Towards Demystifying the Impact of Dependency Structures on Bug Locations in Deep Learning Libraries

Di Cui
Xidian University, China
cuidi@xidian.edu.cn

Xingyu Li
Xidian University, China
lixy@stu.xidian.edu.cn

Feiyang Liu
Xidian University, China
liufeiyang@stu.xidian.edu.cn

Siqi Wang
Xidian University, China
20031211660@stu.xidian.edu.cn

Jie Dai
Xidian University, China
daijiexd@foxmail.com

Lu Wang
Xidian University, China
wanglu@xidian.edu.cn

Qingshan Li*
Xidian University, China
qshli@mail.xidian.edu.cn

ABSTRACT

Background: Many safety-critical industrial applications have turned to deep learning systems as a fundamental component. Most of these systems rely on deep learning libraries, and bugs of such libraries can have irreparable consequences. **Aims:** Over the years, dependency structure has shown to be a practical indicator of software quality, widely used in numerous bug prediction techniques. The problem is that when analyzing bugs in deep learning libraries, researchers are unclear whether dependency structures still have a high correlation and which forms of dependency structures perform the best. **Method:** In this paper, we present a systematic investigation of the above question and implement a dependency structure-centric bug analysis tool: Depend4BL, capturing the interaction between dependency structures and bug locations in deep learning libraries. **Results:** We employ Depend4BL to analyze the top 5 open-source deep learning libraries on Github in terms of stars and forks, with 279,788 revision commits and 8,715 bug fixes. The results demonstrate the significant differences among syntactic, history, and semantic structures, and their vastly different impacts on bug locations. Their combinations have the potential to further improve bug prediction for deep learning libraries. **Conclusions:** In summary, our work provides a new perspective regarding to the correlation between dependency structures and bug locations in deep learning libraries. We release a large set of benchmarks and a prototype toolkit to automatically detect various forms of dependency structures for deep learning libraries. Our study also unveils useful findings based on quantitative and qualitative analysis that benefit bug prediction techniques for deep learning libraries.

*Corresponding author.

ESEM '22, September 19–23, 2022, Helsinki, Finland

ACM ISBN 978-1-4503-9427-7/22/09…$15.00
https://doi.org/10.1145/3544902.3546246

CCS CONCEPTS

• **Software and its engineering → Software Maintenance**.

KEYWORDS

Deep Learning System, Dependency Structure, Bug Prediction.

ACM Reference Format:
Di Cui, Xingyu Li, Feiyang Liu, Siqi Wang, Jie Dai, Lu Wang, and Qingshan Li*. 2022. Towards Demystifying the Impact of Dependency Structures on Bug Locations in Deep Learning Libraries. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '22), September 19–23, 2022, Helsinki, Finland*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3544902.3546246

1 INTRODUCTION

In recent years, deep learning has received much attention in many security-critical domains, such as autonomous driving [17] and healthcare [38]. However, it has been demonstrated that deep learning systems are bug-prone and may cause serious consequences, such as the Tesla/Uber accidents [8, 11]. The great majority of these systems rely on deep learning libraries, and their quality assurance is becoming increasingly crucial.

Previous studies [15, 24, 42, 46, 51, 57, 64] revealed that dependency structure is a valuable source to inform bugs and numerous approaches have been proposed to use dependency structure as the basis for bug prediction: three forms of dependency structures have been frequently and intensively studied: syntactic structure (derived from source code, capturing syntactic dependencies such as method call and inheritance), history structure (derived from the revision , capturing the co-change coupling among software entities), and semantic structure (derived from identifiers and comments, calculating the textual similarity among software entities). All of these dependency structures are considered in previous bug prediction techniques.

The problem is that when analyzing bugs in a specific and influential type of software system: deep learning libraries, it is not clear whether dependency structures still present a high correlation like existing studied software systems, and which forms of dependency structures perform best. The fundamental problem of the correlation between dependency structures and bug locations in deep learning libraries has not been adequately explored in previous

research. The answer will provide insights on designing effective and efficient bug prediction techniques for deep learning libraries to assure their quality.

In this paper, we explore the following research questions about syntactic, history, and semantic structures in deep learning libraries:

RQ1: For these dependency structures in deep learning libraries, to what extent are they similar to each other?
RQ2: For these dependency structures in deep learning libraries, do they present similar performances on bug prediction?
RQ3: Whether combinations of these dependency structures have the potential to improve bug prediction for deep learning libraries?

To answer these questions, we systematically investigate the impact of three dependency structures on bug locations. Figure 1 presents two different decision pipelines between machine learning-based and dependency structure-centric bug analysis process. For the former pipeline: machine learning-based bug analysis process, the researcher is confused about why the bug location is detected. For the latter pipeline: dependency structure-centric bug analysis process, after the applying of bug analysis tool on dependency structures, the calculated interaction is capable of providing explainable results for particular data decisions. We adopt the dependency-centric bug analysis process in our study.

