2017 IEEE 25th International Conference on Program Comprehension (ICPC)

Bug Localization with Combination of Deep Learning and Information Retrieval

An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen
Iowa State University, USA
Email: {anlam,anhnt,hoan}@iastate.edu

Tien N. Nguyen
University of Texas at Dallas, USA
Email: tien.n.nguyen@utdallas.edu

***Abstract*—The automated task of locating the potential buggy files in a software project given a bug report is called *bug localization*. Bug localization helps developers focus on crucial files. However, the existing automated bug localization approaches face a key challenge, called *lexical mismatch*. Specifically, the terms used in bug reports to describe a bug are different from the terms and code tokens used in source files. To address that, we present a novel approach that uses *deep neural network* (DNN) in combination with *rVSM, an information retrieval (IR) technique*. rVSM collects the feature on the textual similarity between bug reports and source files. DNN is used to learn to relate the terms in bug reports to potentially different code tokens and terms in source files. Our empirical evaluation on real-world bug reports in the open-source projects shows that DNN and IR complement well to each other to achieve higher bug localization accuracy than individual models. Importantly, our new model, DNNLOC, with a combination of the features built from DNN, rVSM, and project's bug-fixing history, achieves higher accuracy than the state-of-the-art IR and machine learning techniques. In half of the cases, it is correct with just a single suggested file. In 66% of the time, a correct buggy file is in the list of three suggested files. With 5 suggested files, it is correct in almost 70% of the cases.**

***Keywords*-Bug Localization; Deep Learning; Code Retrieval**

I. INTRODUCTION

In software development, engineers spend a great deal of effort in debugging and fixing software bugs. Different stakeholders including end-users, testers, or even developers report defects in the documents called *issue or bug reports*. Those documents contain the descriptions on the scenarios in which the software does not perform as expected. They could also provide other relevant information regarding the reported defects. A bug report will be assigned to a developer who will then investigate and locate the potential buggy files that are relevant to the defects.

The process of localizing the defective files after analyzing the reported bugs is important for developers to quickly and efficiently fix them. However, in a project, developers might have to investigate a large number of source files, keep the logics to connect from the description of the bug(s) in the bug report to the relevant source files, and then leverage the domain knowledge to locate the buggy files. This process is called *bug localization*. This process of bug localization to find and understand the cause of a bug can be time-consuming [1].

To help developers in bug localization process, there exist several automated approaches. They help developers to focus their attention to the potential buggy files relevant to a given bug report. The existing automated bug localization approaches can be broadly divided into two categories that complement to each other in localizing the bugs. The first line of approaches is *fault localization* based on the statistics of *program analysis* information [2], [3], [4], [5], [6]. The program analysis-based approaches focus on the program's semantics and execution. Those approaches rely on analyzing the semantics of the program and/or its execution information such as the execution traces in passing/failing test cases. In contrast, the second line of approaches analyzes the content of a given bug report. They use *information retrieval*(IR) [7], [8], [9], [10] to automatically search for the relevant and potential buggy files by extracting important features from the given bug report and the source code of the project. The popular IR techniques include Latent Semantic Indexing (LSI) [11], Latent Dirichlet Allocation [7], [12], probabilistic ranking with execution scenarios, relevance feedback mechanism, advanced Vector Space Model [10], etc.

Machine learning also is a popular direction of bug localization approaches. BugScout [13] is a special topic model that assumes that the technical topic(s) described the given bug report are also those of the buggy files. The topic model is trained to link the topics of the reports and those of source files. Kim *et al.* [14] apply Naive Bayes using previously fixed files as classification labels and use the trained model to assign source files to each report. Ye *et al.* [9] use a learning approach for adaptive ranking with features from source files, API description, bug-fixing and change history. Unfortunately, *lexical mismatch* [9], [13], [15] between natural language texts in bug reports and technical terms in source code has been identified as the key limitation of those bug localization methods. To bridge the lexical gap, Ye *et al.* [9] additionally use *the texts from the documentation* of the APIs used in the source files. However, API documentation contains texts regarding more general tasks than project-specific buggy behaviors. Kim *et al.* [14] do not use the source files' contents to extract features. They use only the names of the fixed files as classification labels on the reports. Thus, for a new report, they cannot suggest the files that have *not* been fixed before. BugScout [13] uses only one level of abstraction in topics, thus, might not sufficiently link the terms in two spaces of the bug reports and the source files.

In this paper, we develop DNNLOC, a model combining *revised Vector Space Model* (rVSM) [10], an advanced IR technique, with Deep Neural Network (DNN) to recommend the potentially buggy files for a bug report. rVSM extracts the feature to measure *textual similarity* between bug reports and source files. DNN is used to measure the *relevancy* between two sides by learning to relate the terms in bug reports and the potentially different code tokens and terms in source files if they appear frequently enough in the pairs of reports and corresponding buggy files. DNN is expected to help bridge the lexical gap by learning to link high-level, abstract concepts between bug reports and source code. To extract feature for the relevancy between a bug report and a source file, we use DNN as follows. After being projected into the spaces with a smaller number of dimensions, the projected features are fed into an single DNN model with the expectation that it can emphasize the relations of the texts in bug reports and the corresponding features in the buggy source files for bug localization.

For DNNLOC to be scalable, we added into the DNN model an autoencoder [16] with the goal of reducing the dimensions of the feature spaces, while maintaining important features and removing the redundant ones. Each counting feature in the input is projected into a continuous-valued feature space with a smaller number of dimensions. We use the auto-encoder as the projection layer for all features. In addition to the *textual similarity feature* computed from rVSM and the *relevancy feature* from the DNN model, we also integrate another kind of feature, called *metadata features*, extracted from the bug-fixing history. For example, a recently fixed file is more likely to still contain bugs than a file that was last fixed long time ago. Thus, we compute the bug-fixing recency score for a file and use it as a metadata feature. Details of metadata features will be presented in Section IV-D. To combine those three types of features (textual similarity, relevancy, and metadata), we use another DNN as a *non-linear* feature combinator to compute the final score of a file with regard to a bug report.

We have conducted empirical evaluation on several real-world projects. The results show that combining rVSM and DNN achieves higher accuracy than individual models. DNNLOC with the combination of three above types of features improves up to 24.7% at top-1 accuracy over the state-of-the-art approach in Ye *et al.* [9]. DNNLOC achieves up to 50.2% higher top-1 accuracy than the Naive Bayes approach used in Kim *et al.* [14]. We also showed that DNN and rVSM complement well and DNN can link terms in bug reports and different terms and code tokens in relevant source files.

The key contributions of this paper include:

1. DNNLOC, an approach for bug localization combining IR and DNN, and

2. An empirical evaluation on real-world projects to show that the improvement of DNNLOC over the existing approaches.

II. BACKGROUND

This section presents the background on Deep Neural Network (DNN) technology. DNN is a family of algorithms in artificial neural network (ANN) that aim to represent high-level abstractions in data by using model architectures with multiple non-linear transformations [17]. ANN itself is a family of statistical learning algorithms that are used to estimate the functions that can depend on a large number of inputs [18]. The ANNs have been successfully applied in several areas as their architectures have evolved over the years. Advanced architectures such as multi-layer ANNs (*e.g.,* recurrent nets, back-propagation) have been explored. A few key challenges have been preventing the practical applications of such multi-layer form in ANNs [19]. Among them is the high computational cost in training large data [18]. In the mid-2000s, Hinton and Salakhutdinov had showed how a multi-layered feed-forward DNN could be effectively pre-trained one layer at a time. They treat each layer as an unsupervised Restricted Boltzmann Machine (RBM), and use supervised back-propagation for fine-tuning [17]. Since then, DNN has become more successful due to more scalability and efficiency in training. Successful applications of DNN include speech recognition, NLP, and signal/image/text processing, etc [19].

