecture of the model is shown in Figure 8. It includes five components, i.e., structural context encoder, textual context encoder, representation fusion layer, fully connected layer, and

*https://nlp.stanford.edu/projects/glove

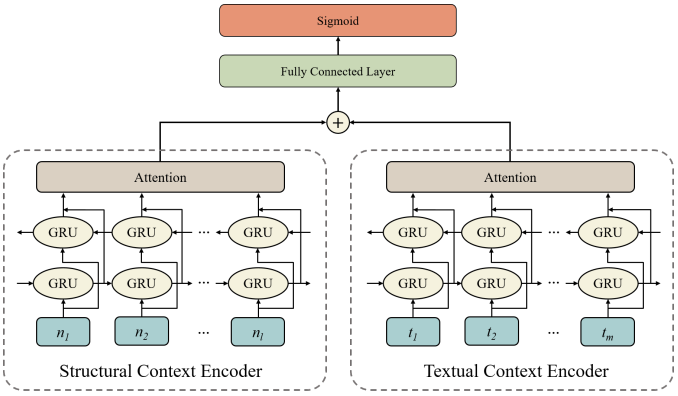

Fig. 8. The Annotation Prediction Model of DeepAnna

sigmoid output layer. The input of the model includes the structural contexts and textual contexts of code snippets, i.e., AST node sequences and token sequences.

For the code snippet of a training sample, its AST node sequence is encoded into a distributed vector representation via the structural context encoder, which includes three layers. The first layer is an embedding layer that embeds each AST node n_j into a dense vector $e_j \in R^d$ through a projection matrix, which is initialized randomly. After the vectors of all the AST nodes are produced, the vector sequence $e_1, e_2, \cdots, e_l$ is fed into a Bi-GRU layer. Formally, the forward hidden state h_j is recursively updated as $h_j = GRU(e_j, h_{j-1})$ and the backward hidden state h'_j is calculated reversely. As the importance of each AST node varies in different contexts, we weight each AST node with an attention layer. After calculating the attention weights and weighted hidden states as Eq 1, the vector representation of the structural contexts of the code snippet z_n is obtained.

$$z_n = \sum_j \alpha_j [h_j, h'_j] \tag{1}$$

where the attention weights are computed as follows:

$$\alpha_j = \frac{\exp(a^T u_j)}{\sum_{j'} \exp(a^T u_{j'})} \tag{2}$$

$$u_j = \tanh(W_s [h_j, h'_j] + b_s) \tag{3}$$

where W_s, b_s and a are learnable parameters.

The textual context encoder for token sequences has a similar structure with the structural context encoder. We denote the output (i.e., vector representation) of the textual context encoder as z_t.

The vector representations of the structural contexts and textual contexts (i.e., z_n and z_t) of a code snippet are then concatenated in the fusion layer, which is followed by a fully connected layer and a sigmoid output layer.

Here we adopt first-order strategy to tackle multi-label learning which decomposes the problem into N independent binary classification problems [6]. We use the Binary Cross-Entropy (BCE) loss function of the predicted probability and the ground truth label of the training sample and use the Adam optimization method to train the model.

TABLE I
THE NUMBER OF REPOSITORIES USED BY DIFFERENT FRAMEWORKS

Framework	Repo.	Framework Description
Hibernate	91	An object relational mapping framework
Jakarta EE	134	Jakarta enterprise edition
Junit	62	Java unit testing framework
Lombok	11	A Java framework to replace template code
Spring	643	An inversion of control framework for Java platform
Guice	20	A lightweight dependency injection framework
Jackson	39	A framework for JSON processing in Java
Total	1,000	–

D. Annotation Prediction

In the prediction phase, DeepAnna extracts the code contexts of a given class or method in the same way as training set construction. It then feeds the extracted code contexts into the trained model and outputs the probabilities of all the candidate annotations. Finally it selects the annotations whose probabilities are greater than a predefined threshold as the recommended annotations for the given class or method.

IV. EVALUATION

We implement DeepAnna to support annotation recommendation and misuse detection for the seven major Java frameworks identified in our empirical study. We implement the deep learning based prediction model using PyTorch 1.9.0[†].

We train the model on a server with Xeon E5-2620 2.1GHz CPUs, 128GB RAM, four Nvidia 1080Ti GPUs, and running Ubuntu 16.04. The settings of the training are as follows: the embedding sizes of AST nodes and tokens are both 32; the hidden size of Bi-GRU is 32; the batch size is 64; the attention size is 16; and the learning rate is 0.005. During the training, we set the training epochs to 10 and take the model with the best F1 score on the validation set as the final trained model.

