DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization

Xia Li
UT Dallas, USA
xxl124730@utdallas.edu

Wei Li
SUSTech*, China
liw7@mail.sustech.edu.cn

Yuqun Zhang
SUSTech, China
zhangyq@sustech.edu.cn

Lingming Zhang
UT Dallas, USA
lingming.zhang@utdallas.edu

ABSTRACT

Learning-based fault localization has been intensively studied recently. Prior studies have shown that traditional Learning-to-Rank techniques can help precisely diagnose fault locations using various dimensions of fault-diagnosis features, such as suspiciousness values computed by various off-the-shelf fault localization techniques. However, with the increasing dimensions of features considered by advanced fault localization techniques, it can be quite challenging for the traditional Learning-to-Rank algorithms to automatically identify effective existing/latent features. In this work, we propose DeepFL, a deep learning approach to automatically learn the most effective existing/latent features for precise fault localization. Although the approach is general, in this work, we collect various suspiciousness-value-based, fault-proneness-based and textual-similarity-based features from the fault localization, defect prediction and information retrieval areas, respectively. DeepFL has been studied on 395 real bugs from the widely used Defects4J benchmark. The experimental results show DeepFL can significantly outperform state-of-the-art TraPT/FLUCCS (e.g., localizing 50+ more faults within Top-1). We also investigate the impacts of deep model configurations (e.g., loss functions and epoch settings) and features. Furthermore, DeepFL is also surprisingly effective for cross-project prediction.

CCS CONCEPTS

• **Software and its engineering** → Software testing and debugging.

KEYWORDS

Fault localization, Deep learning, Mutation testing

ACM Reference Format:
Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3293882.3330574

* SUSTech is short for Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China.

1 INTRODUCTION

It has been reported that debugging software faults can take up to 80% of the total software cost for some projects [1]. Therefore, it is essential to develop automated techniques to help reduce the manual efforts during the software debugging process. In the literature, various *fault localization* techniques [2–10] have been proposed to automatically localize potential faulty code locations. Fault localization techniques usually analyze various dynamic execution information to compute the suspiciousness (i.e, probability to be faulty) of each program element (such as statement or method). Then, a ranked list of program elements (based on the descending order of suspiciousness values) can be either provided to developers for manual fixing or serve as the first step of automated program repair [11–17]. Please refer to a recent survey for more details [18].

Among the existing fault localization methodologies, *spectrum-based fault localization* (SBFL) has been intensively studied due to its lightweightness and effectiveness. Typical SBFL techniques (such as Tarantula [19], Ochiai [20], DStar [21], Jaccard [5], Kulczynski2 [22]) simply apply statistical analysis on the coverage data of failed/passed tests to compute the suspiciousness of code elements. The basic intuition is that code elements executed by more failed tests are more suspicious. Despite widely studied, SBFL has clear limitations in design, i.e., elements executed by failed tests may not have caused the test to fail and faulty elements may also be executed by passed tests coincidentally. To bridge the gap, *mutation-based fault localization* (MBFL) was proposed to mutate program source code to check the actual impact of each code element on the outcomes of tests [23–26]. Typical MBFL techniques (such as FIFL [27], Metallaxis [26] and MUSE [23]) first apply *mutation testing* [28] to generate *mutants* for the original program under test. Each mutant is exactly the same with the original program but with one syntactic change based on the predefined rules (called *mutation operators*, such as changing `if(a>b)` into `if(a<b)`). Then, MBFL techniques use mutants to check the impacts of code elements on the test outcomes for precise fault localization.

Although MBFL techniques consider the impact information, they may still perform poorly for some cases, e.g., some elements may not have any mutant to simulate its impact [29]. Actually, to date, there is no optimal fault localization technique that can perform the best for all cases. Therefore, recently, researchers started to combine the strengths of various traditional fault localization techniques via machine learning. Learning-to-Rank [30], a supervised machine learning technique for solving ranking problems in the field of information retrieval, has been widely used to combine the suspiciousness values computed by various fault localization techniques for more effective fault localization [8, 9, 29, 31]. In typical Learning-to-Rank fault localization techniques, various suspiciousness values computed by different fault localization techniques are used as learning features and whether the element is faulty is treated as the label information. E.g., MULTRIC [9] combines various SBFL techniques via Learning-to-Rank, while TraPT [29] combines SBFL and MBFL techniques via Learning-to-Rank.

Although Learning-to-Rank has been demonstrated to be effective for fault localization, it may not fully utilize the training data information since it lacks the capability to automatically select existing powerful features and discover new advanced features. Therefore, in this work, we propose DeepFL, a general deep-learning approach to automatically identify or create the most effective features for precise fault localization. DeepFL takes suspiciousness-based features from the fault localization area (including both SBFL and MBFL), fault-proneness-based features (e.g., code-complexity metrics) from the defect prediction area [32] and textual-similarity-based features from the information retrieval area [33]. We have built various DeepFL techniques based on the TensorFlow framework [34], including the basic Multi-layer Perceptron (MLP) and Recurrent Neural Networks models, as well as our tailored model variants considering the *hierarchical* connections between different feature groups for deep fault localization (i.e., MLP_{DFL}). To evaluate DeepFL, we perform a study on 395 real software faults from the widely used Defects4J benchmark (V1.2.0) [35]. The experimental results show that DeepFL can significantly outperform state-of-the-art TraPT [29] and FLUCCS [31] (e.g., localizing 213 faults within Top-1, 50+ more than TraPT/FLUCCS). The study also investigates the impacts of different deep model configurations and features. The results reveal various interesting findings, including: (1) deep models considering *hierarchical* feature group connections can significantly outperform the traditional Learning-to-Rank algorithms, while simply increasing deep learning layers does not help; (2) the mutation-based features are the most important ones for fault localization; (3) the *softmax* loss function is more stable than the *pairwise* loss function for DeepFL. Finally, the study shows that DeepFL can even perform precise fault localization for cross-project prediction, indicating a promising future for deep-learning-based fault localization. This paper makes the following contributions:

- **DeepFL** A deep-learning-based approach to predict potential faulty locations via incorporating various dimensions of fault diagnosis information.
- **Techniques** A set of DeepFL techniques (implemented using TensorFlow) based on widely used neural networks (such as MLP, RNN), and our tailored MLP_{DFL}.
- **Study** An extensive study demonstrating the effectiveness of DeepFL techniques on localizing real-world bugs as well as investigating various configurations of DeepFL.
- **Dataset** An extensive fault localzation dataset for all 395 Defects4J V1.2.0 bugs, including 200+ unique dynamic or static fault localization features for each program element within each buggy version.

2 BACKGROUND AND RELATED WORK

2.1 Traditional Fault Localization

Spectrum-based Fault Localization. SBFL [2, 5, 19, 20, 22, 36–39] has been intensively studied in the literature. Although various SBFL techniques have been proposed, they share the same basic insight, i.e., code elements mainly executed by failed tests are more suspicious. The input of SBFL is the coverage information of all tests and the output is a ranked list of code elements (e.g., statements or methods) according to their descending order of suspiciousness values calculated by specific formulae. To date, various SBFL techniques have been proposed, including Tarantula [40], SBI [37], Ochiai [20] and Jaccard [5]. The calculation of these SBFL techniques mainly rely on: (1) the set of all failed/passed tests, i.e., T_f/T_p, (2) the set of failed/passed tests executing code element e, i.e., $T_f(e)/T_p(e)$, and (3) the set of failed/passed tests that do not execute code element e, i.e., $T_f(\bar{e})/T_p(\bar{e})$. For example, SBI formula can calculate the suspiciousness value of one code element e as $Susp(e) = |T_f(e)|/(|T_f(e)|+|T_p(e)|)$. The objective of SBFL is to rank the actual faulty elements as high as possible to save developers' efforts in manually inspecting the ranked elements, or save the CPU time during automated program repair [13–15, 41–44].

Mutation-based Fault Localization. MBFL [23–27] aims to additionally consider impact information for fault localization. Since code elements covered by failed/passed tests may not have any impact on the corresponding test outcomes, typical MBFL techniques use mutation testing [28, 45, 46] to simulate the impact of each code element for more precise fault localization. Here we mainly discuss the general MBFL techniques since the regression MBFL techniques (e.g., FIFL [27]) are not closely related to this work. The first general MBFL technique, Metallaxis [24, 26] is based on the following intuition: if one mutant has impacts on failed tests (e.g., the tests outcomes change after mutation), its corresponding code element may have caused the test failures; similarly, if one mutant has impacts on passed tests, its corresponding code element may not be faulty (otherwise the passed tests would have failed). Metallaxis treats mutants that have impacts on tests as elements covered by the tests while the others as uncovered. Then it applies traditional SBFL formulae to calculate the suspiciousness of each mutant. Finally, the maximum suspiciousness value of mutants is returned as the suspiciousness of their corresponding code element. Assume formula SBI is used, the suspiciousness of mutant m is $Susp(m) = |T_f^{(m)}(e)|/(|T_f^{(m)}(e)|+|T_p^{(m)}(e)|)$, where $|T_f^{(m)}(e)|/|T_p^{(m)}(e)|$ is the number of failed/passed tests impacted by m on element e.