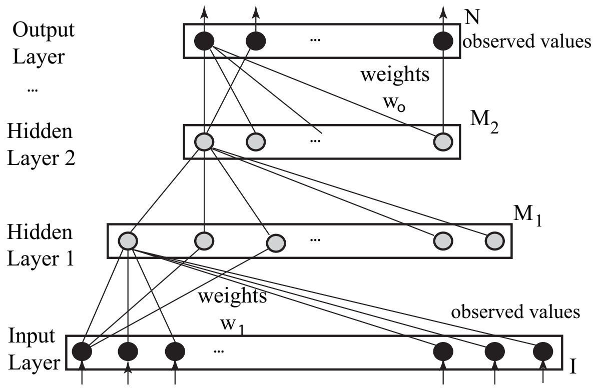

Fig. 1: Deep Neural Network

An RBM-based DNN can be seen as having multiple hidden layers of units between the input and output layers (Fig. 1), where the higher layers enable composition of features from lower layers [17]. The features extracted from the input are fed to the input layer. An DNN might have multiple hidden layers. Mathematically, the output of the the input layer is computed and used as the input for the first hidden layer H_1 [20]:

$$h_{1j} = func\left(\sum_{i=1}^{I} w_1(i,j)I_i + b_{1j}\right), \forall j = 1, \cdots, M_1$$

I_i is the value of the i^{th} node in input layer I; h_{1j} is the output value of the j^{th} node in H_1; $w_1(i,j)$ is the weight of the connection from the i^{th} node in the input to the j^{th} node in the output of H_1; b_{1j} is a bias value for the j^{th} node in the output (i.e., favoring that node); and func is a non-linear function such as a hyperbolic tangent or sigmoid function. The computation for the second layer or subsequence layers is similar.

In Fig. 1, the values at the outputs of H_2 are used to compute the values at the output. An output value is computed as:

$$o_j = \frac{\exp(s_j)}{\sum_{k=1}^{N}\exp(s_k)} \quad where\ s_j = \sum_{i=1}^{M_2} w_o(i,j)h_{2i} + b_{oj}, \forall j = 1, \cdots, N$$

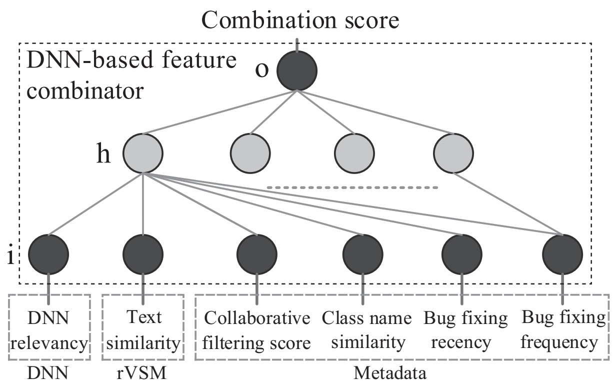

Fig. 2: DNNLoc Bug Localization

o_j is the output value of the j^{th} node in O, which is a normalization of s_j. s_j is the linear combination of all values in the hidden layer H_2, considering the weights $w_o(i,j)$ between those two nodes. Details on the process of training to learn the weights of the layers and predicting can be found in [19].

III. Model Architecture

Fig. 2 shows the architecture of our model. For training, we create a positive pair of a bug report and one of its corresponding buggy files. The pairs are created for all corresponding buggy files and for all bug reports as well. We also create the negative pairs by selecting for every bug report the non-buggy files that are textually similar to the bug reports. For each pair of a bug report and a source file, we extract the features from them, and build the feature vectors. Each vector has its elements being the significance values of features in the feature types. Details of feature extraction and vector building are presented in Section IV. The feature vectors for each type of features are projected into a continuous space with smaller dimensions using the autoencoder implemented via DNN (Section V). The reason for the use of feature reduction via autoencoder [16] is that the number of features extracted from bug reports and source files during training can be very large. The idea is to train the DNN model for projection in which it can encode the original feature vectors in a way that maximizes the similarity between those original vectors and the decoded vectors from the encoded ones (Section V).

All vectors after projection are fed into a DNN to learn the relevancy of each file with respect to a bug report (Section VI). Inspired by the success of DNN in capturing high-level abstractions, we expect DNN to bridge the lexical gap by learning to relate the terms in bug reports to the code tokens or comments in source code that might *not* be textually similar to one another. To compute *the relevancy* between a bug report and a file, we treat the reports as texts and extract the textual tokens. In the source code space, we extract the following features: 1) identifiers, 2) API method calls and classes, 3) comments, and 4) textual descriptions of APIs used in the code. Instead of putting them in the same feature space as in existing

Bug Report 320437
Summary: PageBook toolbars lose the view menu drop down on a page change
Description:
If you start clean then the Outline view will have a view menu drop down but once you open an editor it doesn't...
Switching perspectives will bring it back (likely because we re-render the toolbar, causing the code in the RenderedToolbarRenderer to put it back...perhaps we can change the pagebook's update logic to do the same...

Fig. 3: Bug Report 320437 in Eclipse

IR and ML approaches, in our DNN, we collect the features of different types into *separate spaces* and build the feature vectors (Section VI). We expect that the DNNs can recognize the relations via the weights among the features of different nature in two spaces if they occurred frequently enough in the pairs of bug reports and the corresponding buggy files.

The output of the DNN relevancy estimator, called *relevancy score*, is fed into the DNN-based feature combinator (Section VII), which learns the weights to combine three types of features (relevancy computed from DNN, textual similarity computed via rVSM, and the metadata of the bug-fixing history). The weights of the feature combinator help set the importance of the features in determining the buggy files for a bug report.

We aim to combine DNN (to handle lexical mismatch) and rVSM [10] (an IR model to measure textual similarity) to complement each other. Moreover, we consider the features extracted from the bug-fixing history of a project including the recency and frequencies of bug-fixing files, etc. Instead of using a hill-climbing algorithm for adaptive ranking, we chose to combine the features via another DNN. We expect that with sufficient training data, the weights in the layers reflect the weights of features in the combination. The combination via DNN with non-linear function is expected to perform better than the linear combination in IR-based adaptive learning [19]. DNN will combine 3 types of features (textual similarity, relevancy, and project's metadata).

The DNNs for relevancy estimation and feature combination in our stack-based architecture can be trained independently one layer at a time where each layer is treated as an unsupervised restricted Boltzmann machine. We take advantage of this advance of multi-layer architecture in DNN over regular ANN to scale DNNLoc to large projects. (In an ANN, all the weights must be trained at the same time in the same model).

For prediction, for a given bug report B, we will pair it with each source file f. The features are extracted and feature vectors are built for each pair. Our model produces the final score for f with respect to B. The higher score implies that the model estimates that f is more potentially buggy for B. All the scores for all the source files are used to rank them.

IV. Feature Extraction

This section presents how we extract the features from bug reports and source code. Let us explain that via an example. Fig. 3 shows the bug report #320437 in Eclipse project regarding a bug on drop-down menu as a page changes. A report generally has a short summary and a description regarding the bug. Sometimes, a report also contains execution traces that lead to reported crashes and/or comments on suggested fixes.

```
1 /**
2  This class encapsulates the functionality necessary to manage
3  stacks of parts in a 'lazy loading' manner. For these stacks only
4  the currently 'active' child <b>most</b> be rendered so in this
5  class we over ride that default behavior for processing the
6  stack's contents to prevent all of the contents from being
7  rendered, calling 'childAdded' instead. This not only saves time
8  and SWT resources but is necessary in an IDE world where we must
9  not arbitrarily cause plug-in loading.
10 */
11 public abstract class LazyStackRenderer extends SWTPartRenderer {
12   private EventHandler lazyLoader = new EventHandler() {
13     public void handleEvent(Event event) {
14       Object element =event.getProperty(UIEvents.EventTags.ELEMENT);
15   if  (!( element instanceof MGenericStack<?>))
16     return;
17   MGenericStack<MUIElement> stack = (MGenericStack <MUIElement>)
           element;
18   LazyStackRenderer lsr= (LazyStackRenderer) stack.getRenderer();
19       ...
20 }
```

Fig. 4: Fixed File LazyStackRenderer.java for Report 320437

After examining a reported bug, the developer(s) who was assigned to fix the bug made the fixing changes to the buggy files and committed them. Fig. 4 shows LazyStackRenderer.java, one of the two files that were fixed for the bug report #320437 in Fig. 3. Another file (ActionBars.java) was fixed as well, but is not shown. We proceed to extract the features in the bug reports and the corresponding fixed files as follows.