Based on the implementation, we evaluate DeepAnna with the following three research questions:

- **RQ3:** What is the effectiveness of DeepAnna in annotation recommendation?
- **RQ4:** What is the contribution of different code contexts in DeepAnna to the achieved effectiveness?
- **RQ5:** What is the effectiveness of DeepAnna in annotation misuse detection?

A. Dataset

We crawl 1,000 open-source Java projects that meet the following criteria from GitHub: mentioning one of the seven frameworks in the project descriptions; having at least 50 stars. For the Spring framework we crawl 643 projects that have the most stars. For all the other frameworks we crawl all the projects that meet the two criteria. We randomly divide the crawl projects into training set, validation set, and test set by a ratio of 8:1:1. As there are many classes and methods that do not have any annotations, we ensure that the training samples without annotations account for 1/3 of all the training samples

[†]https://pytorch.org/

by randomly selecting a part of the classes and methods that have no annotations.

Table I reports the number of repositories used by different frameworks, and a brief description of each framework. Specifically, the training, validation and test datasets are constructed based on 800, 100 and 100 repositories respectively. Numerically, we finally obtain large-scale training samples for annotated recommendation and misuse detection, i.e., 139,188 class-level training samples and 762,964 method-level training samples. The validation dataset contains 25,730 class-level samples and 113,019 method-level samples. The test dataset contains 24,489 class-level samples and 124,628 method-level samples.

B. Evaluation Metrics

To evaluate the effectiveness of DeepAnna in annotation recommendation, we use the following eight evaluation metrics for multi-label classification. Before elaborating the metrics, we first introduce some symbols that will be used in the metric definitions. m denotes the number of samples. N denotes the number of labels. $T = \{(x_1, Y_1), (x_2, Y_2), \ldots, (x_m, Y_m)\}$ denotes the test set, where x_i is the i-th sample and Y_i is the label set for x_i. $f(x_i)$ denotes the value function for the i-th sample. A multi-label classification algorithm learns the mapping relation of $f(x_i)$. This function is supposed to have a higher value for those labels which are in the label set Y_i. Y_i^- denotes the set of irrelevant labels for the i-th sample, and Y_i^+ denotes the set of relevant labels for the i-th sample. $rank_f$ denotes the number of labels whose rank is higher than the predicted labels. j denotes a member of the set of Y_i^+.

Hamming Loss. It denotes the proportion of incorrect samples for all labels, and is defined by Eq. 4,

$$Hamming\ Loss = \frac{1}{m}\sum_{i=1}^{m}\frac{XOR(Y_{i,j}, P_{i,j})}{N} \tag{4}$$

where $Y_{i,j}$ is the indicator of class j of sample i, and $P_{i,j}$ is the prediction score of class j of sample i. The smaller the hamming loss, the better the prediction.

One Error. It is used to evaluate the number of times the label ranked in first is not the true label. The smaller the value, the better the prediction. It is defined by Eq. 5.

$$One\ Error = \frac{1}{m}\sum_{i=1}^{m}\mathbb{I}\left[\arg\max f(\boldsymbol{x}_i) \notin Y_i^+\right] \tag{5}$$

Coverage. It denotes the average rank of the furthest true labels among all labels. The smaller the value, the better the prediction. It is defined by Eq. 6.

$$Coverage = \frac{1}{m}\sum_{i=1}^{m}\mathbb{I}\left[\max_{j\in Y_i^+}\mathrm{rank}_f(\boldsymbol{x}_i, j) - 1\right] \tag{6}$$

Ranking Loss. The set of relevant labels is compared with the set of irrelevant labels, and then the number of times the prediction likelihood of the relevant labels is smaller than the prediction likelihood of the irrelevant labels is defined as ranking loss, as formulated in Eq. 7 (i.e., the average score of the pairs of labels in the sample being predicted in the opposite direction). The smaller the value, the better the prediction.

$$Ranking\ Loss = \frac{1}{m}\sum_{i=1}^{m}\frac{\left|\mathcal{S}_{\text{rank}}^i\right|}{\left|Y_i^+\right|\left|Y_i^-\right|} \tag{7}$$

where $\mathcal{S}_{\text{rank}}^i$ denotes the number of pairs of label y_p and y_q where y_p belongs to Y_i^+, y_q belongs to Y_i^-, and the value function of y_p is less than or equal to the value function of y_q.

Average Precision. The average precision is the average accuracy of each point. It is defined by Eq. 8,

$$Average\ Precision = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{\left|Y_i^+\right|}\sum_{j\in Y_i^+}\frac{\left|S_{\text{precision}}^{ij}\right|}{\mathrm{rank}_f(\boldsymbol{x}_i, j)} \tag{8}$$

where $S_{\text{precision}}^{ij}$ denotes the rank of j-th label among all the probability distribution of the ground truth labels for the i-th sample.