The more recent MUSE [23] technique has similar insights: (1) mutating faulty elements may cause more failed tests to pass than mutating correct elements; (2) mutating correct elements may cause more passed tests to fail than mutating faulty elements. In MUSE, the suspiciousness of mutant m is computed as $Susp(m) = |T_f^{(m)}(e)|/|T_f|-\alpha * |T_p^{(m)}(e)|/|T_p|$, where T_p/T_f denotes the set of originally passed/failed tests, and $T_f^{(m)}(e)/T_p^{(m)}(e)$ denotes the set of originally failed/passed tests that have changed outcomes on mutant m. α is a balancing weight and can be calculated as $(f2p/|T_f|) * (|T_p|/p2f)$, where $f2p/p2f$ denotes the total number of failed/-passed tests that changed outcomes during mutation.

2.2 Learning-Based Fault Localization

Learning-to-Rank is an application of supervised machine learning for solving ranking problems in information retrieval [30]. In the *learning* phase, Learning-to-Rank takes a group of training data as input, and learns a ranking model by taking specific attributes of documents and queries as different features, e.g., cosine similarity and proximity values. The ranking model is usually an optimal combination of weights for different features. Then, in the *ranking* phase, the ranking model is used to predict a ranked list of documents by accepting a set of test data including new queries and documents. Learning-to-Rank has three major categories of approaches: (1) pointwise approach, (2) pairwise approach, and (3) listwise approach. *Pointwise* indicates that each document in the training data has its own label. *Pairwise* indicates that each pair of two documents can be computed a label based on their ordering for the given query. *Listwise* indicates that the order of a list of documents is considered for prediction.

Recently, Learning-to-Rank has been applied to improve the effectiveness of spectrum-based fault localization [8, 9, 29, 31]. The basic idea of Learning-to-Rank fault localization is to learn the potential faulty locations via combining suspiciousness values computed by various fault localization techniques (i.e., features). MULTRIC [9] is the first Learning-to-Rank fault localization technique to combine different suspiciousness values from SBFL. Later on, researchers have also combined SBFL suspiciousness values with other information, e.g., program invariant [8] or source code complexity information [31], for more effective Learning-to-Rank fault localization. TraPT [29] has been proposed to combine suspiciousness values not only from SBFL, but also from MBFL. More specifically, TraPT further extends Metallaxis and MUSE to have MBFL at different test outcome levels. Recently, another independent work also proposed to combine various different existing techniques via Learning-to-Rank [47]. In all traditional Learning-to-Rank fault localization techniques, pairwise approach is selected because fault localization aims to rank faulty elements higher than correct ones, and other relationships within faulty or correct elements are not considered. Formally, assume $\mathbf{x}_e$ denotes the corresponding training feature vector for element e, then the ith feature value of e is $\mathbf{x}_e^{(i)}$ ($i = 1, 2, \ldots n$). A Learning-to-Rank algorithm can learn the weight for each feature of e as $\mathbf{W}_e^{(i)}$ ($i = 1, 2, \ldots n$), and the new combined suspiciousness value of element e can be calculated as: $\hat{\mathbf{y}}_e = \mathbf{W}_e^\top \mathbf{x}_e$. Then the code elements can be ranked according to new suspiciousness values, and the loss function can be defined as the number of incorrectly rankings: $\mathcal{L} = \sum_{\langle e^+, e^- \rangle} \parallel \hat{\mathbf{y}}_{e^+} \leq \hat{\mathbf{y}}_{e^-} \parallel$, where e^+ and e^- denote any pair of faulty and correct code elements.

Actually, various other machine learning techniques, and even neural networks have also been applied to fault localization [48–51]. However, those techniques mainly work on the test coverage information, which has clear limitations (e.g., it cannot distinguish elements accidentally executed by failed tests and the actual faulty elements) [29], and are usually studied on artificial faults or small programs. In contrast, DeepFL is the first deep-learning-based approach to incorporate various dimensions of fault diagnosis information, and has been studied on real faults of real-world projects.

3 APPROACH

In this section, we discuss the basic ideas of the used deep learning models, as well as how we apply them to DeepFL (Section 3.1). Then we introduce the features used in DeepFL (Section 3.2). Finally, we discuss about the DeepFL loss functions (Section 3.3).

Function	Formula	Range
sigmoid(z)	$\frac{1}{1+e^{-z}}$	(0,1)
tanh(z)	$\frac{1-e^{-2z}}{1+e^{-2z}}$	(-1,1)
ReLU(z)	$max(0, z)$	[0,∞)
softmax($z^{(i)}$)	$\frac{e^{z^{(i)}}}{\Sigma_{j=1}^{l} e^{z^{(j)}}}$	(0,1)

Figure 1: Activation func.

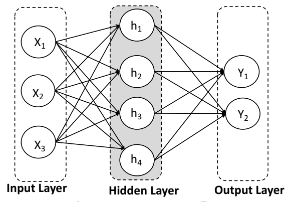


Figure 2: MLP

3.1 Deep Learning for Fault Localization

Previous fault localization techniques (such as MULTRIC [9], Savant [8], FLUCCS [31], and TraPT [29]) all utilized the Learning-to-Rank algorithms to predict potential fault locations based on various fault diagnosis features (shown in Section 3.2). However, traditional machine learning techniques (including Learning-to-Rank) largely rely on existing features, and cannot discover new latent features for more effective learning. Furthermore, it is also important to identify the most important features for effective fault localization with the increasing number of features. Recently, deep learning has demonstrated its superiority in feature engineering (i.e., including selecting useful existing features and learning latent features), and has been widely used to solve software engineering problems, such as defect prediction [32] and requirements traceability [52]. Deep learning [53] is the application of artificial neural network (ANN) with hidden layers to solve machine learning tasks. Backpropagation [54] is widely used for training and adjusting ANN internal weights to better compute the representation in each layer. In this section, we will introduce how we adapt the traditional MLP [55] and RNN networks [56] for fault localization; lastly, we will also present our tailored MLP models for deep fault localization.

3.1.1 Multi-Layer Perceptron. Multi-layer Perceptron (MLP) is a basic class of feedforward ANNs which indicates that the network does not have any loop and the output of each node does not affect the node itself [57]. MLP is a supervised learning algorithm that learns a function f mapping from $\mathbb{R}^n$ to $\mathbb{R}^l$ by training a dataset which includes n features and l labels. It can learn a non-linear function approximator for either classification or regression problem. Assume that there is a set of training data D={($\mathbf{x}_1$, $\mathbf{y}_1$), ($\mathbf{x}_2$, $\mathbf{y}_2$), ... , ($\mathbf{x}_m$, $\mathbf{y}_m$)}, where $\mathbf{x}_i \in \mathbb{R}^n$ and $\mathbf{y}_i \in \mathbb{R}^l$, and one hidden layer with k nodes, the function that MLP learns is as following:

$$f(\mathbf{x}) = \sigma_o(\mathbf{W}_{ho}^\top * \sigma_h(\mathbf{W}_{ih}^\top * \mathbf{x} + \mathbf{b}_h) + \mathbf{b}_o) \quad (1)$$

where $\boldsymbol{W}_{ih} \in \mathbb{R}^{n \times k}$ represents the weights between input layer and hidden layer and $\boldsymbol{W}_{ho} \in \mathbb{R}^{k \times l}$ represents the weights between hidden layer and output layer. $\mathbf{b}_h \in \mathbb{R}^k$ and $\mathbf{b}_o \in \mathbb{R}^l$ represent the bias of hidden layer and output layer, respectively. σ_h and σ_o represents the activation functions (such as *tanh*, *ReLU*, *sigmoid*, and *softmax*) for the hidden layer and output layer, respectively. Figure 1 (where $z^{(i)}$ presents the ith dimension of vector z) presents the definitions for four widely used activation functions in neural networks. Usually, *tanh*, *ReLU*, and *sigmoid* can be used as hidden layer activation function, while *softmax* can be used as the output layer activation function for multi-class classification problems. To illustrate, Figure 2 shows a classic MLP with one input layer, one hidden layer and one output layer.

For our study of fault localization using MLP, we directly feed our training data to MLP and use the MLP output information to

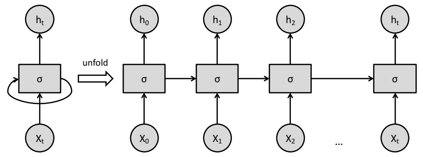


Figure 3: Standard RNN (left) and its unfolded structure

rank suspicious program elements. Note that our training labels can only be buggy or correct, thus we have two possible output classes, i.e., the number of output layer nodes (i.e., l) is always 2. Therefore, we use the *softmax* function to determine the probability of a program element being buggy. Given m program elements, each with feature set $\mathbf{x}_i$ ($i \in [1, m]$), prediction results $\hat{\mathbf{y}}_i$ is:

$$\hat{\mathbf{y}}_i = f(\mathbf{x}_i) = (Pr[\mathbf{y}_i = (1,0)], Pr[\mathbf{y}_i = (0,1)]) \tag{2}$$

where $\mathbf{y}_i = (1,0)$ denotes that the output belongs to the first output class (e.g., buggy class), while $Pr[.]$ denotes the probability function. Therefore, the first element for $\hat{\mathbf{y}}_i$ (i.e., $\hat{\mathbf{y}}_i^{(1)}$) represents the probability of the element being buggy.