A. *Extract Textual Features from Bug Reports*

We parse a bug report into words using whitespaces. The punctuation and standard stopwords such as grammatical terms (*e.g.,* "a", "the") are removed. A compound word is split into individual words based on the capital letters in the word if any. For example, "childAdded" is broken into "child" and "added". The words are converted to their stems using the standard Porter stemming program. The program entities in a bug report are detected via an island grammar. Then, the name of an entity is split into individual words using CamelCase or Hungarian notations. For example, the class name RenderedToolbarRenderer is broken into Rendered, Toolbar, and Renderer, which also go through stemming. We also keep the original name. Finally, the term significance weights of all the words are computed using Term Frequency - Inverse Document Frequency (tf-idf) [21]. The weights of the terms in a bug report are used as its features and fed into the projection module (see Section V).

B. *Extract Code and Textual Features from Source Files*

For a source file, we extract four types of features. The first two types of features are code-based and the other two types are textual. The first type of feature is the names of identifiers in a source file. The examples of this type of features in Fig. 4 include lazyLoader, handleEvent, event, element, stack, lsr, LazyStackRenderer, etc. The second type is the names of API classes and interfaces that are used in the source file. For example, the buggy file in Fig. 4 uses the API classes/methods MUIElement, UIEvents, MGenericStack, SWTPartRenderer, getRenderer, etc. The terms in these two types of features (identifiers and API elements) are also split into individual words using CamelCase or Hungarian notations. Then, their term significance weights via tf-idf are computed and then used as code features in our model.

We also extract two other types of textual features in source code. First, we extract the comments and string literals in a source file, *e.g.,* from lines 1–10 of Fig. 4. We process it in the same manner as the texts in the bug reports. Second, we also enhance the features in a source file by taking into account the textual descriptions of the API classes/methods and interfaces that are used in the source file.

C. *Textual Similarity Features*

In DNNLOC, we consider as a feature the *textual similarity* between a bug report and a corresponding source file. To extract such feature, we adopted the revised Vector Space Model (rVSM) from Zhou *et al.* [10] since it has been shown to perform better than the classic VSM. We expect that the textual similarity feature and the relevancy feature (computed from the DNN relevancy estimator) will complement each other in linking bug reports and buggy files in both cases of similar and dissimilar terms. The textual similarity of a bug report B and a file f computed by rVSM is used as the score of this feature for the pair (B, f). For such similarity score, we merge the vocabulary from the reports with that of the source files.

D. *Extract Bug-Fixing, Metadata Features*

We adopted four types of features from bug-fixing history of a project that were proposed in [9]. The first one is how recent a file has been fixed for some bug(s). Several researchers have shown that recently fixed files in the recent past are the ones that are likely to be fixed in the near future [22]. We compute the bug-fixing recency score as follows. Let us call S the set of bug reports that were filed before bug report B and were fixed in file f. Among S, let B' be the report that was most recently fixed. The bug-fixing recency for a pair of a bug report B and a file f is defined as $(B.month - B'.month + 1)^{-1}$. If f was fixed for a report B' in the same month that B was filed, the value is 1. The further in the past f was last fixed, the smaller the bug-fixing recency score.

We also extract the bug-fixing frequency for a source file f. We count the number of times that f was fixed before the bug report B was filed, and use that number as a feature. We also adopted the feature called *collaborative filtering score* [9]. The score aims to measure the similarity of a bug report and previously fixed bug reports by the same file. It is defined as follows. Given a bug report B and a source code file f, the score is the textual similarity (measured by rVSM) between B and all the combined texts in the bug reports that were fixed by f and before B was reported. Finally, we also consider the textual similarity between the names of classes mentioned in a bug report and those in a source file.

V. FEATURE REDUCTION WITH AUTOENCODER

Reducing the dimensions of feature spaces is crucial to make our model scale to real-world data. It helps in both reducing the computational cost for deep learning due to smaller

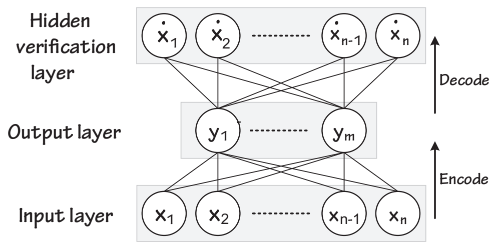
Hidden verification layer
Output layer
Input layer
$\dot{x}_1$ $\dot{x}_2$ $\dot{x}_{n-1}$ $\dot{x}_n$
y_1 y_m
x_1 x_2 x_{n-1} x_n
Decode
Encode

Fig. 5: Dimension Reduction of a Feature Space with DNN

vector sizes and eliminating the redundant information as in LSI [11]. In DnnLoc, inspired by the de-noising technique autoencoder [16], we develop a DNN model for dimension reduction. Fig. 5 shows our dimension reduction DNN. It has three layers, in which the output layer has a smaller dimension than the input layer and the hidden layer is used during training for verifying whether the output is acceptable yet.

The training has two steps via DNN layers. First, the encode step is aimed to project the input feature vectors (as occurrence-count vectors) into a continuous-valued space with a smaller dimension. However, to ensure that the reduced space does not lose information from the original input vectors, we need the second step: decode transforms the vectors in the dimension-reduced space into the vectors in the original occurrence-count space. The training goal is *to maximize the similarity between the original input vectors and the "reconstructed" vectors*.

Specifically, in training, the feature vectors in training dataset are used as the input. Initially, the model with its initial weights operates on each of the input vectors X= $[x_1,x_2,...,x_n]$ to estimate the continuous-valued vectors Y= $[y_1,y_2,...,y_m]$ ($m < n$), which in turn are used to estimate the reconstructed vectors $\dot{X}$= $[\dot{x}_1,...,\dot{x}_n]$. The more the similarity between the original vectors and their respective reconstructed vectors, the better the model is. That criteria is measured via a similarity function. The goal of training process is to perform tuning to the weights in the DNN layers to increase the similarity between the original and reconstructed vectors. The tuning process is done via feed-forward and back-propagation [16].

In predicting, the continuous-valued vectors at *the output layer* are used to represent a bug report or a source file, and fed into the DNN relevancy estimator (see Fig. 2).

While our model has the same goal as LSI, it is different from LSI in two main points. First, LSI uses Singular Vector Decomposition (SVD), an algebra approach of matrix analysis, while we use a deep learning approach. The limitation of LSI is that it cannot scale for a large matrix (*e.g.,* a matrix of size 50,000 examples $\times$ 50,000 features in our experiment). Second, the transformation in LSI is based on linear transformations, while the transformation in this model is based on non-linear functions (*e.g.,*sigmoid, tanh). The latter ones have been shown to be more resilient to noises with various data types [19].

VI. DNN-based Relevancy Estimation

Fig. 6 shows the two DNNs for feature reduction and relevancy estimation between a bug report and a source file. Their feature vectors are fed into the autoencoder for projection (Section V). The DNN estimator takes those inputs, and outputs a relevant score for a bug report and a file.