We implement a dependency structure-centric bug analysis tool: Depend4BL to reveal the intrinsic correlation between dependency structures and bug locations in deep learning libraries. Compared with state-of-the-art related reverse engineering techniques, Depend4BL focuses on capturing the interplay between dependency structures and bug locations, as well as presenting interpretability, comprehensibility and time overhead advantages.

In this paper, we report our comprehensive empirical study to investigate the correlation of three dependency structures: syntactic, history, and semantic with bug locations in deep learning libraries. With the assistance of Depend4BL, we intensively studied the top 5 open-source deep learning libraries on Github, ranked by stars and forks, with 279,788 revision commits and 8,715 bug fixes. Using Depend4BL, for each dependency structure, we calculate its interaction with bug locations. Based on these data, we answer the three research questions as follows:

(1) **Similarity Analysis**. Comparing syntactic, history, and semantic structures, only about 6% of these structures in deep learning libraries are similar. This implies that their effectiveness should be varied as well when used to predict bugs and it is imperative to further explore their influences on bug locations.
(2) **Coverage Analysis**. Syntactic, history, and semantic structures capture different subsets of bug locations in deep learning libraries. The semantic structure captures the least locations but with the highest efficiency. By contrast, the history structure captures the most locations with moderate efficiency. The syntactic structure also captures moderate locations but with the lowest efficiency. Each structure has its advantages and drawbacks.
(3) **Combination Analysis**. Combinations of syntactic, history and semantic structure have the potential to effectively improve bug prediction performance: unions of all structures cover more bug locations, and combinations involving semantic structure capture severe bug locations more efficiently. Designing flexible strategies of structure combinations can enhance the bug prediction performance.

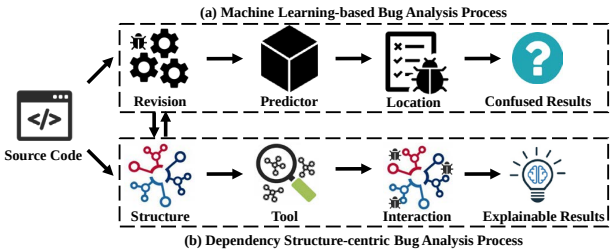

Figure 1: Illustration of machine learning-based and dependency structure-centric bug analysis process.

The contributions of our work are:

- A systematic methodology for comparing various forms of dependency architectures and their effects on bug locations in deep learning libraries. We perform similarity, coverage, and combination analysis on syntactic, history, and semantic structures. Our results first revealed the significant difference between these dependency structures in deep learning libraries, as well as their drastically different correlations with bug locations, which advanced our understanding of dependency structures in deep learning libraries.
- A new perspective to look at bugs of deep learning libraries. Our empirical study revealed that various forms of dependency structure capture vastly different subsets of bug locations. When designing a bug prediction technique for deep learning libraries, the designer should take differences of syntactic, history, and semantic structures into consideration. The intersection of semantic structure captures severe locations more efficiently, whereas the union of all structures captures more bug locations.
- Our study enables several follow-up research directions with a benchmark of deep learning libraries including the top 5 open-source deep learning libraries on Github with 279,788 revision commits, 8,715 bug fixes,1,715 bug locations, 73,225 syntactic dependencies, 203,867 history dependencies and 32,773 semantic dependencies, and a reusable toolkit: Depend4BL to automatically detect various forms of dependency structures and calculate their interactions with bug locations. Our findings can also unveil useful suggestions for bug prediction in deep learning libraries. The benchmark and toolkit are publicly available [3].

2 PRELIMINARY

In this section, we explain the terminologies used in our paper.

Dependency Structure For a software system, we model its dependency structure as a directed multi-graph which is illustrated as follows:

$$DS = \{V, E_{syn}, E_{his}, E_{sem}\} \quad (1)$$

where V represents the set of code entities within the system. Packages, files, classes, methods, and statements are all examples of code entities. We use the file as the code entity in this paper. The syntactic structure, historical structure, and semantic structure among code entities are represented by $E_{syn}, E_{his}, E_{sem}$, which are further explained as follows:

Syntactic Structure is the most frequently used dependency structure among code entities, including call, cast, contain, create, extend, implement, import, parameter, return, throw, and use [4]. Given two classes A and B, we further describe the description of each syntactic type as follows:

(1) Call, is a dependency of method invocation, such as calling the method b of class B in class A: "B.b()".
(2) Cast, is a dependency of expression and its cast type, such as casting the variable a into class B type in class A: "(B) a".
(3) Contain, indicates the definition of class, method, or variable, such as containing the variable a of class B type in class A: "B a".
(4) Create, is a dependency of method and its object created, such as creating an instance of class B type: "new B()";
(5) Extend, means inheritance of object-oriented design, such as extending class B in class A: "class A extends B {...}".
(6) Implement, is a dependency of the method implementation and its interface/abstract class, such as implementing abstract class B in class A: "class A implements B {...}".
(7) Import, indicates the introduction of class, method or variable, such as importing variable b of class B in class A: "import B.b;".
(8) Parameter, is a dependency of method and its parameter, such as variable b of class B type as the parameter in class A: "void a(B b)".
(9) Return, is a dependency of method and its return type, such as returning class B in method a of class A: "B a(){...}".
(10) Throw, is similar as Return, is a dependency of method and its throw type, such as throwing class B in class A: "throw new B();".
(11) Use, is a dependency of expression and its used types/variables, such as using variable b of class B in class A: "int a=B.b*10";

These syntactic dependencies exist in classes, methods and statements. In our paper, we use the file as the basic unit and aggregate above syntactic dependencies within files as syntactic structure: E_{syn}.

History Structure, also known as evolutionary structure, is derived from the revision history of a software system, modeling the co-change relationship among files. It is defined as follows:

$$E_{his} = \{(f_i, f_j) \mid cochange(f_i, f_j) > th \wedge f_i, f_j \in V\} \quad (2)$$

where f_i and f_j represent the file within code entities in a software system. $cochange(f_i, f_j)$ represents how f_i and f_j change together in the revision history, and its calculation is further illustrated in Section 3.

Semantic Structure is derived from the source code lexicon to capture the textual similarity between files.

$$E_{sem} = \{(f_i, f_j) \mid textsim(f_i, f_j) > th \wedge f_i, f_j \in V\} \quad (3)$$

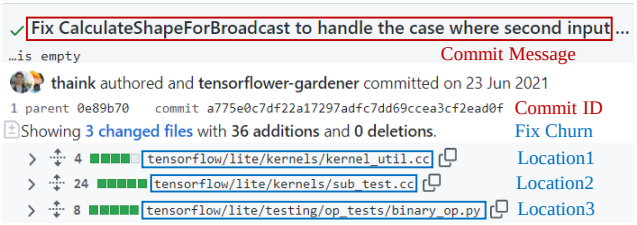

Figure 2: The record of a bug fix: commit a775e0c in TensorFlow

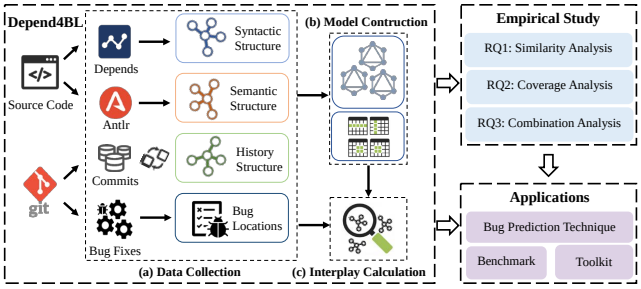

Figure 3: Overview of our study and its applications.

where f_i and f_j represent the file within code entities in a software system. th represents the threshold. $textsim(f_i, f_j)$ represents the textual similarity between f_i and f_j, and its calculation is further illustrated in Section 3.

Bug Fix is an instance of code commit which is applied to fix bugs in version control systems such as SVN [10] and Git [5]. Typically, this commit adds code changes on bug locations and also reports textual description as commit message. Figure 2 shows such a record of bug fix: commit a775e0c [1] in TensorFlow [12]. In this bug fix, we mark the commit message and commit ID. We also outline fix churn (lines of code) and bug locations. In this paper, we model a bug fix (fix_i) as a set of fixed files and involved lines of code on these files, which are illustrated as follows:

$$Fix_i = \{(f_j, churn(Fix_i, f_j)) \mid j = 1, 2, \cdots, m\} \quad (4)$$

where m represents the number of fixed files. f_j and $churn(Fix_i, f_j)$ represent each fixed file and the involved lines of code.

Bug Location describes a set of code locations frequently involved in bug fixes. In this paper, for a software system, we model the bug location (BL) as follows:

$$\begin{aligned} BL &= \{(f_j, bl_j) \mid j = 1, 2, ..., m\} \\ bl_j &= \{(Fix_i, churn(Fix_i, f_j)) \mid i = 1, 2, ..., n\} \end{aligned} \quad (5)$$

where m and n represent the number of fixed files and the number of bug fixes. For a fixed file: f_j, Fix_i and $churn(Fix_i, f_j)$ represent the involved bug fix and lines of code spent on Fix_i.