3.1.2 Recurrent Neural Network. Recurrent Neural Network (RNN) is widely used in Natural Language Processing [56]. The primary difference between RNN and other deep learning models is that RNN considers the time-evolving state. Figure 3 shows a classic structure of RNN and a fully unfolded network. "Unfolded" means that the network is represented as the complete sequence. For example, if the input is a sentence of 5 words, the network would be unfolded into 5-layer. In this figure, $\mathbf{x}_t$ is the input and $\mathbf{h}_t$ is the output at time state t. Assume, the input layer has n nodes and the hidden layer has k nodes, $\mathbf{h}_t$ can be calculated based on the previous hidden output $\mathbf{h}_{t-1}$ and the input at the current state:

$$\mathbf{h}_t = tanh(\boldsymbol{W}^\top \mathbf{x}_t + \boldsymbol{U}\mathbf{h}_{t-1} + \mathbf{b}) \tag{3}$$

where $\boldsymbol{W} \in \mathbb{R}^{n \times k}$, $\boldsymbol{U} \in \mathbb{R}^{k \times k}$ and $\mathbf{b} \in \mathbb{R}^k$ represent the weight matrices and bias vector.

However, standard RNN has a primary disadvantage that the ability of network may downgrade due to vanishing gradient [58]. To overcome this disadvantage, a variant of RNN called Long Short Term Memory (LSTM) was introduced [59] to preserve long-term dependencies. The basic idea of LSTM is that a memory cell vector is introduced to preserve its state over time. The memory cell consists of an explicit memory and gating units which can control the information flow into and out of the memory. LSTM uses *input gates* to control what new information is added to cell state from current input, *forget gates* to control what information to throw away from memory, and *output gates* to decide what information to output from the memory. The state of each gate is decided by $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$. The state of forget and input gates can be calculated as:

$$\begin{aligned} \mathbf{f}_t &= sigmoid(\mathbf{W}_f^\top \mathbf{x}_t + \mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{b}_f) \\ \mathbf{i}_t &= sigmoid(\mathbf{W}_i^\top \mathbf{x}_t + \mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{b}_i) \end{aligned} \tag{4}$$

To update the information in the memory cell, a memory candidate vector $\tilde{\mathbf{c}_t}$ is firstly calculated. Then, the cell state vector aggregates old memory via the forget gate and new memory via the input gate, to update its information (Hadamard product $\circ$ is used to control the information passing the gates):

$$\begin{aligned} \tilde{\mathbf{c}_t} &= tanh(\mathbf{W}_c^\top \mathbf{x}_t + \mathbf{U}_c \mathbf{h}_{t-1} + \mathbf{b}_c) \\ \mathbf{c}_t &= \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \tilde{\mathbf{c}_t} \end{aligned} \tag{5}$$

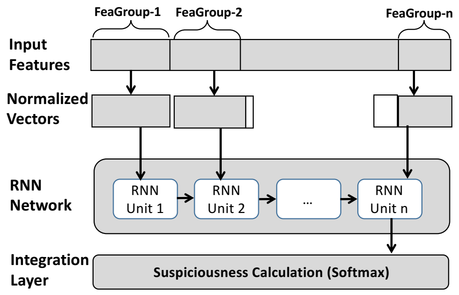


Figure 4: Fault localization architecture via RNN

Finally, LSTM calculates its output h_t based on output gate state:

$$\begin{aligned} \mathbf{o}_t &= sigmoid(\mathbf{W}_o^\top \mathbf{x}_t + \mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{b}_o) \\ \mathbf{h}_t &= \mathbf{o}_t \circ tanh(\mathbf{c}_t) \end{aligned} \tag{6}$$

When performing learning-based fault localization, we found that similar features may provide fault diagnosis information from the same dimension or information source. For example, all the suspicious values computed by spectrum-based fault localization techniques try to provide fault diagnosis information from the execution dimension, while all the code-based metric values try to provide fault diagnosis information from the code complexity dimension. Therefore, we can group all the available features into different groups based on their information sources. Then, the features in the same group can be processed together first to provide the most useful fault diagnosis information from that information source. Finally, the fault diagnosis information from different information sources can be further linked together to provide the final fault diagnosis supports. Actually, RNN with LSTM is a natural choice – different feature groups can be treated as inputs for different time steps in RNN; different feature groups can also be linked together via the shared time state to provide the final output. Figure 4 presents our network architecture of the deep fault localization technique via RNN. Shown in the figure, the inputs are the original features of each instance program element. Shown in Section 3.2, the 7 different feature groups of DeepFL can have different sizes, while RNN requires inputs for different time steps to have the same shape. Therefore, each feature group is normalized to have the same length as the largest feature group via *zero padding*. Then, the normalized feature groups can be represented as a set of vectors with uniform length, which can be directly fed into RNN for deep fault localization. By design, each RNN unit will compute the current feature group information while also considering the earlier feature groups. Then, the output of the last RNN unit actually integrates all the information learnt from all the feature groups. Finally, the output of the last RNN unit will go through the last integration layer to compute the final suspiciousness value for the program element. Assume each RNN unit has r hidden nodes, then the output of the last RNN unit for input $\mathbf{x}_i$ can be represented as $\mathbf{z}_i \in \mathbb{R}^r$, and the final prediction results can be computed as:

$$\hat{\mathbf{y}}_i = softmax(\boldsymbol{W}^\top \mathbf{z}_i + \mathbf{b}) \tag{7}$$

where $\boldsymbol{W} \in \mathbb{R}^{r \times 2}$, $\mathbf{b} \in \mathbb{R}^2$ since there are only two final output nodes (i.e., denoting buggy or correct).

3.1.3 Tailored MLP Based Neural Network. The classic MLP model introduced in Section 3.1.1 treats all the feature groups uniformly;

the hidden and output layers then compute the final prediction results via connecting all the feature groups. Such models may incur a huge number of weights (for connecting all the feature groups via fully-connected layers) to be trained, making the model training expensive and inferior. In fact, as discussed in Section 3.1.2, different feature groups contain different dimensions of debugging information, and can be processed separately first and then connected later to save the number of weights. Indeed, the RNN model discussed in Section 3.1.2 actively considered feature group information and can greatly reduce the number of weights. However, the RNN model has two clear drawbacks: (1) all the feature groups have to share the same weights based on the native design of RNN, while different feature groups contain different debugging information and should not be processed in the same way; (2) the RNN model is quite inflexible and can only take a set of plain feature groups and treat them uniformly, while different feature groups may share some hierarchical relationships (explained in more details in the next paragraph). Therefore, besides the commonly used deep learning structures, in this section we also design variants of the classic MLP model tailored for DeepFL, which we called MLP$_{DFL}$.

MLP$_{DFL}$$^{(1)}$. Figure 5 shows the network architecture of the basic variant, called MLP$_{DFL}$$^{(1)}$. As shown in the figure, each feature group is first connected with its own fully-connected layer to compute and extract the useful debugging information within this particular group. Then, the extracted debugging information for each group are concatenated together to construct a complete layer. This new complete layer can be viewed as the new hidden layer. Note that each node within this new hidden layer is only connected to the nodes within its corresponding feature group, while each node within the hidden layer of the traditional MLP is connected with nodes from all feature groups. In this way, a lot of unnecessary edges (and thus weights) can be reduced. Finally, the fully-connected layer is connected to the output layer to perform the prediction. Formally, the computation of MLP$_{DFL}$$^{(1)}$ is:

$$\begin{aligned} \mathbf{h}_t &= \sigma_{th}(\boldsymbol{W}_{th}^{\top} * \mathbf{x_t} + \mathbf{b}_{th}) \\ \mathbf{g} &= [\mathbf{h}_1, \mathbf{h}_2, ...\mathbf{h}_n] \\ f([\mathbf{x}_1, \mathbf{x}_2, ...\mathbf{x}_n]) &= \sigma_o(\mathbf{W}_{ho}^{\top} * \mathbf{g} + \mathbf{b}_o) \end{aligned} \tag{8}$$

where $\mathbf{x_t} \in \mathbb{R}^{m_t}$ represents the tth feature group (m_t is the number of features within this group), while $\mathbf{h}_t \in \mathbb{R}^{k_t}$ represents the single fully-connected layer connected with the tth feature group (k_t is the number of nodes of the fully-connected layer). n represents the number of the feature groups (e.g., 7 for Figure 5). $\mathbf{g}$ represents the complete fully-connected layer and f represents the final output. $\boldsymbol{W}_{th}^{\top} \in \mathbb{R}^{m_t \times k_t}$ represents the weights between input layer and fully-connected layer of tth feature group. $\boldsymbol{W}_{ho}^{\top} \in \mathbb{R}^{(\Sigma_{t=1}^{n} k_t) * l}$ represents the weights between complete fully-connected layer and output layer (l is the number of labels). $\mathbf{b}_{th}$ represents the bias of the fully-connected layer of tth feature group and $\mathbf{b}_o$ represents the bias of the output layer. σ_{th} and σ_o represents the activation functions for the fully-connected layer of tth feature group and output layer, respectively.