In the relevancy estimation DNN and combination DNN, all the nodes in a previous layer are connected to all nodes in the next layer via weighted edges. The higher value of a weight, the higher impact of a node in the previous layer to the connected node in the next layer.

For the relevancy estimator, we use the same DNN architecture as the RBM machine in Figure 1 except that we use only 1 hidden layer. The formula to compute the output of the relevancy estimator is the same.

For each DNN at the hidden layer H of this model, we use the standard DNN formula to compute the output value:

$$h'(y) = tanh\left(\sum_{x=1}^{M_1} w_i(x,y)h_i(x) + b_i(y)\right), \forall y = 1,\cdots,M_2$$

Let us use $h_i(x)$ to denote the value of node x at the location i in the input of H, and $h(y)$ is the output value of node y at this layer. $w_i(x,y)$ is the weight showing the impact of the input x on the output y of this layer. $b_i(y)$ is the bias value for calculation at location y. $tanh$ is a common function for the classification problem.

We expect that the DNN for the relevancy estimator would relate the features in bug reports and those in source code, and the features in bug reports and those in comments and API descriptions. The relations are expected to be expressed via the weights from the input features to the nodes in the output of a DNN. For example, "resource", "connection", "session", in a bug report are expected to be strongly related to some hidden nodes that are also impacted much by the nodes representing getCounterRec, ctx, getPermission in the identifier group and by the nodes representing addLocalEjb and sessionExpired in the API group. The output is computed as in a RBM-based DNN. In training, its output is a score of 0 or 1 where 1 indicates relevancy. In predicting, the relevancy score is between [0,1]. The higher the score, the higher the confidence the model estimates the relevancy between a bug report and a file.

VII. Feature Combination via DNN

The relevancy score from the DNN-based relevancy estimator described above is considered as a feature and combined with the textual similarity feature and metadata features (Section IV-D). We use another DNN for feature combination. The non-linear combination by DNN has been shown to improve over the linear feature combination in the learn-to-rank approach [19]. We develop a DNN for feature combination (Fig. 2) with 6 input nodes representing the following features: 1) relevancy score (the output of the relevancy estimator in Fig. 6); 2) the textual similarity feature between a bug report and a source file (see Section IV-C); 3) collaborative filtering score representing the similarity of bug reports (Section IV-D); 4) similarity between the names of entities mentioned in a bug report and a class name in a file; 5) bug-fixing recency score; and 6) bug-fixing frequency score (see Section IV-D). All the scores are scaled and normalized to be within 0–1.

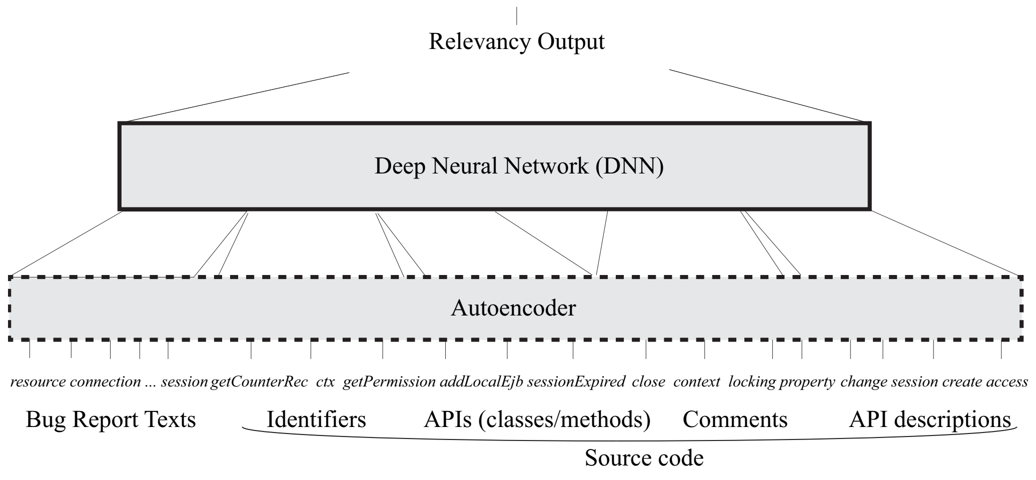
Relevancy Output
Deep Neural Network (DNN)
Autoencoder
resource connection ... session getCounterRec ctx getPermission addLocalEjb sessionExpired close context locking property change session create access
Bug Report Texts
Identifiers
APIs (classes/methods)
Comments
API descriptions
Source code

Fig. 6: DNNs for Autoencoder and Relevancy Estimation

TABLE I
SUBJECT PROJECTS [9]

Project	Time Range	#Bug Reports	#Source Files
AspectJ	03/02-01/14	593	4,439
Birt	06/05-12/13	4,178	6,841
Eclipse UI	10/01-01/14	6,495	3,454
JDT	10/01-01/14	6,274	8,184
SWT	02/02-01/14	4,151	2,056
Tomcat	07/02-01/14	1,056	1,552

The output is a single score indicating the bugginess level of a source file with respect to a bug report. We used the standard DNN training and predicting as explained in Section II.

For ranking, we calculate all relevant scores between a bug report and all the files. The files are ranked according to their scores and the top-ranked files are suggested.

VIII. EMPIRICAL EVALUATION

We conducted several experiments to 1) evaluate DNNLOC's *accuracy* and time efficiency in localizing buggy files from bug reports; 2) study the impacts on accuracy of DNNLOC's *parameters/components* and features; and 3) compare DNNLOC to existing IR and ML approaches for bug localization.

A. Experimental Setting and Metrics

Benchmark DataSet. For comparison, we used the same dataset provided by Ye *et al.* [9] (Table I). All the bug reports, source code links and buggy files, API documentation, and the oracle of bug-to-file mappings are all publicly available at http://dx.doi.org/10.6084/m9.figshare.951967 thanks to the authors.

For comparison, we used the same procedure as in Ye *et al.* [9]. Note that, the number of all source files in a project is very large, making the training time with all negative samples impossible. Thus, for training, they (and we) used text similarity measure to rank all the files, and selected only the top 300 similar files to the bug report as the negative samples. For prediction, we still compute the scores and rank all the files in a project. We sorted the bug reports chronologically by their report timestamps. For all projects except the smallest project AspectJ, we divided the bug reports into 10 folds with equal sizes in which $fold_1$ is the oldest and $fold_{10}$ is the newest. We trained a model on $fold_i$ and tested it on $fold_{i+1}$. For AspectJ, we divided the bug reports into 3 folds since it has smaller number of reports, and used the same testing strategy.

We used three metrics for evaluation. *Top-ranked accuracy* is measured as follows. If one of the buggy files of a given bug report is within the top-*k* list of files, we count it as a hit. Otherwise, we consider that as a miss. The top-*k* accuracy is measured by the percentage of hits over the total number of suggestions in all the folds. To consider the cases of a bug report with multiple buggy files, we also measured Mean Average Precision (*MAP*). It is defined as the mean of the Average Precision (AvgPre) values obtained for all the predictions:

$$MAP = \sum_{b=1}^{|B|} \frac{AvgPre(BR_b)}{|B|},\ AvgPre(BR_b) = \sum_{k \in K} \frac{Pre_k(BR_b)}{|K|}$$

B is the set of all bug reports. K is the set of the positions of the buggy files in the ranked list, as computed by a model. Pre_k is defined as the ratio of the number of actual buggy files in top k over k. We also measured Mean Reciprocal Rank (*MRR*) to compare the models based on the ranked list. MRR is computed as the average of the reciprocal ranks of results for a set of cases: $MRR = \frac{1}{|T_{testset}|} \sum_{i=1}^{|T_{testset}|} \frac{1}{index_i}$. where $index_i$ is the index of the highest-ranked actual buggy file in the list at the i-th suggestion, and $|T_{testset}|$ is the number of cases. The values of MRR and MAP are in [0–1]. The higher they are the better the overall ranking of actual buggy files from a model.