Precision, Recall and F1 Score. In addition to the evaluation metrics for multi-label classification, we also use the traditional classification evaluation metrics, i.e., precision, recall and F1 Score. They are defined by Eq. 9, 10 and 11,

$$Precision = \frac{TP}{TP + FP} \tag{9}$$

$$Recall = \frac{TP}{TP + FN} \tag{10}$$

$$F1\ Score = \frac{2TP}{2TP + FP + FN} \tag{11}$$

where TP is the number of annotations that are correct and classified as correct, FN is the number of annotations that are correct but are classified as incorrect, and FP is the number of annotations that are not correct but are classified as correct.

C. Effectiveness Evaluation (RQ3)

We formulate annotation recommendation as a multi-label classification problem. There are other multi-label text classification approaches, which can be applied to this task by taking code snippets as texts directly. Therefore, we compare DeepAnna with several multi-label text classification baselines on annotation recommendation.

1) Method: To evaluate the effectiveness of DeepAnna, we compare it with several state-of-the-art machine learning based and deep learning based multi-label classification approaches. Specifically, we select ML-KNN [7] and ML-ARAM [8] as the state-of-the-art machine learning based approaches, and select Transformer [9] and RCNN [10] as the state-of-the-art deep learning based approaches. Previous studies have demonstrated that RCNN can achieve the best performance on multi-label classification tasks [11]. All models use the same training parameters, training environment and training strategy.

TABLE II
COMPARISON RESULTS OF DIFFERENT APPROACHES ON CLASS-LEVEL ANNOTATION RECOMMENDATION

Metrics	DeepAnna	Transformer	RCNN	ML-KNN	ML-ARAM
Hamming Loss	**0.00014**	0.00018	0.00015	0.00026	0.00035
One Error	**0.13005**	0.16245	0.13965	0.37674	0.37674
Coverage	93.1906	134.6173	**92.9327**	173.2289	361.7455
Ranking Loss	**0.02773**	0.04035	0.02789	0.05134	0.09985
Average Precision	**0.85144**	0.78703	0.83215	0.23774	0.23774
Precision	0.86894	0.85325	**0.86914**	0.76145	0.62326
Recall	**0.80438**	0.70613	0.76833	0.58740	0.49827
F1 Score	**0.83538**	0.77270	0.81560	0.66319	0.55380

TABLE III
COMPARISON RESULTS OF DIFFERENT APPROACHES ON METHOD-LEVEL ANNOTATION RECOMMENDATION

Metrics	DeepAnna	Transformer	RCNN	ML-KNN	ML-ARAM
Hamming Loss	**0.00024**	0.00025	0.00025	0.00029	0.00035
One Error	**0.30190**	0.32161	0.32102	0.43716	0.43680
Coverage	43.5066	67.5626	57.5358	76.5337	**4.41485**
Ranking Loss	0.01311	0.02173	0.01822	0.02444	**0.00115**
Average Precision	**0.72217**	0.71051	0.69141	0.55745	0.36998
Precision	0.74104	**0.77611**	0.73482	0.68971	0.56317
Recall	**0.62066**	0.52212	0.59241	0.46261	0.53865
F1 Score	**0.67550**	0.62121	0.65594	0.55378	0.55065

2) Results: The class-level comparison results are shown in Table II. We can see that DeepAnna achieves the best performance in almost all metrics except for coverage and precision. RCNN achieves the best performance among the four baselines, while the two machine learning based baselines ML-KNN and ML-ARAM have the worst performance. Overall, DeepAnna achieves better performance than RCNN on all metrics except for coverage and precision. Specifically, DeepAnna improves RCNN by about 2% in both average precision and F1 score. Besides, the results of method-level comparison are reported in Table III. DeepAnna achieves the best performance in all metrics except for coverage, ranking loss and precision. Relatively, RCNN is the best among the four baselines, while ML-KNN and ML-ARAM are the worst.

Comparing the evaluation metrics at the class level and method level, we can see that the metrics at the class level are higher than the metrics at the method level. This is because in most cases we can extract more structural context and textual context from class than from method, which allows the model to better learn the context information in the code, thus improving the evaluation metrics.

Summary. DeepAnna can effectively recommend annotations with an average precision of 0.85 and an F1 score of 0.84 at the class level and an average precision of 0.72 and an F1 score of 0.68 at the method level. DeepAnna also outperforms the state-of-the-art multi-label text classification approaches on most metrics at both the class and method level.

D. Ablation Study (RQ4)

Both structural and textual contexts in the code are leveraged in DeepAnna. Therefore, we measure how each code context contributes to the overall recommendation performance.