MLP$_{DFL}$$^{(2)}$. Note that some feature groups may share similar information and can also be connected earlier before connecting with all other feature groups. E.g., the four groups of mutation-based debugging information (shown in Section 3.2) are close to each other

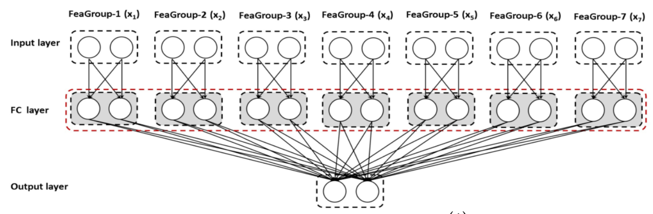


Figure 5: Basic MLP$_{DFL}$$^{(1)}$

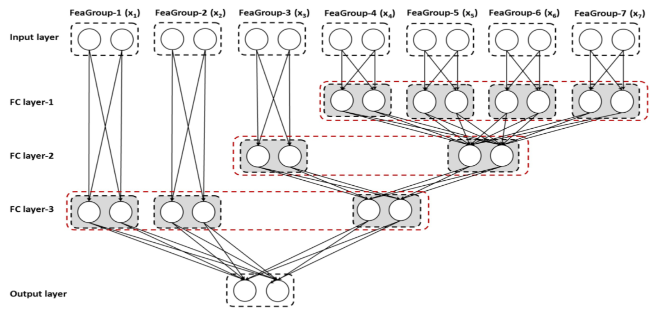


Figure 6: More advanced MLP$_{DFL}$$^{(2)}$

and should be merged first; also, the merged mutation information is more similar to the spectrum-based feature group (since they are both dynamic execution-based debugging information), and should also be merged before merging with other groups. Therefore, we further design another advanced variant (called MLP$_{DFL}$$^{(2)}$) by concatenating the single fully-connected layers of similar feature groups first. Figure 6 shows the architecture of MLP$_{DFL}$$^{(2)}$. For example, instead of connecting fully-connected layers of all feature groups, this mode first concatenates the fully-connected layers of feature groups $\mathbf{x}_4$ to $\mathbf{x}_7$ (assuming they correspond to the 4 mutation-based groups). Then, the extracted mutation-based information is further combined with the spectrum-based feature group (i.e., $\mathbf{x}_3$) since they both belong to dynamic-execution-based information. Lastly, the useful extracted dynamic-execution-based information is further concatenated with the rest two static feature groups. The motivation of the advanced MLP$_{DFL}$$^{(2)}$ variant is that it can dig more helpful and hidden debugging information in a hierarchical way before constructing the complete fully-connected layer, similar to the idea of the traditional Convolutional Neural Network (CNN) [60]. We omit the equations for the advanced variant since it can be easily inferred from Equation (8).

3.2 DeepFL Features

Existing learning-based fault localization studies [8, 9, 29, 31] have demonstrated that suspiciousness values computed from various traditional fault localization techniques can provide useful guidelines for localizing the buggy program elements. Recently, Li et al. [29] showed that combining spectrum-based and mutation-based suspiciousness values can achieve the best fault localization results. Although suspiciousness-based features are effective, they only consider the runtime correlation between program elements and the failed/passed tests. Actually, the fault-proneness of the program elements can also be useful for fault localization [31]. E.g., despite two program elements have equivalent suspiciousness values, one may still have higher probability to be faulty since it is simply more fault-prone (e.g., more complex). In addition, textual

Table 1: Studied code metrics

Source Code	Number of Class Casts	Number of Operators	Cyclomatic Complexity
	Number of Statements	Halstead Bugs	Total Depth of Nesting
	Halstead Difficulty	Number of Variable Declarations	Halstead Effort
	Number of Variable References	Halstead Length	Halstead Vocabulary
	Halstead Volume	Number of Loops	Max Depth of Nesting
	Number of Operands	Number of Java Expressions	Lines of Code
	Number of Arguments	Number of Comments	Number of Comment Lines
Bytecode	Number of Frame	Number of Insn	Number of VarInsn
	Number of TypeInsn	Number of FieldInsn	Number of MethInsn
	Number of IntInsn	Number of InvokeDynamicInsn	Number of JumpInsn
	Number of LabelInsn	Number of LdcInsn	Number of IincInsn
	Number of TableSwitchInsn	Number of LookupSwitchInsn	Number of MultiANewArrayInsn
	Number of TryCatchBlock		

similarity information from the information retrieval area can be used as another feature dimension because it can reflect the textual relevance between source code and failed tests. Thus, we include all above feature dimensions in our DeepFL general framework.

Spectrum-based Suspiciousness. Xuan and Monperrus firstly proposed MULTRIC [9] to learn faulty locations based on suspiciousness values computed by 25 traditional spectrum-based fault localization techniques, such as Jaccard [5], Ochiai [20], Ochiai2 [22], and Kulczynski2 [22]. Recently, Xie et al. [61] found that some manually-created spectrum-based formulae (such as ER1a and ER5c) can perform extremely well on single-fault programs in theory. In addition, Xie et al. [62] also generate a set of optimal spectrum-based formulae via genetic programming (GP). Following recent learning-based fault localization work [8, 29, 31], we use all the above mentioned formulae to collect the suspiciousness-based features. In total, we have the same 34 SBFL features as TraPT [29] (the duplicated formulae among prior work [9, 61, 62] were removed).

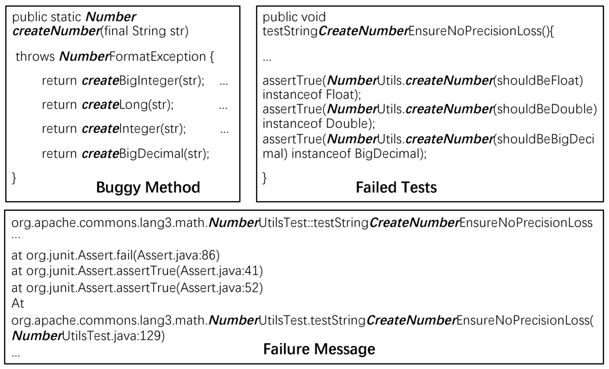
Mutation-based Suspiciousness. The recent TraPT work [29] by Li et al. firstly use suspiciousness values computed by MBFL techniques to perform learning-based fault localization, and demonstrate that they can significantly boost fault localization results. In that work, both existing general MBFL techniques (i.e., Metallaxis [26] and MUSE [23]) are used. Since Metallaxis uses traditional spectrum-based formulae, 34 Metallaxis variants are used based on the 34 used spectrum-based formulae. Furthermore, the TraPT work firstly shows that MBFL techniques can perform differently using different types of failure outputs/messages on different faulty programs. The reason is that different number of tests are computed as impacted based on different levels of test outcomes. For example, when a test t changes its exception type on a mutant m, t can be treated as impacted by m when using exception type information as test outcomes, while not impacted by m when using only pass/fail information as test outcomes (since t fails both before and after m). Therefore, TraPT further considers MBFL results with the following four different types of test outcomes: (1) Type1: pass/fail information, (2) Type2: exception type information, (3) Type3: exception type and message, and (4) Type4: exception type, message, and the full stack trace information. Note that we do not use the assertion-level information since it is not applicable to all programs, but DeepFL is general and can easily include it in the future. In total, we implement 34 Metallaxis variants and MUSE at each of the four test outcome types studied in TraPT, yielding $(34 + 1) \times 4 = 140$ suspiciousness values.

Complexity-based Fault Proneness. In the literature, code complexity metrics have been widely used to estimate the fault-proneness of code elements in the field of defect prediction [32, 63–65]. Code metrics can measure various software characteristics to reveal quality information. For example, Cyclomatic Complexity measures the number of linearly independent paths within the underlying program elements, and a program element with more independent paths may have high probability to be faulty; Halstead Difficulty [66] measures the difficulty to write or understand (e.g. when doing code review) the underlying code elements, and it can be harder to test/verify a program element that is more difficult to write or review. Recently, code metrics have been utilized to learning-based fault localization [31]. However, only three types of code complexity metrics (number of formal arguments, local variables, and statements/instructions) were used. In addition, the code complexity metrics were always applied together with change history information, making prior work only applicable to a subset of projects. To fully evaluate the effectiveness of code complexity metrics for fault localization, we collect all the 21 widely-used code complexity metrics (shown in Table 1) for the method level (since DeepFL works at the method level). In addition, we also collect the statistics for each type of Java ASM [67] bytecode instructions shown Table 1. In total, we have 37 complexity-based features.