B. Impacts of Components and Parameters on Accuracy

In this experiment, we evaluated the impact of different components and parameters of DNNLOC on its accuracy. We chose Tomcat, one of the smaller subject systems in our dataset.

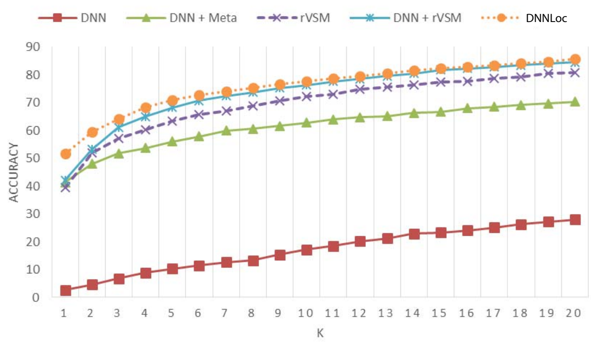

Fig. 7: Top-*k* Accuracy with Different Components

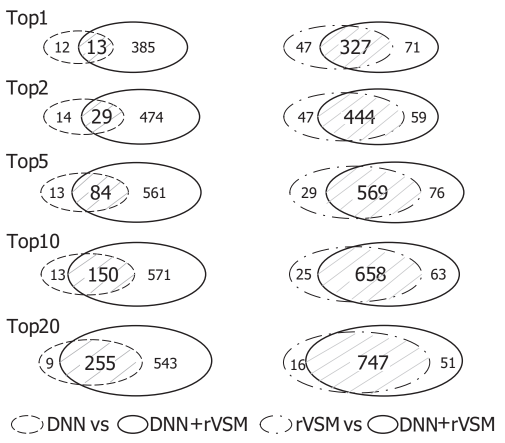

Fig. 8: Venn Diagrams for Correct Results of Approaches

1) Accuracy with Different Components: In this experiment, we varied different components and compared the accuracy of newly configured models to learn their impacts on accuracy. Fig. 7 shows the result. The lines marked with DNN shows for the accuracy of the model using only DNN to compute bug-report-to-file relevancy as the sole feature. As seen, DNN by itself does not give high accuracy. Investigating further, we see that due to projection to a new space with smaller dimensions, more source files with the same technical topics are included but they are not buggy with respect to a given bug report. When using the relevancy feature via DNN and the bug-fixing metadata features (line DNN+Meta), the accuracy is much improved (top-1 accuracy from 3% to 41%, and top-5 accuracy from 10% to 56%). The model using text similarity (rVSM) is equivalent to BugLocator [10] (*i.e.,* without using DNN and bug-fixing metadata features). The top-ranked accuracy ranges from 36–71%. However, when we combined relevancy feature computed by DNN and textual similarity feature computed by rVSM (see line DNN+rVSM), the accuracy is higher than those of both DNN and rVSM individually. The absolute improvement values over DNN are from 39–60% for top-ranked accuracy. The corresponding improvement values over rVSM are from 1–5% (relative improvement from 2.5–8%).

To study further the results from the DNN, rVSM, and DNN+rVSM models, we drew their venn diagrams (Fig. 8). Our goal is to evaluate how much overlapping between the correct result from the combining model DNN+rVSM and those from the individual models DNN and rVSM. We computed such overlaps for different top-ranked resulting lists (with 1,2,5,10, and 20 files). That is, if a buggy file was in a top-ranked list of a model, we consider that it is a correct case and vice versa. The left column shows the overlapping between the results of the model DNN and the combining model DNN+rVSM. For example, at the first diagram on the top left corner of Fig. 8, for a top-1 list, in a total of 410 correct cases, DNN+rVSM is correct in 13+385=398 cases. Only 12 of them were correctly identified by DNN but not by DNN+rVSM, and 385 of them were correctly identified by DNN+rVSM, but not by DNN. In general, across all top ranks (1–20), the combining model covers a good percentage (52–96%) of the results by DNN, while it correctly identified many more results that DNN did not.

The second column of Fig. 8 displays the overlapping between the correct result of rVSM and that of the combining model DNN+rVSM. It is consistent across all top-ranked accuracy that the correct result of DNN+rVSM covers a large percentage (88–98%) of that of rVSM. Moreover, the correct results of (DNN+rVSM $\setminus$ rVSM) are multiple times (1.3–3.2x) more than those of (rVSM $\setminus$ DNN+rVSM) (*i.e.,* 2.1–6.9% of all total results). Interestingly, there are 18 cases in which both DNN and rVSM does not put the file in the top-20 list, but DNN+rVSM does.

Finally, as seen in the line DNNLOC of Fig. 8, DNNLOC achieves highest accuracy with the combination of relevancy via DNN, textual similarity via rVSM, and the metadata features. With a single suggestion, it has the correct buggy file in almost 54% of the cases. Two out of three cases, a correct buggy file is in the list of 3 suggested files. With 5 suggested files, it is correct in almost 75% of the cases.

2) Example: We looked closely on the cases where DNN contributes to improve accuracy while the IR model rVSM does not put the actual buggy files in a top-ranked list. They correspond to the cases in (DNN+rVSM $\setminus$ rVSM) (Fig. 8) and/or DNN ranks the actual files higher in the resulting list than rVSM. We found a common phenomenon in those cases that reporters described in the bug reports the scenario(s) leading to failure or unexpected behaviors using texts without many code tokens. In those cases, the buggy files have few comments and do not contain many identifiers and terms that appear also in bug reports. Fig. 9 shows the bug report #25060 from Tomcat. A user described the bug in which the system incorrectly opened jndi data source connections when it reloads an abandoned context. Among all 250 words in the report (excluding stopwords), there are only 7 terms (context, jndi, exceptions, reload, development, environment, and production) that appeared in the buggy file StandardContext.java (not shown). The texts in the report and the file are not sufficiently similar and rVSM did not rank it in the top-5 list, while both DNN and the combining model DNN+rVSM ranked it as the first place.

Bug Report 25060
Summary: Reloading context orphans currently open jndi datasource connections
Description: ... I have a jndi datasource connected to a postgresql server. I have two jndi resources (a reader and writer) so that later I can implement a system with replication etc. To summarize, after using the system, there are two connections to postgres that get reused: one reader and writer... It appears that after a reload, the 'persisted connections' get abandoned/orphaned. Eventually, I hit my max connections and cannot aquire any more and the system fails. I have tried the abandond collection parameters and have added logging to my code to ensure that I am calling close on the connections I checkout, even on exceptions and error cases. Under normal useage without reloads, no connection leakage happens...

Fig. 9: Bug Report 25060 in Tomcat

TABLE II
LINKING TERMS IN BUG REPORTS AND TERMS/TOKENS IN SOURCE FILES

Bug Report	Token 1	Token 2	Token 3	Token 4
context	authorization	ctx	envCtx	asyncContext
resource	virtualClasspath	changeSessID	setSecureClass	addLocalEjb
writer	globalCacheSize	charset	index	charsWritten
read	headerLength	InternalBuffer	readBytes	dir

3) Examples of Linking Terms in Two Spaces: In NLP, researchers have shown that DNN is able to project and link the words that are semantically or grammatically related into nearby locations (at least along some dimensions) in a continuous-valued feature space [19]. In this experiment, we study the examples in which DNNLOC with its DNN machinery is able to learn the relations between the terms in the bug reports and the terms/tokens in the relevant source files.