TABLE IV
COMPARISON RESULTS OF OUR ABLATION STUDY ON CLASS-LEVEL ANNOTATION RECOMMENDATION

Metrics	DeepAnna-Textual	DeepAnna-Structural	DeepAnna
Hamming Loss	**0.00014**	0.00034	**0.00014**
One Error	0.13489	0.30494	**0.13005**
Coverage	95.0783	147.1698	**93.1906**
Ranking Loss	0.02851	0.04408	**0.02773**
Average Precision	0.84253	0.55693	**0.85144**
Precision	0.86818	0.75744	**0.86894**
Recall	0.79162	0.32112	**0.80438**
F1 Score	0.82812	0.45094	**0.83538**

TABLE V
COMPARISON RESULTS OF OUR ABLATION STUDY ON METHOD-LEVEL ANNOTATION RECOMMENDATION

Metrics	DeepAnna-Textual	DeepAnna-Structural	DeepAnna
Hamming Loss	**0.00024**	0.00028	**0.00024**
One Error	0.30965	0.35458	**0.30190**
Coverage	48.5020	56.9429	**43.5066**
Ranking Loss	0.01549	0.01805	**0.01311**
Average Precision	0.70974	0.60820	**0.72217**
Precision	0.73461	0.69156	**0.74105**
Recall	0.61029	0.54166	**0.62066**
F1 Score	0.66666	0.60742	**0.67550**

1) Method: We design an ablation study by removing each code context, i.e., the structural context encoder and the textual context encoder, from the model in DeepAnna. The variant, where the textual context encoder is removed, is referred to as DeepAnna-Structural, while the variant, where the structural context encoder is removed, is referred to as DeepAnna-

Textual. Then, we compare DeepAnna with these two variants with the same training parameters, training environment and training strategy.

2) Results: The comparison results of class-level annotation recommendation are shown in Table IV. We can see that DeepAnna-Textual achieves better performance than DeepAnna-Structural in all metrics, which indicates that textual context is more effective for the annotation recommendation task compared to structural context. DeepAnna, leveraging a combination of textual and structural context, achieves the best performance in all metrics, which indicates that for the annotation recommendation task, we need to consider not only the textual context of the code but also the structural context of the code in order to achieve the best recommendation result. The fusion of the two contexts can effectively improve the overall performance. This also indicates that for class-level annotation recommendation, the GRU model has better information extraction capability for textual context than for structure context.

The comparison results of method-level annotation recommendation are reported in Table V. Similar to the class-level annotation recommendation, DeepAnna-Textual outperforms DeepAnna-Structural in all metrics, which further indicates that textual context is more important than structural context on this task; and DeepAnna achieves the best performance in all metrics. The potential reason for a worse performance of DeepAnna-Structural is that there is much redundant information in ASTs, and most of the nodes in ASTs contain limited syntactic information. If we want to further improve the effectiveness contribution of ASTs, we need to further optimize the extraction method of structural context.

Summary. Both structural and textual context contribute to the achieved effectiveness of DeepAnna, while textual context achieve more contribution than structural context.

E. Annotation Misuse Detection (RQ5)

DeepAnna can automatically recommend annotations for developers to use when programming, which can improve the development productivity of developers by saving the time of query annotations. In addition, DeepAnna can be used to detect annotation misuses in code by analyzing the code, comparing the recommended annotations with the actual annotations in the code, and detecting inconsistencies as annotation misuses. To investigate whether DeepAnna can detect annotation misuses in a real-world scenario, we design the following two experiments.

1) Method: The first experiment is designed to evaluate the effectiveness of annotation misuse detection on real-world annotation misuses. To this end, we first search and crawl all the issues and pull requests in GitHub from November 29, 2020 to May 16, 2021. Then, we identify the issues and pull requests where the programming language is Java, the annotation keyword appears in the title, and the status is closed. By this way, we collect a total of 15,000 issues and pull requests. After that, we keep only the issues and pull requests that are associated with the code commits. Then, we manually review the content of the posts and select the issues and pull requests which fix annotation misuses. Finally, we collect 123 instances of annotation misuses at the class level and 186 instances of annotation misuses at the method level. We extract the code in the commit, use the code before the commit as the code sample (i.e., the input of our model), and use the annotation fixed after the commit as the annotation ground truth, and compare the prediction results of DeepAnna with the annotation ground truth.