Textual Similarity Information. Besides various suspiciousness and fault-proneness information, information retrieval (i.e.,IR) techniques have also been applied for bug localization [33, 68–71]. Such IR-based techniques investigate the textual similarity between source files and bug reports, treating each bug report as a *query* and the source files to be searched as a *document collection*. Then such techniques can return a ranked list of candidate source files based their predicted relevance with bug reports. However, in practice, bug reports are not always available, thus limiting the applications of the IR-based techniques. In this work, inspired by the idea of IR-based techniques, we investigate the textual similarity between source code methods and failed test information. Similar with the structured information retrieval work [69], we also collect different fields for both queries and documents. Query fields come from failed tests, including the *name* of failed tests, the *source code* of failed tests and the complete *failure message* (including exception type, message, and stacktrace). Document fields come from source code methods, including: the full qualified *name* of the method, *accessed classes*, *method invocations*, *used variables*, and *comments*. Each field of query can be searched on each field of document, yielding 15 combinations. For each combination, we calculate the similarity score between the query field and the document field by using the popular TF.IDF model [72, 73]. Then, we treat the similarity scores of the 15 combinations as 15 features for our DeepFL. For example, Figure 7 shows the buggy method, failed test and failure message of our subject Lang-3. From this simple example, we can find that multiple occurrences of words "create","number" and "createnumber" can be extracted from both the failed test (including failure message) and the buggy method. In this case, the similarity score between them is high based on the IR techniques, which can further help improve the effectiveness of fault localization.

3.3 DeepFL Loss Function

Loss functions guide the learning process towards certain goals and are essential for DeepFL. The *pairwise* function has been widely used for the traditional Learning-to-Rank, thus we also adapt it for our DeepFL. We also use the *cross-entropy* function [74] since it has been widely used in deep learning based classification problems.



```
public static Number
createNumber(final String str)

 throws NumberFormatException {

       return createBigInteger(str);    ...

       return createLong(str);          ...

       return createInteger(str);       ...

       return createBigDecimal(str);

}
```

Buggy Method

```
public void
testStringCreateNumberEnsureNoPrecisionLoss(){

...

assertTrue(NumberUtils.createNumber(shouldBeFloat)
instanceof Float);
assertTrue(NumberUtils.createNumber(shouldBeDouble)
instanceof Double);
assertTrue(NumberUtils.createNumber(shouldBeBigDeci
mal) instanceof BigDecimal);

}
```

Failed Tests

```
org.apache.commons.lang3.math.NumberUtilsTest::testStringCreateNumberEnsureNoPrecisionLoss
...
at org.junit.Assert.fail(Assert.java:86)
at org.junit.Assert.assertTrue(Assert.java:41)
at org.junit.Assert.assertTrue(Assert.java:52)
At
org.apache.commons.lang3.math.NumberUtilsTest.testStringCreateNumberEnsureNoPrecisionLoss(
NumberUtilsTest.java:129)
...
```

Failure Message

Figure 7: Textual similarity example from Lang-3

Pairwise. According to prior Learning-to-Rank work [75], the *pairwise* loss function which can be defined as:

$$\mathcal{L}(\theta) = \sum_{s=1}^{n-1} \sum_{i=1, \mathbf{y}_i < \mathbf{y}_s}^{n} \phi(z) + \frac{\lambda}{2} \|\theta\|_2^2 \tag{9}$$

where $z = \hat{\mathbf{y}}_s - \hat{\mathbf{y}}_i$ and the function ϕ can be $(1-z)_+$ (hinge function), e^{-z} (exponential function) or $log(1 + e^{-z})$ (logistic function). $\mathbf{y}_s$ or $\mathbf{y}_i$ denotes the label vector, and $\hat{\mathbf{y}}_s$ or $\hat{\mathbf{y}}_i$ denotes the prediction result for the sth and ith instance.

Cross-entropy. Since we transfer the fault localization problem into classification problem using MLP and RNN, we also use the widely used *cross-entropy* loss function for our approach. The *cross-entropy* loss function for m instances can be computed as:

$$\mathcal{L}(\theta) = -\frac{1}{m} \sum_{i=1}^{m} (\mathbf{y}_i^{(1)} \log \hat{\mathbf{y}}_i^{(1)} + \mathbf{y}_i^{(2)} \log \hat{\mathbf{y}}_i^{(2)}) + \frac{\lambda}{2} \|\theta\|_2^2 \tag{10}$$

where $\mathbf{y}_i^{(1)}$ denotes the first element in vector $\mathbf{y}$.

In the above loss functions, θ is the parameters involved in training the neural networks. $\frac{\lambda}{2}\|\theta\|_2^2$ denotes the L2-Regularization, which has been widely used to penalize complex models to reduce overfitting. λ is a hyperparameter that controls the magnitude of the penalty. Based on a certain loss function, various optimizer methods can be used to update the network parameters to minimize the loss, such as Stochastic Gradient Descent (SGD) [76], Nesterov Accelerated Gradient (NAG) [77], and Adaptive Moment Estimation (Adam) [78]. We use the Adam optimizer since it is a recent advanced optimizer handling both sparse gradients and non-stationary objectives well. Furthermore, Adam has been shown to work well in practice and converge faster than others [78].

4 EXPERIMENTAL SETUP

This work investigates the following research questions:

- **RQ1:** How does DeepFL perform in localizing real faults compared with state-of-the-art techniques?
- **RQ2:** How do different DeepFL architectures and different feature dimensions impact the DeepFL effectiveness?
- **RQ3:** How do different epochs and loss functions impact DeepFL results?
- **RQ4:** How does DeepFL perform in cross-project prediction?

To investigate the RQs, we use the Defects4J benchmark [35] that has been widely used in software testing research [8, 12, 13, 29, 31, 79]. We use all the 6 Defects4J V1.2.0 subjects, totalling 395 real faults detected during actual software development.

4.1 Implementation Details

Spectrum-based Features. We implement all the 34 spectrum-based fault localization formulae (Section 3.2) in Java. To collect coverage information required by SBFL, we perform on-the-fly bytecode instrumentation using ASM [67] and Java Agent [80].

Mutation-based Features. Following TraPT, we modify the widely used PIT [81] mutation testing tool (Version 1.1.5) to implement Metallaxis and MUSE: (1) we enable PIT on programs with failed tests by disabling the PIT `isgreen` check, (2) we force PIT to execute each mutant against the remaining tests after the mutant is killed by earlier tests, (3) we enable PIT to further capture detailed test outputs/messages (i.e., exception types, exception messages, and stack traces) via overriding `onTestFailure()`. Same with TraPT, we used all the 16 mutation operators available in PIT-1.1.5.

Complexity-based Features. For complexity-based features, we use Jhawk [82] to collect all the 21 source code metrics at the method level by parsing generated XML files. We also use the ASM bytecode manipulation framework to collect 16 bytecode metrics by overriding the method instruction visitor methods, such as `visitTypeInsn`, and `visitFieldInsn`.

Textual-based Features. For textual-similarity-based features, we use Indri [83] (version 5.11), a search engine that provides state-of-the-art text search and a structured query language for text collections, to generate the 15 types of similarity scores between source code methods and failed tests. We follow similar configuration with prior work [69], e.g., keeping both full and split tokens during camel case splitting, setting the baseline method as `tfidf`, and setting parameters `k1` and `b` as 1 and 0.3, respectively.

Learning Techniques. We build DeepFL models based on TensorFlow [34] (Version 1.5.0), one of the most widely used deep learning frameworks. For MLP, we implement two variants, one with one hidden layer and one with two hidden layers. For RNN, we implement Bidirectional RNN (BiRNN) with LSTM. While standard RNNs have restrictions as the future input information cannot be reached from the current state, BiRNNs allow future input information to be reachable from the current state. Lastly, we implement two variants of MLP_{DFL} – (1) the basic $\text{MLP}_{DFL}{}^{(1)}$ that treats all 7 groups of features uniformly, and (2) the hierarchical $\text{MLP}_{DFL}{}^{(2)}$. *Note that $MLP_{DFL}{}^{(2)}$ is the default DeepFL model for this work.*

For all studied techniques, we set the number of nodes in hidden layer(s) to be the same as the number of input features. We globally use the widely used learning rate of 0.001, batch size of 500, and the default training epoch of 55. We use the default *softmax* loss function (from `tf.nn.softmax_cross_entropy_with_logits`), and implement the *pairwise* loss function from scratch due to its absence in TensorFlow. Note that we tried all three ϕ functions for the *pairwise* loss function, and only kept the exponential one due to its effectiveness. Furthermore, we apply the L2-Regularization with the widely used λ value of 0.0001, and Dropout Regularization with the rate of 0.30 to reduce overfitting. For the baseline Learning-to-Rank technique, we use RankSVM with linear kernel (version 1.95) from the widely used LIBSVM [84]. We use the same LIBSVM settings as existing Learning-to-Rank fault localization techniques [8, 29, 31].

4.2 Measurements and Experimental Setup

Researchers have shown that statement-level fault localization may be too fine-grained and miss useful context information [85] and class-level fault localization is too coarse-grained to help understand and fix the bug within a class [6]. Furthermore, recent advanced program repair techniques also require precise localization of buggy methods [86, 87]. Therefore, following recent work on fault localization [8, 33, 79, 88], we perform fault localization techniques on method-level, i.e., localizing the faulty methods among all source code methods. The following measurements are used:

Recall at Top-N: In practice, most developers usually only inspect the top-ranked code elements during fault localization, e.g., 73.58% developers only check Top-5 localized elements according to a recent study [6]. Therefore, following prior work, we use Top-N (N=1, 3, 5) to denote the number of faults with at least one faulty element located within the first N positions, emphasizing earlier fault detection [69, 70, 88]. *Note that even when multiple faulty elements of a fault is localized within Top-N, it is only counted once.*

Mean Average Rank (MAR): For precise localization of all faulty elements of each fault, we compute the average ranking of all the faulty elements for each fault. MAR of each project is simply the mean of the average ranking of all its faults.