We conducted an experiment on the DNN model in Fig. 6 after training. We used each of the textual tokens in all bug reports as the input. We paired it with each of the code tokens in source files and the textual tokens in comments/documentations. We used that DNN to compute the score output for each of such pairs. Finally, for each textual token in a bug report, we produced the list of "relevant" tokens in source files ranked by the relevancy score between two tokens. Table II displays a few examples of "related" terms in the bug report 25060 with top scores computed by the DNN model. As seen, despite using different lexical terms (e.g., context versus ctx and envCtx), DNN is able to relate them in two spaces. Interestingly, the term resource does not appear in the buggy file StandardContext.java, however, DNN can relate it to the terms in that file such as virtualClasspath, changeSessID, setSecureClass, etc. since they appear frequently in the pairs of bug reports and buggy source files. Thus, DNN is able locate and rank that buggy file highest.

4) Accuracy when Varying Size of DNNs of the Estimator: Fig. 10 shows the accuracy when we vary the number of hidden nodes M in the DNN model in DNNLOC. The shapes of the graphs for all top-ranked accuracy are consistent. The accuracy values do not change much as the number of hidden nodes increases. Overfitting will occur when M is comparable to the number of inputs. In later experiments, we tuned M between 1,000–1,100 since the accuracy is slightly better.

As the number of nodes in the hidden layer M is small (M<200), accuracy is low. This is reasonable because the number of dimensions in the new space might be too small to distinguish a large number of inputs representing the input entities. As M increases, accuracy gradually increases. When

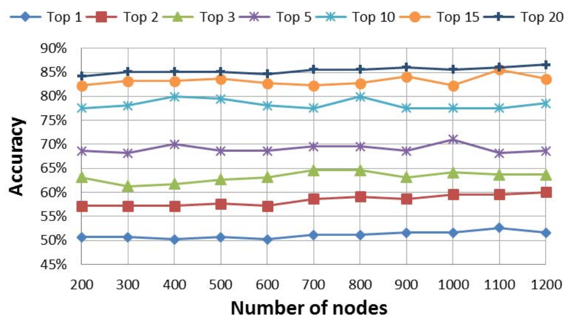

Fig. 10: Top-k Accuracy with Varied Numbers of Hidden Nodes

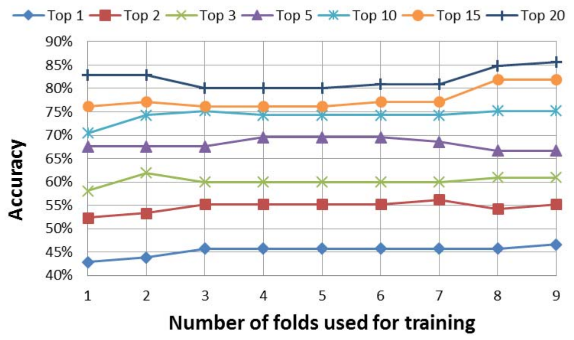

Fig. 11: Top-k Accuracy with Varied Training Data Sizes

M is larger than or equal to 1,000, accuracy is more stable. This suggests that around that number, we could get high, stable accuracy. The number of dimensions in these ranges now might provide sufficiently fine granularity to distinguish the inputs for this project.

5) Accuracy with Different Training Data Sizes: We conducted an experiment to evaluate the impact of training data sizes on DNNLOC's accuracy. We used data in $fold_{10}$ for testing. We first used $fold_9$ and then increased the number of folds including $fold_8$ and so on as training data. Finally, we used all 9 folds from 1–9 for training.

Fig. 11 shows the accuracy corresponding to different numbers of folds used in training. As seen, accuracy does not increase much as more folds are used in training. This result is reasonable due to the time locality of bug fixing on the same files. That is, the files that were recently fixed for a bug report is likely to be fixed again in a near-future bug report. Thus, the fixed files for much older bug reports are not as useful for newer bug reports as the recent fixed files and reports. Moreover, this allows us to achieve near optimal accuracy using only a relatively small training dataset. This has beneficial impact on time and memory complexity.

C. Accuracy Comparison

Our next experiment aims to compare DNNLOC to the state-of-the-art approaches including the Naive Bayes (NB) approach by Kim *et al.* [14], the LR (learn-to-rank) approach by Ye *et*

TABLE III
TOP-*k* ACCURACY COMPARISON WITH LR [9], BL [10], AND NB [14]

Project	Model	1	2	3	4	5	10	15	MRR	MAP
TomCat	DNNLOC	53.9	64.6	67.3	71.8	72.9	80.4	85.9	0.60	0.52
	LR [9]	46.2	54.2	59.8	62.3	66.5	74.7	80.1	0.55	0.49
	BL [10]	35.5	48.7	52.9	58.7	61.8	71.1	77.3	0.48	0.43
	NB [14]	5.2	6.9	8.3	8.8	9.0	11.9	14.5	0.08	0.07
AspectJ	DNNLOC	47.8	56.7	65.3	66.3	71.2	85.0	86.2	0.52	0.32
	LR [9]	20.2	32.1	38.5	41.3	45.5	61.1	68.2	0.33	0.25
	BL [10]	20.1	30.5	40.1	43.3	47.7	57.0	62.1	0.32	0.22
	NB [14]	4.2	8.0	11.3	11.7	16.0	21.1	26.3	0.10	0.07
Birt	DNNLOC	25.2	30.4	34.7	38.4	42.2	50.9	57.2	0.28	0.20
	LR [9]	12.4	18.1	22.5	25.1	27.9	37.3	42.4	0.20	0.15
	BL [10]	11.1	16.2	20.0	22.4	24.9	32.1	37.0	0.18	0.14
	NB [14]	2.9	4.7	6.5	7.9	8.7	13.8	15.9	0.06	0.05
Eclipse	DNNLOC	45.8	55.9	61.8	67.4	70.5	78.2	83.3	0.51	0.41
	LR [9]	36.5	47.0	52.0	58.0	60.1	70.7	75.3	0.47	0.40
	BL [10]	26.5	34.9	40.3	44.8	49.3	60.1	67.3	0.37	0.31
	NB [14]	3.8	6.1	8.3	9.6	10.6	14.7	16.8	0.07	0.06
JDT	DNNLOC	40.3	49.1	57.2	61.6	65.0	74.3	79.4	0.45	0.34
	LR [9]	30.0	40.3	48.2	51.1	55.2	68.1	72.4	0.42	0.34
	BL [10]	19.1	26.2	31.6	37.4	40.2	51.2	57.7	0.30	0.23
	NB [14]	6.6	9.7	11.8	13.6	15.0	20.0	22.9	0.11	0.08
SWT	DNNLOC	35.2	49.2	58.4	63.6	69.0	80.3	85.3	0.45	0.37
	LR [9]	28.3	39.4	47.9	52.7	58.2	70.0	76.8	0.41	0.36
	BL [10]	19.3	24.5	30.0	34.4	38.3	51.1	58.5	0.28	0.25
	NB [14]	7.4	11.8	14.9	17.0	19.0	26.9	31.3	0.14	0.11

al. [9], and BugLocator by Zhou *et al.* [10]. While NB is an ML approach, BugLocator is IR-based, and LR is a hybrid one. We used the same dataset provided by Ye *et al.* [9] and conducted our experiment in the same procedure and metrics. Therefore, we used the results reported in the paper by Ye *et al.* [9] for comparison. In their paper, they also reported the result from running BugLocator on the same dataset. Thus, we also used that result for BugLocator. For the NB approach in Kim *et al.* [14] (which is not publicly available), we followed their description for re-implementation. Their model has two phases. It first classifies if a bug report is "predictable" or "deficient", and then predicts only for the cases of "predictable" bug reports. Accuracy is computed as the ratio of the correctly detected cases over the total number of predictable cases.

Table III shows the accuracy comparison for different approaches. First, as seen, in comparison with BugLocator [10], at top-1 accuracy, DNNLOC achieves from 12–25.8% higher. At top-5 accuracy, the improvement is from 13.2–28.3%. As explained in Section VIII-B1, the relevancy feature computed by the DNN model helps to complement the textual similarity feature computed by rVSM in BugLocator to bridge the lexical gap between bug reports and source files. Thus, DNNLOC with an additional feature of project's metadata achieves even higher accuracy than BugLocator.