TABLE VI
ANNOTATE MISUSE DEETECTION RESULTS

Metrics	Class Level	Method Level
Hamming Loss	0.00040	0.00042
One Error	0.28155	0.61271
Coverage	122.6165	102.6332
Ranking Loss	0.03655	0.03525
Average Precision	0.53579	0.35744
Precision	0.71493	0.44336
Recall	0.47447	0.27554
F1 Score	0.57040	0.33981

The second experiments is designed to evaluate whether DeepAnna can detect new annotation misuses in real-world projects. To this end, we crawl Java repositories with different star intervals from GitHub. Specifically, we crawl 200 projects in descending order of the number of stars from four star intervals, i.e., [50, 100), [100, 200), [200, 300), [300, 500), respectively. Notice that we remove the repositories that belong to the previously constructed dataset in Section IV-A. Then, DeepAnna is applied to detect annotation misuse in these repositories, i.e., the currently used annotation is different from our predicted annotation. When DeepAnna detects an annotation misuse, we send a pull request to the GitHub repository, where the misused annotation is fixed.

2) Results: The result of our first experiment is shown in Table VI. As can be seen from the results, DeepAnna can detect annotation misuses in real scenarios with a precision of 0.71 at the class level and a precision of 0.44 at the method level. These results demonstrate that DeepAnna can detect annotation misuses with moderate accuracy. We can also see that the evaluation metrics of annotation misuse detection are lower than the evaluation metrics in RQ3. This is because the dataset in annotation misuse detection contains annotations that developers will misuse in actual programming scenarios, while the dataset in RQ3 has no consideration of the possibility of annotation misuse actually occurring. These two datasets differ in the distribution of different types of annotations.

For the second experiment, the total number of pull requests we have submitted to GitHub is 85. Among them, 20 pull requests have been accepted by developers. 9 pull requests have been refused, and the developers consider these pull requests as wrong modifications. The remaining 56 pull requests are still in open status and are waiting the developers to give responses. These promising results demonstrate the capability of DeepAnna in detect annotation misuses. Figure 9 shows a practical example of a pull request we have submitted. A

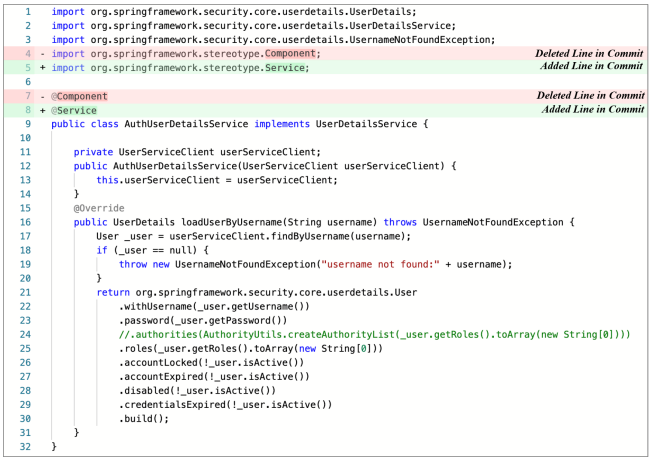

Fig. 9. Pull Request Demo of Annotation Misuse in Github

Spring bean in the service layer should be annotated using `@Service` annotation instead of `@Component` annotation. `@Service` annotation is a specialization of `@Component` in the service layer. By using a specialized annotation, it can serve two purposes. First, annotated classes are treated as Spring bean. Second, special behavior can be put in this layer.

Summary. DeepAnna can detect annotation misuses with moderate accuracy. Based on DeepAnna, we submit 85 bug-fixing pull requests for annotation misuse defects in open-source projects and 20 of them have been accepted and merged.

V. Threats to Validity

The threats to the empirical study mainly lie in the selection and analysis of the annotation related Stack Overflow questions. These questions are selected based on the "java" and "annotations" tags. It is thus possible that Java annotation related questions that have no "java" or "annotations" tags are missed in the analysis. The categories of the problems involved in the questions are analyzed manually and thus may not accurately reflect the intents of the questioner. To alleviate this threat we invite two annotators to classify the questions independently and calculate their agreement on the results.

The threats to the internal validity of the evaluation mainly lie in the selection and quality of the open-source projects used for model training and testing. First, we only consider the Java projects that mention the subject frameworks in their descriptions. Some projects using the seven frameworks are possibly missed as their descriptions do not mention the frameworks. Second, we construct the training set and test set based on the assumption that the annotations are correctly used in these projects. This assumption may not be true for some samples. To alleviate this threat we only choose the projects with at least 50 stars to avoid introducing too much noise. Besides, we apply a dropout layer in the model to avoid overfitting and enhance the model robustness for noisy data. The threats to the external validity of the evaluation mainly lie in the fact that we only consider seven major Java frameworks and the results may not be generalized to the annotations defined by other frameworks.