Mean First Rank (MFR): For a fault with multiple faulty elements, the localization of the first faulty element is critical since the rest faulty elements may be directly localized after that. Therefore, for each project, we use MFR to compute the mean of the first faulty element's rank for each fault.

Following prior fault localization work on Defects4J [8, 29], we perform leave-one-out cross validation on the faults for each project. To illustrate, for a project with k faulty versions in total, we separate them into two groups: one faulty version as test data to predict its rank and other k-1 faulty versions (of the same project) as training data to build the ranking model. Please note each buggy version includes a large number of training instances (i.e., methods executed by failed tests), e.g., Closure itself already has 100,000+ training instances, much larger than the famous MNIST deep-learning dataset [89]. We also applied cross-validation/L2-Regularization/Dropout to prevent overfitting. Besides the above *within-project* scenario, we also perform *cross-project* learning-based fault localization in RQ4, i.e., for localizing one faulty version of a project, all faulty versions from other five projects are used as training data. Our experiments were conducted on a Dell machine with Intel Xeon CPU E5-2660 v4@2.00GHz and 220G RAM, running Ubuntu 14.04. Our data/script are publicly available [90].

5 RESULT ANALYSIS

RQ1: DeepFL Effectiveness. To answer this RQ, we compare the effectiveness of DeepFL (with its default $\text{MLP}_{DFL}^{(2)}$ model and default setting) with state-of-the-art fault localization techniques, including MULTRIC [9], FLUCCS [31], and TraPT [29]. We also compare DeepFL with the most effective traditional SBFL and MBFL techniques, i.e., Ochiai [20] and Metallaxis [24, 26] with the Ochiai formula. Note that the original FLUCCS uses software age and change information which is not always available and cannot be applied to all Defects4J subjects, so we simplify it to integrate only its complexity metrics and spectrum-based suspiciousness values after method-level aggregation via LIBSVM. Table 2 presents the detailed experimental results. In the table, Column 1 lists all the studied Defects4J subjects; Column 2 lists all the compared techniques; the remaining columns present the main measurements used in this work. From the table, we can clearly find that DeepFL can achieve very promising overall fault localization results than other techniques. Surprisingly, DeepFL is able to localize 213 faults within Top-1, i.e., 57/53 more Top-1 faults than TraPT and FLUCCS. Also, the MFR and MAR values of DeepFL are the best among all studied techniques. We think the reason to be that $\text{MLP}_{DFL}^{(2)}$ considers the hierarchical connections between different feature groups, providing more effective fault-diagnosis information analysis.

Table 2: Comparision with the state-of-art

Subjects	Techniques	Top-1	Top-3	Top-5	MFR	MAR
Chart	Ochiai	6	14	15	9.00	9.51
	Me-Ochiai	7	15	17	12.68	13.31
	MULTRIC	7	15	16	8.08	8.85
	FLUCCS	15	19	20	3.68	4.30
	TraPT	10	15	16	5.04	5.70
	$\text{MLP}_{DFL}^{(2)}$	12	20	20	3.52	4.11
Lang	Ochiai	24	44	50	4.63	5.01
	Me-Ochiai	32	51	56	2.84	3.15
	MULTRIC	23	42	49	5.53	5.85
	FLUCCS	40	53	55	3.40	3.63
	TraPT	42	55	58	2.89	3.18
	$\text{MLP}_{DFL}^{(2)}$	46	54	59	2.15	2.53
Math	Ochiai	23	52	62	9.73	11.72
	Me-Ochiai	20	51	71	7.74	9.31
	MULTRIC	21	50	58	10.44	12.69
	FLUCCS	48	77	83	4.64	5.66
	TraPT	34	63	77	5.20	6.84
	$\text{MLP}_{DFL}^{(2)}$	63	85	91	3.72	4.84
Time	Ochiai	6	11	13	15.96	18.87
	Me-Ochiai	7	12	15	12.35	14.82
	MULTRIC	6	13	13	24.58	27.33
	FLUCCS	8	15	18	9.00	11.90
	TraPT	7	13	16	11.85	13.19
	$\text{MLP}_{DFL}^{(2)}$	13	17	17	11.92	12.62
Mockito	Ochiai	7	14	18	20.22	24.77
	Me-Ochiai	9	15	21	23.47	28.38
	MULTRIC	6	12	18	21.33	26.37
	FLUCCS	7	19	22	14.78	18.63
	TraPT	12	20	22	22.67	26.37
	$\text{MLP}_{DFL}^{(2)}$	12	19	22	11.75	13.78
Closure	Ochiai	14	30	38	90.28	102.28
	Me-Ochiai	19	47	64	23.06	27.47
	MULTRIC	17	31	41	87.34	100.85
	FLUCCS	42	66	77	36.61	48.61
	TraPT	51	83	92	14.11	19.34
	$\text{MLP}_{DFL}^{(2)}$	67	87	96	9.20	12.14
Overall	Ochiai	80	165	196	37.74	43.09
	Me-Ochiai	94	191	244	14.28	16.93
	MULTRIC	80	163	195	37.71	43.68
	FLUCCS	160	249	275	16.53	21.53
	TraPT	156	249	281	9.94	12.70
	$\text{MLP}_{DFL}^{(2)}$	213	282	305	6.63	8.27

To investigate whether the differences between DeepFL with other state-of-the-art techniques are statistically significant, we further perform Wilcoxon signed-rank test [91] with Bonferroni corrections [92]. The results show that DeepFL is significantly better than all existing techniques in terms of buggy-method rankings at significance level of 0.05 (with p-values from 1.74e-28 to 4.63e-7 after corrections and effect-size from 0.14 to 0.33). The reason for small effect-size is that bug-ranking has huge standard-deviations across all bugs for all fault localization techniques (e.g.,from 1st

Table 3: Effectiveness of different deep learning models

Techniques	Top-1	Top-3	Top-5	MFR	MAR
LIBSVM	185	268	299	9.28	11.85
MLP	174	244	269	23.94	28.15
MLP2	169	262	290	8.40	10.31
BiRNN	192	277	310	7.23	9.52
$MLP_{DFL}^{(1)}$	202	280	309	6.36	8.09
$MLP_{DFL}^{(2)}$	213	282	305	6.63	8.27

Table 4: Impacts of different dimensions of features

Techniques	Top-1	Top-3	Top-5	MFR	MAR
$MLP_{DFL}^{(1)}$	202	280	309	6.36	8.09
$MLP_{DFL}^{(1)}$-SpectrumInfor	197	269	304	7.06	9.12
$MLP_{DFL}^{(1)}$-MutationInfor	166	245	276	14.91	17.89
$MLP_{DFL}^{(1)}$-MetricsInfor	177	275	305	6.98	9.35
$MLP_{DFL}^{(1)}$-TextualInfo	198	275	308	7.02	9.09

to 498th for TraPT). Furthermore, it is recommended that effect-size should be always shown with mean difference [93], which is actually large, e.g., from prior best 12.70 to 8.27 (35% improvement).

RQ2: Impacts of DeepFL Models and Feature Dimensions. The first RQ has demonstrated the overall effectiveness of our default DeepFL model. However, it is still not clear how different DeepFL models perform for fault localization and whether we need deep learning for fault localization at all. Therefore, we compare our two MLP_{DFL} models with the three traditional deep learning models, MLP, MLP2 and BiRNN (with the default 55 epochs and *softmax* loss function), as well the traditional Learning-to-Rank technique (i.e., LIBSVM) using the same set of DeepFL features in terms of effectiveness. From the results shown in Table 3, we have following findings. First, as expected, LIBSVM can be more effective than TraPT and FLUCCS due to the additional complexity-based features and textual-similarity-based features, e.g., localizing 29/25 more faults within Top-1. Second, the traditional deep learning models can barely outperform LIBSVM, e.g., only BiRNN can slightly outperform LIBSVM. An interesting finding is that MLP with two hidden layers tends to perform even worse than MLP with one hidden layer, indicating that simply including more hidden layers cannot improve fault localization. Third, both our MLP_{DFL} variants can significantly outperform LIBSVM (e.g., $MLP_{DFL}^{(2)}$ localizes 28 more faults within Top-1 and achieves 28.56%/30.21% more precise MFR/MAR), demonstrating the effectiveness of considering the hierarchical connections of different feature groups.

We further present the time costs for collecting all DeepFL features and applying both our default DeepFL model (i.e., $MLP_{DFL}^{(2)}$) and LIBSVM on the first version (i.e., the latest and usually the largest version) of each studied subject in Table 5. In the table, Column 1 lists all the subjects. Columns 2 to Columns 5 list the time for collecting the spectrum-based, mutation-based, complexity-based and textual-similarity-based features. Columns 6 and 7 list the training and test time for DeepFL, while Columns 8 and 9 list the training and test time for the widely used LIBSVM. From the table, we can observe that the feature collection time is less than 40 minutes even for the largest subject, Closure, indicating the lightweightness of DeepFL. Note that according to prior TraPT work [29], only the *suspicious mutants* (i.e., the mutants whose mutated statements are covered by failed tests) require execution for MBFL. In our work, we execute all the suspicious mutants using 2 threads. Also, we can find that DeepFL always consumes more training time than LIBSVM due to the large number of parameters involved during the deep learning process. However, we can also observe that the DeepFL training process costs less than 7 minutes even for the largest Closure project. Such training cost is acceptable in practice since the training process is usually performed offline beforehand (e.g., before triggering the faults). Furthermore, we find that due to the optimized matrix computation used by TensorFlow, the DeepFL test time is extremely short, e.g., at most 0.12s for the studied subjects. On the contrary, LIBSVM can consume up to 2 minutes and 25 seconds for Closure, because LIBSVM tends to dump extremely large prediction model files [29]. Therefore, DeepFL can provide fault localization feedback much faster than LIBSVM (e.g., up to 1200X speedup) after the training model is ready.