Second, in comparison with the learn-to-rank (LR) approach [9], at top-1 accuracy, DNNLOC achieves from 6.7–25.6% higher. The key to bridge the lexical gap in the LR approach is to add to the feature set the terms in the API documentation for the APIs used in the source files. We found that such addition is not effective in the cases of popular APIs such as java.util in Java Development Kit (JDK). Those APIs do not help in linking a bug report to source code because the report describes a more specific erroneous functionality in a system.

TABLE IV
TRAINING AND PREDICTING TIME IN MINUTES

	Training for one fold		Predicting for one report	
System	Max	Average	Max	Average
Tomcat	70	65	1.5	1.0
AspectJ	70	70	4.1	2.4
Birt	98	84	4.5	3.1
Eclipse	120	90	3.8	2.1
JDT	122	94	4.8	3.3
SWT	95	83	2.4	1.8

In contrast, the DNN-based relevancy estimator can handle the lexical mismatch in those cases since it connects a report to a buggy file if the terms appear frequently enough in the pairs of bug reports and corresponding buggy files.

Third, we found that DNNLOC is able to correctly suggest several cases where the NB approach in Kim *et al.* [14] missed because their approach has not seen the fixed files before. The key reason is that their approach uses the previously fixed files as labels for training, but does not consider their contents. At top-1 accuracy, DNNLOC achieves from 20.2–52.1% higher.

As seen, DNNLOC also consistently achieves highest accuracy in MRR and MAP. Its MRR values are from 0.28–0.6. That is, in the best scenario, among 3 cases, it would rank an actual buggy file at the 2nd place in two cases, and likely rank another actual buggy file as the top candidate in the other case.

Feature combination comparison. To compare feature combination by DNN and by learn-to-rank approach [9], we built another experimental model in which all three types of features in our model is combined via learn-to-rank, instead of DNN. We then compared the accuracy of that newly built model (called LRCombine) and the model where all features are combined via DNN. Our result shows that at top-1 accuracy, with DNN, our model achieves higher than LRCombine 26%, while the improvement at top-5 is 13%. This shows that the non-linear combination of those features is better than the linear combination via learn-to-rank for bug localization.

D. Time Efficiency

Table IV shows DNNLOC's training and predicting time. All experiments were run on a computer with CPU Intel Xeon CPU E5-2650 2.00GHz (32 cores), 126 GB RAM. Since we need only one fold for training to predict the next fold, we measured the training time for one fold. We measured the predicting time for individual bug reports. As expected, training time is large for a solution involving one thread to run DNN. However, we could investigate parallel computing infrastructures for DNNs or the incremental training techniques to update the model after getting more bug reports and fixed files over time. Predicting time is reasonable (within a few minutes for one bug report).

Threats to Validity. The subject projects might not be representative. For comparison, we ran all approaches on the same dataset. We do not use the existing tools since they are not available. However, we used the same dataset and follow the same procedure, thus, we used the results reported in their papers. Term splitting from comments could affect accuracy.

IX. Related Work

DnnLoc is related to the learn-to-rank (LR) approach in Ye *et al.* [9]. Ye *et al.* [9] is an IR approach but using adaptive learning to derive the weights of the features extracted from source files, API descriptions, bug-fixing history. Despite using a similar set of features, DnnLoc has key differences. First, DnnLoc is a combination approach between DNN and IR. LR treats source code as texts in the same space as those from bug reports. In DnnLoc, features in bug reports and source files are handled in different spaces and related to one another via DNN. Second, LR bridges the lexical gap by using the terms in the descriptions of the APIs used in source files. DnnLoc uses DNN to bridge that gap. Third, we use DNNs for learning the weights, rather than using adaptive learning. Finally, our projection layer helps to reduce the feature dimensions.

A line of approaches for bug localization related to DnnLoc is based on *machine learning*. Kim *et al.* [14] extract the features from the texts of the bug reports and their metadata including version, platform, priority, and use Naive Bayes for training with previously fixed files as classification labels. For prediction, their trained model assigns source files to a given bug report. They propose a two-phase model in which they first classify a new bug report as either "predictable" or "deficient", and then predict only for the former type. In comparison, DnnLoc is a combination between ML and IR. We use deep learning to learn the connections between the terms in bug reports and source files. Their approach with Naive Bayes uses the previously fixed files as labels for training, thus, cannot suggest files that *have not been fixed before* for a new report. Finally, while DnnLoc treats different features in different spaces and uses DNN to learn the connections, Kim *et al.* use the fixed files as labels *without using the files' contents*.

BugScout [13] is a topic modeling approach for bug localization. BugScout assumes that the topics in a bug report overlap with those in the corresponding buggy files. It uses topic modeling to bridge the lexical gap, rather than DNN.

There are several *IR-based approaches* for bug localization [7], [8], [9], [10]. BugLocator [10] computes textual similarity of bug reports and source files with an advanced VSM model, rVSM. We use rVSM in combining with DNN. BugLocator also considers the similarity between bug reports and use fixed files for similar reports. Other common IR-based approaches in the context of feature/concept location include Latent Semantic Indexing (LSI) [11], and VSM [23]. Latent Dirichlet Allocation [7], [12] was also used for comparing the topics to connect software artifacts. DebugAdvisor [24] enables users to search with both structured and unstructured data for bugs. It builds a graph connecting assignees, related files and functions to the bugs. Users' feedbacks are also used to improve accuracy [25]. A comparative study was conducted on IR-based bug localization methods [8]. In general, despite their successes, the common issue of those IR-based bug localization solutions is lexical mismatch. DnnLoc uses deep learning to address that. Other sources of information have been used to improve bug localization accuracy. The information on stack-trace [26] and source file segmentation [27] is also used to improve bug localization performance. Saha *et al.* [28], [29] use structured information to integrate into IR to improve accuracy in bug localization.

The above IR and ML approaches help developers to localize the files that need to be examined. After that, the *program analysis*-based approaches can be applied to further localize the erroneous methods and lines of code. For such fine-grained localization, they focus on the program's semantics and the execution information, rather than on the analysis of the description of the bugs in a report. The approaches based on dynamic analysis include statistical debugging [6], [5], statistic analysis on passing/failing test runs [3], [4], [30], change impacts in dynamic call-graphs [31], [32], machine learning on dynamic properties of execution [33], etc. Static analysis is also popularly used in fine-grained fault localization including program slicing [34], postmortem symbolic evaluation [35], change impact analysis [36]. There is a rich literature in *bug prediction research based on mining software repositories* [37], [38], [39], [40]. They focus more on predicting error-prone code entities, rather than localizing a fault.

In this work, we extend our previous model in Lam *et al.* [41]. We used a single DNN-based auto-encoder for feature reduction to make the model scalable, instead of using the buit-in Word2Vec projection layer in deeplearning4j [42] as in [41]. The DNN model for relevance estimation (Fig. 2) allows us to achieve better accuracy. The experiments on sensitivity over different components, data sizes, and the nodes in the hidden layers of the DNN model (Section VIII) showed us that DnnLoc still achieves better accuracy.

Several other approaches have applied deep neural network language models and NLP techniques in software engineering applications including code completion [43], [44], API call suggestion [45], [46], [47], language migration [48], code synthesis [49], [50].

X. Conclusion

This paper presents a combining approach between *rVSM, an information retrieval (IR) technique*, and *deep neural network* (DNN) in which DNN is used to learn to connect the concrete terms in bug reports to the code tokens and terms in source files. DNN by itself does not achieve high accuracy due to the projection to a space with a smaller number of dimensions. However, when combining DNN with rVSM, DNN is able to link the bug reports and relevant buggy files that are not textually similar, but are "nearby" after the projection. Our empirical evaluation on several real-world projects shows that DNN and IR complement well to each other to achieve higher bug localization accuracy than individual models. Finally, our new model, DnnLoc, achieves higher accuracy than state-of-the-art IR and machine learning techniques in bug localization.