VI. Related Work

Empirical Studies on Annotation. Many researchers have conducted empirical studies on Java annotation since its introduction as a feature of Java language in 2004. Rocha et al. [12] investigate how annotations are used in 106 open-source projects and find that annotation hell (i.e., poor code readability and understandability caused by the abuse of annotations) occurs in many of the studied projects. Yu et al. [13] present an empirical study on the usage, evolution, and impact of annotations in 1,094 high-quality Java projects on GitHub. They find that annotations are widely used and developer's changes to annotations may cause defects, so they suggest that systematic annotation testing tools are needed to check the use of annotations. Some other researchers conduct empirical studies on specific types of annotations. Tempero et al. [14] conduct an empirical study on the usage of `@Override` annotations in 100 open-source projects and find that most subclasses override at least one method of the parent classes. Dyer et al. [15] present an empirical study on the actual usage of Java language features, including the declaration and use of Java annotations. Parnin et al. [16] empirically study the usage of Java generic types and Java annotations in practice and compare their differences. They find that annotations are more frequently used than generic types in Java projects. Dong et al. [17] find that annotation misuse is common in programs using testing frameworks, so they suggest to conduct researches on annotation recommendation and misuse detection for testing frameworks. These empirical studies motivate our work on Java annotation recommendation and misuse detection in this paper.

Rule Based Annotation Misuse Detection. Some researchers explore the problem of annotation misuse and propose rule-base approaches for annotation misuse detection. Córdoba-Sánchez et al. [18] propose a modelling language called Ann for efficiently designing and verifying Java annotations. Ann uses dependency and integrity constraints between annotations to check whether the annotations used in programs satisfy the constraints. They evaluate the language using a set of annotations from Jakarta EE and the results show that Ann can express sufficiently rich semantic information. Darwin et al. [19] propose an approach that uses DSL-based declarations to verify the correctness of annotation usage. Noguera et al. [20] develop a tool that allows developers to define and verify annotation frameworks using a domain model. Noguera et al. [21] propose a framework for validating procedures for attribute-oriented programming by defining annotation usage rules and an extensible set of meta-annotations. These annotation misuse detection approaches rely on developers' expertise to define framework specific rules or constraints and cannot reflect the latent relationships between the source code and annotations used. In contrast, DeepAnna is a learning based approach that does not rely on developers' expertise and can

reveal the latent relationships between the source code and annotations used.

Deep Learning for Software Engineering. With the rapid development of deep learning in recent years, more and more neural network models are being used in the fields of software engineering, such as code completion and API recommendation, comment generation and code search.

In code completion and API recommendation direction, deep learning models are used to learn representation of source code and recommend next token or API for incomplete code. White et al. [22] point out that deep learning models can learn textual context of code, and propose recurrent neural network (RNN) based code recommendation methods. Dam et al. [23] propose a code completion method using long short term memory (LSTM) instead of basic to solve the long-range dependency problem and thus improve the completion accuracy.

Deep learning based approaches achieve success in comment generation task by modeling the task as a machine translation problem from program language to natural language. Lyer et al. [24] consider source code as token sequences and propose a comment generation model (named CODE-NN) based on Seq2Seq architecture with attention mechanism. Hu et al. [25] consider both textual and structural context and fed them into the Seq2Seq architecture separately to generate higher quality comments.

Deep learning based code search approaches represent the given natural language (NL) query and code snippets into the same vector space and rank the code snippets via their similarity scores. Gu et al. [26] propose CODEnn, a deep learning-based code search method, which uses RNN and multi-layer perceptron for NL and code without considering code structural context. Xu et al. [27] propose a two-stage attentional neural network structure TabCS for code search and their experimental results show that TabCS achieves the best code search results compared to existing approaches.

Deep learning based approaches are proven successful in these research directions related to source code semantics. We also apply representation learning for source code considering structural and textual code contexts and obtain practical results on annotation prediction task.

Text Multi-label Classification. Text multi-labe classification is a classical task in natural language processing and has many application scenarios. Many learning based classification approaches are proposed in recent years. Clare et al. [28] propose a decision tree based multi-label classification algorithm that adapts the traditional decision tree algorithm C4.5 [29] to multi-classification tasks. The prediction results on the genetic dataset show that the authors' proposed method can achieve a high prediction accuracy. Zhang et al. [7] propose a multi-label classification method named ML-KNN based on the traditional K-nearest neighbor (KNN) method. In addition to KNN, ML-KNN leverages the maximum a posteriori (MAP) principle to determine the labels of unseen instances. Read et al. [30] use a new classifier chain approach that models the association between labels and determines final prediction by voting among multiple classifier chain models under an acceptable computational complexity. Benites et al. [8] propose an adaptive associative mapping neural network that clusters the learned prototypes via additional ART layers. This model achieved a significant classification performance and a fast prediction speed. Zhou et al. [11] propose four deep learning-based multi-label classification algorithms, i.e., TagCNN, TagRNN, TagHAN, and TagRCNN, based on CNN [31], RNN [10], HAN [32] and RCNN [33] respectively. Compared with traditional multi-label classification algorithms, experimental results show that TagCNN and TagRCNN achieve better recommendation results. Although these classification models achieved state-of-the-art performance, they are designed for classification task on text instead of source code. Our approach models annotation recommendation and misuse detection as a multi-label classification task by considering the annotations as labels. Besides textual context, we also take structural context of code into consideration to better capture code features.