So far we always apply DeepFL with all four different dimensions of features, i.e., spectrum-based, mutation-based, fault-proneness-based and textual-similarity-based features. However, it is not clear if all the four feature dimensions are necessary to perform DeepFL. Therefore, we further investigate the importance of each feature dimension for DeepFL by removing it from the entire feature set. Please note that we use our basic $MLP_{DFL}^{(1)}$ here since our default $MLP_{DFL}^{(2)}$ is hierarchical and hard to exclude features. Table 4 presents the overall result of different settings. From the table, we have following findings: (1) as expected, using all four dimensions of features can achieve the best performance; (2) the result when removing mutation-based features is the worst (only 166 faults within Top-1), showing that mutation information is essential for DeepFL; (3) removing textual-similarity-based features achieves better result than removing other feature dimensions, since the failed tests do not always present useful textual debugging information. Note that we still use the default $MLP_{DFL}^{(2)}$ to study all remaining RQs.

RQ3: Impacts of Epochs and Loss Functions. Shown in Section 4.1, our neural network models directly use widely used hyperparameters. However, the best training epoch number can be different for different problems/data. In addition, different loss functions can also affect the final results. Therefore, we further study the impact of training epoch number (i.e., less than 60) and two different loss functions on DeepFL in this RQ. Note that we do not show the impacts of learning rates since our results show they only affect convergence-rate but not effectiveness. In Figure 8, each sub-plot presents the trend of one specific measurement value when using different number of epochs and loss functions; lines with different colors and point types present two different loss functions, e.g., *softmax* and *pairwise*. From the figure, we can clearly find that *softmax* is much better than *pairwise* after around 10 epochs in terms of all measurement values. We also find that the loss functions *softmax* and *pairwise* show very different trends. With the increasing epoch number, DeepFL with *softmax* loss function achieves better results and it has a slightly slow learning curve, e.g., the results are very close after around 10 epochs in terms of Top-3/5. However, the results of DeepFL with *pairwise* loss function are almost opposite compared with *softmax*. The best result appears in around 5 epochs and other results become much worse with increasing epoch numbers. These findings actually show that *pairwise* loss function can converge to a maximum within a small number of epochs, and may incur overfitting problem with increasing training epochs. In this RQ, *softmax* loss function needs more training time to achieve better results and its stable learning curve shows that it does not suffer from the overfitting problem. One potential reason could be

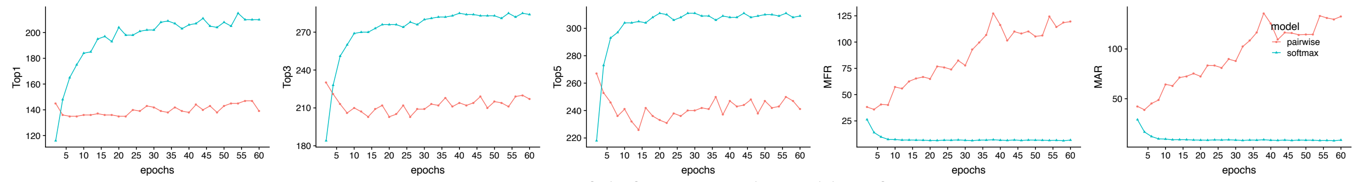


Figure 8: Impacts of different epochs and loss functions

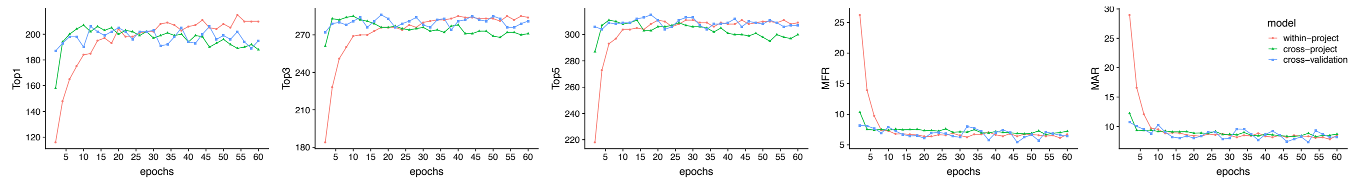


Figure 9: Impacts of different epochs for within-project, cross-project and cross-validation predictions

that *pairwise* only focuses on the comparison among buggy and bug-free elements and may overlook the overall prediction correctness. In summary, our default *softmax* loss function is most stable and effective for DeepFL.

RQ4: Cross-project Prediction. The training data and test data in previous RQs come from the same project. However, in practice, the project under debugging may not have any historical bug data available for training; even it has historical bug data, the data may not be sufficient to train an effective model. To overcome the limitations, we further extend our DeepFL across different projects to investigate the effectiveness of cross-project fault localization. That is, when performing fault localization on one buggy version of one subject, we perform the training process by using all the buggy versions of other projects. Furthermore, following prior FLUCCS work [31], we also use 10-fold cross validation to evaluate the effectiveness of DeepFL. Given the training instance data of all faults, we randomly divide them into ten different sets. Each set is used as test data with the other nine sets as training data. Figure 9 presents the main experimental results of DeepFL with *softmax* loss function for the cross-project, within-project and cross-validation predictions. The results show that the best Top-1/3/5 values of cross-project and cross-validation configurations are 207/285/308 and 206/282/310, respectively, still significantly outperforming the existing techniques. We also find that the within-project prediction performs better than cross-project and cross-validation for Top-1 after around 45 epochs. This is as expected since in the within-project scenario the training data from the same project tends to have similar distribution with the test data, and thus can perform better. Meanwhile, we also observe that the best Top-N result (i.e., 207 Top-1 faults) for cross-project prediction can be achieved much faster than the within-project and cross-validation predictions, e.g, in 10 epochs. Also, during all 60 epochs, the results of cross-validation don't fluctuate too much. These findings show that cross-project prediction can converge to an optimal value in few epochs and cross-validation keeps a stable training process. The reason is that training data of cross-project/cross-validation prediction are collected from various projects, providing more valuable instances for learning.

Threats to Validity. The main threat to *internal* validity is the potential mistake in our feature collection and technique implementation. To reduce this threat, we collect features and implement our techniques by utilizing state-of-the-art tools and frameworks, such as ASM, PIT, Jhawk, Indri and TensorFlow. The main threat to *external* validity mainly lies in the selection of the studied subjects. To reduce this threat, we plan to evaluate on more real-world projects. Also, the possible flaky tests in Defects4J [94, 95] may impact our results. Furthermore, due to the randomness of neural network models, the result might be different in different runs. To reduce this potential threat, we rerun DeepFL 10 times and observe a range from 209 to 216 in term of Top-1 value, showing the stability of our model/configuration. The main threat to *construct* validity is that the measurements used may not fully reflect real-world situations. To reduce this threat, we use Top-N, MAR and MFR metrics, which have been widely used in previous studies [9, 29, 31, 79].

Table 5: Efficiency of different techniques

Subjects	Feature collection				$\text{MLP}_{DFL}^{(2)}$		LIBSVM	
	Cov	Mutants	Metrics	Text	Train	Test	Train	Test
Chart-1	11.84s	1m 53s	14.40s	3m 20s	11.61s	0.05s	0.77s	0.26s
Time-1	9.05s	44.81s	12.06s	1m 38s	14.34s	0.05s	1.11s	0.28s
Lang-1	22.26s	50.99s	8.94s	1m 8s	3.68s	0.04s	0.38s	0.05s
Math-1	2m 15s	12m 5s	22.66s	2m 25s	18.17s	0.06s	2.27s	0.31s
Closure-1	45.65s	35m 20s	17.75s	2m 1s	6m 35s	0.12s	35.39s	2m 25s
Mockito-1	26.38s	4m 5s	3.68s	39s	24.56s	0.04s	1.56s	1.68s

6 CONCLUSION

In this paper, we propose DeepFL, a deep learning approach for localizing faults based on various feature dimensions. The experimental study on 395 real bugs from the widely used Defects4J benchmark shows that DeepFL can significantly outperform existing state-of-the-art learning-based fault localization techniques, e.g., localizing 50+ more faults within Top-1 than the most effective existing technique. The experimental results also show that the default DeepFL with tailored MLP_{DFL} is more effective than the traditional Learning-to-Rank algorithm using the same features, and can also be much faster (e.g., up to 1200X faster) for prediction. In addition, further studies on the impacts of different feature dimensions reveal that all the feature dimensions studied in our paper are useful. Furthermore, the promising result in the cross-project scenario provides a practical guide for real-world debugging.