XI. Acknowledgments

This work was supported in part by the US National Science Foundation (NSF) grants CCF-1723215, CCF-1723432, TWC-1723198, CCF-1518897, and CNS-1513263.

REFERENCES

[1] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, “The design of bug fixes,” in *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*. IEEE, May 2013.

[2] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, “A practical evaluation of spectrum-based fault localization,” *J. Syst. Softw.*, vol. 82, no. 11, pp. 1780–1792, Nov. 2009.

[3] J. A. Jones and M. J. Harrold, “Empirical evaluation of the tarantula automatic fault-localization technique,” in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’05. ACM, 2005, pp. 273–282.

[4] J. A. Jones, M. J. Harrold, and J. Stasko, “Visualization of test information to assist fault localization,” in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE ’02. ACM, 2002, pp. 467–477.

[5] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, “Scalable statistical bug isolation,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’05. ACM, 2005, pp. 15–26.

[6] C. Liu, L. Fei, X. Yan, J. Han, S. Member, and S. P. Midkiff, “Statistical debugging: A hypothesis testing-based approach,” *IEEE Transaction on Software Engineering*, vol. 32, pp. 831–848, 2006.

[7] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, “Bug localization using latent dirichlet allocation,” *Inf. Softw. Technol.*, vol. 52, no. 9, pp. 972–990, Sep. 2010.

[8] S. Rao and A. Kak, “Retrieval from software libraries for bug localization: A comparative study of generic and composite text models,” in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR ’11. ACM, 2011, pp. 43–52.

[9] X. Ye, R. Bunescu, and C. Liu, “Learning to rank relevant files for bug reports using domain knowledge,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. ACM, 2014, pp. 689–699.

[10] J. Zhou, H. Zhang, and D. Lo, “Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports,” in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE ’12. IEEE Press, 2012, pp. 14–24.

[11] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, “Indexing by latent semantic analysis,” *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*, vol. 41, no. 6, pp. 391–407, 1990.

[12] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, “Software traceability with topic modeling,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ser. ICSE’10. ACM, 2010, pp. 95–104.

[13] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen, “A topic-based approach for narrowing the search space of buggy files from a bug report,” in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE’11. IEEE CS, 2011, pp. 263–272.

[14] D. Kim, Y. Tao, S. Kim, and A. Zeller, “Where should we fix this bug? a two-phase recommendation model,” *IEEE Transactions on Software Engineering*, vol. 39, no. 11, pp. 1597–1610, 2013.

[15] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, “What makes a good bug report?” in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT ’08/FSE-16. ACM, 2008, pp. 308–318.

[16] Y. Bengio, *Foundations and Trends in Machine Learning - Learning Deep Architectures for AI*. NOW, the essence of knowledge, 2009.

[17] http://en.wikipedia.org/wiki/Deep_learning.

[18] http://en.wikipedia.org/wiki/Artificial_neural_network.

[19] L. Deng and D. Yu, *Deep Learning Methods and Applications – Foundations and trends in signal processing*. USA: NOW, 2014.

[20] E. Arisoy, T. N. Sainath, B. Kingsbury, and B. Ramabhadran, “Deep neural network language models,” in *Proceedings of the NAACL-HLT 2012 Workshop: Will We Ever Really Replace the N-gram Model? On the Future of Language Modeling for HLT*, ser. WLM ’12. Association for Computational Linguistics, 2012, pp. 20–28.

[21] http://en.wikipedia.org/wiki/Tf-idf.

[22] S. Kim, T. Zimmermann, E. J. Whitehead, and A. Zeller, “Predicting faults from cached history,” in *Proceedings of the 29th international conference on Software Engineering (ICSE’07)*. IEEE CS, 2007, pp. 489–498.

[23] S. Wang, D. Lo, and J. Lawall, “Compositional vector space models for improved bug localization,” in *Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME’14)*. IEEE CS, 2014.

[24] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala, “Debugadvisor: A recommender system for debugging,” in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE ’09. ACM, 2009, pp. 373–382.

[25] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, “Using natural language program analysis to locate and understand action-oriented concerns,” in *Proceedings of the 6th International Conference on Aspect-oriented Software Development*, ser. AOSD ’07. ACM, 2007, pp. 212–224.

[26] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen, “On the use of stack traces to improve text retrieval-based bug localization,” in *IEEE International Conference on Software Maintenance and Evolution (ICSME’14)*. IEEE CS, 2014.

[27] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, “Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis,” in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME’14)*. IEEE CS, 2014, pp. 181–190.

[28] R. Saha, M. Lease, S. Khurshid, and D. Perry, “Improving bug localization using structured information retrieval,” in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE’13)*. IEEE CS, 2013, pp. 345–355.

[29] R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry, “On the effectiveness of information retrieval based bug localization for c programs,” in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME’14)*. IEEE CS, 2014.

[30] H. Cleve and A. Zeller, “Locating causes of program failures,” in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE ’05. ACM, 2005, pp. 342–351.

[31] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, “Chianti: A tool for change impact analysis of java programs,” in *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA ’04. ACM, 2004, pp. 432–448.

[32] O. C. Chesley, X. Ren, B. G. Ryder, and F. Tip, “Crisp–a fault localization tool for java programs,” in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE ’07. IEEE Computer Society, 2007, pp. 775–779.

[33] Y. Brun and M. D. Ernst, “Finding latent code errors via machine learning over program executions,” in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE ’04. IEEE Computer Society, 2004, pp. 480–490.

[34] M. Weiser, “Programmers use slices when debugging,” *Commun. ACM*, vol. 25, no. 7, pp. 446–452, Jul. 1982.

[35] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang, “Pse: Explaining program failures via postmortem static analysis,” in *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, ser. SIGSOFT ’04/FSE-12. ACM, 2004, pp. 63–72.

[36] M. Acharya and B. Robinson, “Practical change impact analysis based on static program slicing for industrial software systems,” in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE ’11. ACM, 2011, pp. 746–755.

[37] S. Kim, E. J. Whitehead, Jr., and Y. Zhang, “Classifying software changes: Clean or buggy?” *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, Mar. 2008. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.70773

[38] A. E. Hassan, “Predicting faults using the complexity of code changes,” in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070510

[39] R. Moser, W. Pedrycz, and G. Succi, “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,” in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE ’08. New York, NY, USA: ACM, 2008, pp. 181–190. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368114

[40] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empirical Softw. Engg.*, vol. 17, no. 4-5, pp. 531–577, Aug. 2012.

[41] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (n)," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*, ser. ASE '15. IEEE CS, 2015, pp. 476–481. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.73

[42] https://deeplearning4j.org/.

[43] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. ACM, 2013, pp. 532–542.

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

[45] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. ACM, 2014, pp. 419–428.

[46] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE 2015. IEEE CS, 2015.

[47] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "API code recommendation using statistical learning from fine-grained changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. ACM, 2016, pp. 511–522. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950333

[48] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Divide-and-conquer approach for multi-phase statistical migration for source code (t)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. IEEE CS, 2015, pp. 585–596. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.74

[49] M. Raghothaman, Y. Wei, and Y. Hamadi, "SWIM: synthesizing what I mean," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE 2016. ACM Press, 2016.

[50] T. Nguyen, P. C. Rigby, A. T. Nguyen, M. Karanfil, and T. N. Nguyen, "T2API: Synthesizing API code usage templates from english texts with statistical translation," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. ACM, 2016, pp. 1013–1017. [Online]. Available: http://doi.acm.org/10.1145/2950290.2983931