VII. Conclusion

Annotation is an important feature of Java language and has been widely used in Java programs. Many popular Java frameworks such as Spring highly rely on annotations to define and implement various mechanisms that are required by their applications. In this paper, we present our work on the recommendation and misuse detection of Java annotations. We first conduct an empirical study on Stack Overflow questions to investigate the major development frameworks that are involved in questions about Java annotations and the main problems encountered by developers in the use of Java annotations. The study reveals seven major Java frameworks that are frequently used by developers, for example Spring, Hibernate, Junit. It also shows that a large part of the developer questions about Java annotations are about the correct usage of annotations. Based on the findings of the study, we propose a deep learning based Java annotation recommendation and misuse detection approach called DeepAnna. DeepAnna trains a deep learning based multi-label classification model by considering both the structural and textual contexts of source code. It can recommend annotations and detect annotation misuse at both class level and method level. Our evaluation shows that DeepAnna outperforms state-of-the-art text multi-label classification approaches in annotation recommendation and can effectively detect annotation misuses. Based on our analysis, we submit 85 bug-fixing pull requests for annotation misuses in open-source projects and 20 of them have been accepted and merged.

Our future work will be focused on improving the accuracy of annotation recommendation and misuse detection on the one hand and exploring the application of the approach in industrial projects on the other hand.

Acknowledgment

This work is supported by National Natural Science Foundation of China under Grant No. 61972098.

REFERENCES

[1] (2017) Oracle. lesson: Annotations. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/annotations/

[2] (2021) Spring. [Online]. Available: https://spring.io/

[3] (2021) Hibernate. [Online]. Available: https://hibernate.org/

[4] B. D. Eugenio and M. Glass, "The kappa statistic: A second look," *Comput. Linguistics*, vol. 30, no. 1, pp. 95–101, 2004. [Online]. Available: https://doi.org/10.1162/089120104773633402

[5] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. [Online]. Available: http://www.aclweb.org/anthology/D14-1162

[6] M. Zhang and Z. Zhou, "A review on multi-label learning algorithms," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 8, pp. 1819–1837, 2014. [Online]. Available: https://doi.org/10.1109/TKDE.2013.39

[7] ——, "ML-KNN: A lazy learning approach to multi-label learning," *Pattern Recognit.*, vol. 40, no. 7, pp. 2038–2048, 2007. [Online]. Available: https://doi.org/10.1016/j.patcog.2006.12.019

[8] F. Benites and E. P. Sapozhnikova, "HARAM: A hierarchical ARAM neural network for large-scale text classification," in *IEEE International Conference on Data Mining Workshop, ICDMW 2015, Atlantic City, NJ, USA, November 14-17, 2015*. IEEE Computer Society, 2015, pp. 847–854. [Online]. Available: https://doi.org/10.1109/ICDMW.2015.14

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 5998–6008. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[10] P. Liu, X. Qiu, and X. Huang, "Recurrent neural network for text classification with multi-task learning," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, S. Kambhampati, Ed. IJCAI/AAAI Press, 2016, pp. 2873–2879. [Online]. Available: http://www.ijcai.org/Abstract/16/408

[11] P. Zhou, J. Liu, X. Liu, Z. Yang, and J. C. Grundy, "Is deep learning better than traditional approaches in tag recommendation for software information sites?" *Inf. Softw. Technol.*, vol. 109, pp. 1–13, 2019. [Online]. Available: https://doi.org/10.1016/j.infsof.2019.01.002

[12] H. Rocha and M. T. Valente, "How annotations are used in java: An empirical study," in *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering (SEKE'2011), Eden Roc Renaissance, Miami Beach, USA, July 7-9, 2011*. Knowledge Systems Institute Graduate School, 2011, pp. 426–431.