ACKNOWLEDGMENTS

This work was partially supported by National Science Foundation under Grant Nos. CCF-1566589 and CCF-1763906, NVIDIA, Amazon, Shenzhen Peacock Plan (Grant No. KQTD2016112514355531), and Science and Technology Innovation Committee Foundation of Shenzhen (Grant No.JCYJ20170817110848086). The authors also thank Zehuai Wang for the help with the artifact preparation.

REFERENCES

[1] S. Planning, "The economic impacts of inadequate infrastructure for software testing," 2002.
[2] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *ICSM*, 2011, pp. 23–32.
[3] X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental evaluation of using dynamic slices for fault location," in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*. ACM, 2005, pp. 33–42.
[4] X. Zhang, N. Gupta, and R. Gupta, "Locating faulty code by multiple points slicing," *Software: Practice and Experience*, vol. 37, no. 9, pp. 935–961, 2007.
[5] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*. IEEE, 2007, pp. 89–98.
[6] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 165–176.
[7] X. Li, M. dâĂŹAmorim, and A. Orso, "Iterative user-driven fault localization," in *Haifa Verification Conference*. Springer, 2016, pp. 82–98.
[8] T.-D. B Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 177–188.
[9] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 191–200.
[10] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*, 2017, pp. 609–620.
[11] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, to appear.
[12] M. Martinez, T. Durieux, J. Xuan, R. Sommerard, and M. Monperrus, "Automatic repair of real bugs: An experience report on the defects4j dataset," *arXiv preprint arXiv:1505.07002*, 2015.
[13] M. Martinez and M. Monperrus, "Astor: a program repair library for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 441–444.
[14] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Software Engineering (ICSE), 2012 34th International Conference on*, 2012, pp. 3–13.
[15] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 772–781.
[16] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury, "A feasibility study of using automated program repair for introductory programming assignments," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 740–751.
[17] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury *et al.*, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 180–182.
[18] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.
[19] 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*. ACM, 2005, pp. 273–282.
[20] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on*, 2006, pp. 39–46.
[21] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using dstar (d*)," in *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*. IEEE, 2012, pp. 21–30.
[22] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, p. 11, 2011.
[23] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 153–162.
[24] M. Papadakis and Y. Le Traon, "Using mutants to locate" unknown" faults," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 691–700.
[25] ——, "Effective fault localization via mutation analysis: A selective mutation approach," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 2014, pp. 1293–1300.
[26] ——, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.
[27] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," in *OOPSLA*, 2013, pp. 765–784.
[28] T. A. Budd, "Mutation analysis of program test data," Ph.D. dissertation, New Haven, CT, USA, 1980, aAI8025191.
[29] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 92, 2017.
[30] T.-Y. Liu *et al.*, "Learning to rank for information retrieval," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 3, pp. 225–331, 2009.
[31] J. Sohn and S. Yoo, "Fluccs: using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 273–283.
[32] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 297–308.
[33] T. Dao, L. Zhang, and N. Meng, "How does execution information help with information-retrieval based bug localization?" in *Proceedings of the 25th International Conference on Program Comprehension*, 2017, pp. 241–250.
[34] "Tensorflow website," 2018. [Online]. Available: https://www.tensorflow.org/
[35] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, San Jose, CA, USA, July 23–25 2014, pp. 437–440.
[36] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, "Effective fault localization using code coverage," in *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, vol. 1. IEEE, 2007, pp. 449–456.
[37] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *ACM SIGPLAN Notices*, vol. 40, no. 6, 2005, pp. 15–26.
[38] L. Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi, "Extended comprehensive study of association measures for fault localization," *Journal of Software: Evolution and Process*, vol. 26, no. 2, pp. 172–219, 2014.
[39] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. van Hoorn, and D. Lo, "A critical evaluation of spectrum-based fault localization techniques on a large-scale software system," in *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on*. IEEE, 2017, pp. 114–125.
[40] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization for fault localization," in *in Proceedings of ICSE 2001 Workshop on Software Visualization*, 2001.
[41] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 295–306.
[42] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," 2018.
[43] S. Kim, C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair (dagstuhl seminar 17022)," in *Dagstuhl Reports*, vol. 7, no. 1. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
[44] D. Gopinath, S. Khurshid, D. Saha, and S. Chandra, "Data-guided repair of selection statements," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 243–253.
[45] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. De Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," in *ICSM*, 2010, pp. 1–10.
[46] V. Musco, M. Monperrus, and P. Preux, "A large-scale study of call graph-based impact prediction using mutation testing," *Software Quality Journal*, pp. 1–30, 2016.
[47] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, 2019.
[48] W. Zheng, D. Hu, and J. Wang, "Fault localization analysis based on deep neural network," *Mathematical Problems in Engineering*, vol. 2016, 2016.
[49] L. C. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with tarantula," in *Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on*. IEEE, 2007, pp. 137–146.
[50] Z. Zhang, Y. Lei, Q. Tan, X. Mao, P. Zeng, and X. Chang, "Deep learning-based fault localization with contextual information," *IEICE Transactions on Information and Systems*, vol. 100, no. 12, pp. 3027–3031, 2017.
[51] W. E. Wong and Y. Qi, "Bp neural network-based effective fault localization," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 04, pp. 573–597, 2009.
[52] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 3–14.
[53] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.
[54] D. E. Rumelhart, G. E. Hinton, R. J. Williams *et al.*, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1, 1988.
[55] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.
[56] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur, "Recurrent neural network based language model." in *Interspeech*, vol. 2, 2010, p. 3.

[57] M.-C. Popescu, V. E. Balas, L. Perescu-Popescu, and N. Mastorakis, "Multilayer perceptron and neural networks," *WSEAS Transactions on Circuits and Systems*, vol. 8, no. 7, pp. 579–588, 2009.
[58] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.
[59] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[60] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.
[61] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 4, p. 31, 2013.
[62] X. Xie, F.-C. Kuo, T. Y. Chen, S. Yoo, and M. Harman, "Provably optimal and human-competitive results in sbse for spectrum based fault localisation," in *International Symposium on Search Based Software Engineering*. Springer, 2013, pp. 224–238.
[63] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, "Using the support vector machine as a classification method for software defect prediction with static code metrics." in *EANN*, vol. 2009. Springer, 2009, pp. 223–234.
[64] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Engineering*, vol. 17, no. 4, pp. 375–407, 2010.
[65] K. Gao, T. M. Khoshgoftaar, H. Wang, and N. Seliya, "Choosing software metrics for defect prediction: an investigation on feature selection techniques," *Software: Practice and Experience*, vol. 41, no. 5, pp. 579–606, 2011.
[66] M. H. Halstead, *Elements of software science*. Elsevier New York, 1977, vol. 7.
[67] "Asm java bytecode manipulation and analysis framework," 2018. [Online]. Available: http://asm.ow2.org/
[68] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.
[69] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, 2013, pp. 345–355.
[70] 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 *Software Engineering (ICSE), 2012 34th International Conference on*, 2012, pp. 14–24.
[71] Q. Wang, C. Parnin, and A. Orso, "Evaluating the usefulness of ir-based fault localization techniques," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 1–11.
[72] H. P. Luhn, "A statistical approach to mechanized encoding and searching of literary information," *IBM Journal of research and development*, vol. 1, no. 4, pp. 309–317, 1957.
[73] K. Sparck Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of documentation*, vol. 28, no. 1, pp. 11–21, 1972.
[74] P.-T. De Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein, "A tutorial on the cross-entropy method," *Annals of operations research*, vol. 134, no. 1, pp. 19–67, 2005.
[75] W. Chen, T. yan Liu, Y. Lan, Z. ming Ma, and H. Li, "Ranking measures and loss functions in learning to rank," in *Advances in Neural Information Processing Systems 22*, Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, Eds. Curran Associates, Inc., 2009, pp. 315–323. [Online]. Available: http://papers.nips.cc/paper/3708-ranking-measures-and-loss-functions-in-learning-to-rank.pdf
[76] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.
[77] Y. Nesterov *et al.*, "Gradient methods for minimizing composite objective function." Core Louvain-la-Neuve, 2007.
[78] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
[79] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using pagerank," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 261–272.
[80] "Java programming language agents," 2018. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html
[81] "Pit mutation testing system," 2018. [Online]. Available: http://pitest.org/
[82] "Jhawk website," 2018. [Online]. Available: http://www.virtualmachinery.com/jhawkprod.htm
[83] "Indri website," 2018. [Online]. Available: https://www.lemurproject.org/indri.php
[84] "Libsvm website," 2018. [Online]. Available: https://www.csie.ntu.edu.tw/~cjlin/libsvm/
[85] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 199–209.
[86] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 637–647.
[87] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1, 2016, pp. 213–224.
[88] T.-D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: better together," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 579–590.
[89] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, vol. 2, 2010.
[90] "Deepfl website," 2019. [Online]. Available: https://github.com/DeepFL/DeepFaultLocalization.git
[91] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.
[92] O. J. Dunn, "Multiple comparisons among means," *Journal of the American statistical association*, vol. 56, no. 293, pp. 52–64, 1961.
[93] "Effect size," 2016. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5122517/
[94] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 643–653.
[95] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 201–211.