[13] Z. Yu, C. Bai, L. Seinturier, and M. Monperrus, "Characterizing the usage, evolution and impact of java annotations in practice," *IEEE Trans. Software Eng.*, vol. 47, no. 5, pp. 969–986, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2019.2910516

[14] E. D. Tempero, S. Counsell, and J. Noble, "An empirical study of overriding in open source java," in *Computer Science 2010, Thirty-Third Australasian Computer Science Conference (ACSC 2010), Brisbane, Australia, January 18-22, 2010, Proceedings*, ser. CRPIT, B. Mans and M. Reynolds, Eds., vol. 102. Australian Computer Society, 2010, pp. 3–12. [Online]. Available: http://portal.acm.org/citation.cfm?id=1862200&CFID=15843676&CFTOKEN=50950122

[15] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "Mining billions of AST nodes to study actual and potential usage of java language features," 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. 779–790. [Online]. Available: https://doi.org/10.1145/2568225.2568295

[16] C. Parnin, C. Bird, and E. R. Murphy-Hill, "Adoption and use of java generics," *Empir. Softw. Eng.*, vol. 18, no. 6, pp. 1047–1089, 2013. [Online]. Available: https://doi.org/10.1007/s10664-012-9236-6

[17] D. J. Kim, N. Tsantalis, T. P. Chen, and J. Yang, "Studying test annotation maintenance in the wild," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 62–73. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00019

[18] I. Córdoba-Sánchez and J. de Lara, "Ann: A domain-specific language for the effective design and validation of java annotations," *Comput. Lang. Syst. Struct.*, vol. 45, pp. 164–190, 2016. [Online]. Available: https://doi.org/10.1016/j.cl.2016.02.002

[19] I. F. Darwin, "Annabot: A static verifier for java annotation usage," *Adv. Softw. Eng.*, vol. 2010, pp. 540 547:1–540 547:7, 2010. [Online]. Available: https://doi.org/10.1155/2010/540547

[20] C. Noguera and L. Duchien, "Annotation framework validation using domain models," in *Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings*, ser. Lecture Notes in Computer Science, I. Schieferdecker and A. Hartman, Eds., vol. 5095. Springer, 2008, pp. 48–62. [Online]. Available: https://doi.org/10.1007/978-3-540-69100-6_4

[21] C. Noguera and R. Pawlak, "Aval: an extensible attribute-oriented programming validator for java," *J. Softw. Maintenance Res. Pract.*, vol. 19, no. 4, pp. 253–275, 2007. [Online]. Available: https://doi.org/10.1002/smr.349

[22] M. White, C. Vendome, M. L. Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*, M. D. Penta, M. Pinzger, and R. Robbes, Eds. IEEE Computer Society, 2015, pp. 334–345. [Online]. Available: https://doi.org/10.1109/MSR.2015.38

[23] H. K. Dam, T. Tran, and T. Pham, "A deep language model for software code," *CoRR*, vol. abs/1608.02715, 2016. [Online]. Available: http://arxiv.org/abs/1608.02715

[24] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016. [Online]. Available: https://doi.org/10.18653/v1/p16-1195

[25] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, F. Khomh, C. K. Roy, and J. Siegmund, Eds. ACM, 2018, pp. 200–210. [Online]. Available: https://doi.org/10.1145/3196321.3196334

[26] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 933–944. [Online]. Available: https://doi.org/10.1145/3180155.3180167

[27] L. Xu, H. Yang, C. Liu, J. Shuai, M. Yan, Y. Lei, and Z. Xu, "Two-stage attention-based model for code search with textual and structural features," in *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021*. IEEE, 2021, pp. 342–353. [Online]. Available: https://doi.org/10.1109/SANER50967.2021.00039

[28] A. Clare and R. D. King, "Knowledge discovery in multi-label phenotype data," in *Principles of Data Mining and Knowledge Discovery, 5th European Conference, PKDD 2001, Freiburg, Germany, September 3-5, 2001, Proceedings*, ser. Lecture Notes in Computer Science, L. D. Raedt and A. Siebes, Eds., vol. 2168. Springer, 2001, pp. 42–53. [Online]. Available: https://doi.org/10.1007/3-540-44794-6_4

[29] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[30] J. Read, B. Pfahringer, G. Holmes, and E. Frank, "Classifier chains for multi-label classification," *Mach. Learn.*, vol. 85, no. 3, pp. 333–359, 2011. [Online]. Available: https://doi.org/10.1007/s10994-011-5256-5

[31] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A convolutional neural network for modelling sentences," in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, Volume 1: Long Papers*. The Association for Computer Linguistics, 2014, pp. 655–665. [Online]. Available: https://doi.org/10.3115/v1/p14-1062

[32] Z. Yang, D. Yang, C. Dyer, X. He, A. J. Smola, and E. H. Hovy, "Hierarchical attention networks for document classification," in *NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016*, K. Knight, A. Nenkova, and O. Rambow, Eds. The Association for Computational Linguistics, 2016, pp. 1480–1489. [Online]. Available: https://doi.org/10.18653/v1/n16-1174

[33] P. Zhou, J. Liu, Z. Yang, and G. Zhou, "Scalable tag recommendation for software information sites," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, M. Pinzger, G. Bavota, and A. Marcus, Eds. IEEE Computer Society, 2017, pp. 272–282. [Online]. Available: https://doi.org/10.1109/SANER.2017.7884628