3 OVERVIEW

Figure 3 presents the overview of our study. We select 5 open-source deep learning libraries as our subjects (Section 3.2), and quantify the impact of various dependency structures on bug locations using

Depend4L by: 1) mining code repositories to collect dependency structures and bugs fixes, (2) modeling dependency structures and bug locations, and (3) calculating the interplay between dependency structures and bug locations. Based on the collected data, we conduct an empirical study by answering three research questions in Section 4. This study enables several follow-up research detailed in Section 5.

3.1 Studied Subjects

We choose 5 open-source deep learning libraries including Caffe [32], Keras [7], Pytorch [45], TensorFlow [12], and Theano [13], as our subjects for they are active in the deep learning community according to stars and forks in Github, and most of them are also frequently investigated in deep learning system research [29–31, 41, 52, 54, 62]. These subject vary in sizes, applications, and implementations.

3.2 Data Collection

Table 1 summarizes the statistical information of the data collected in this paper, including syntactic structures, history structures, semantic structures, bug fixes and bug locations. Affected by the size of the deep learning library and the tools for generating the dependency structure, the scale of dependency structure generated from different libraries can vary significantly. The details are explained as follows:

Syntactic Structure Collection. We employ Depends [4], a state-of-the-art static analysis tool, to extract syntactic dependencies among code entities. Depends can analyze 11 types of syntactic dependencies including call, cast, contain, create, extend, implement, import, parameter, return, throw, and use. Depends also support dependency analysis from different programming languages, including C and Python, which are used in our studied subjects. For each subject, we extract various types of syntactic dependencies and aggregate these dependencies at the file-level as syntactic structure.

History Structure Collection. We employ the history coupling space [40], a novel conditional probability model to manifest how likely a change to a file may influence other files. For a pair of files: f_i and f_j, if they have history dependency, the following condition should be satisfied:

$$cochange(f_i, f_j) > th \Leftrightarrow \#(f_i, f_j) > th_1 \wedge \frac{\#(f_i, f_j)}{\#(f_i)} > th_2 \quad (6)$$

where $\#(f_i, f_j)$ represents the number of co-change frequencies between f_i and f_j in the revision history. $\#(f_i)$ represents the number of change frequencies of f_i in the revision history. th_1 represents the threshold of co-change frequencies and th_2 represents the threshold of co-change probability. Following the work of Mo et al. [40], we heuristically set th_1 and th_2 as 5 and 0.33 respectively. We examined all file pairs and collect file pairs satisfied with conditions as the history structure.

Semantic Structure Collection. We take the following four steps (i.e., extract identifiers and comments, process and filter noise, generate feature vectors, and calculate textual similarity) to capture code semantics.

step1: extract identifiers and comments. We crawled the lexical information including identifiers and comments by manually implementing an Antlr-based lexical parser [2].

step2: preprocess and filter noise. We first split each collected word according to the naming convention, such as camel casing and snake casing, and then use the Porter algorithm [33] to extract the stem. Finally, 345 natural language stop words [33] are filtered from these words.

step3: generate feature vectors. For each file, we generate its feature vector with three state-of-the-art text modeling technique, including TF-IDF [35], Word2Vec [36], and BERT [48], based on preprocessed data.

step4: calculate textual similarity. For all file pairs, we calculate the cosine similarity between generated feature vectors.

For a pair of files: f_i and f_j, if they have semantic dependency, the following condition should be satisfied:

$$\begin{aligned} &textsim(f_i, f_j) > th \Leftrightarrow cosine(tf(f_i), tf(f_j)) > th_1 \\ &\wedge\ cosine(wv(f_i), wv(f_j)) > th_2 \wedge cosine(bt(f_i), bt(f_j)) > th_3 \end{aligned} \quad (7)$$

where *cosine* represents the cosine similarity between feature vectors. $tf(f_i)$, $wv(fi)$, and $bt(f_i)$ represent the feature vector generated by TF-IDF, Word2Vec, and BERT for f_i respectively. th_1, th_2, and th_3 represent the used threshold of cosine similarity in various text modeling techniques. Following the work of Cui et al. [20], we uniformly set the threshold: th_1, th_2, and th_3 as 0.66. We examined all file pairs and collect file pairs satisfied with conditions as the semantic structure.

Bug Fix Collection. Following the previous work [22, 63], we collect bug fixes from commits by heuristically mapping the commit messages and bug-related keywords (namely bug, error, fault, fix, patch or repair). For example, shown in Figure 2, this commit is identified as a bug fix containing the keyword: "Fix". To eliminate noisy data in commits, this paper combines the Meta-change aware SZZ [21], an improvement of the SZZ algorithm, to identify code changes for bug fixes.

Bug Location Collection. For each bug fix, we use GumTree [6], a state-of-the-art code difference detection tool, to detect fine-grained bug locations. For example, shown in Figure 2, this bug fix have 3 bug locations at the file-level: kernel_util.cc, sub_test.cc, and binary_op.py.

Overall, we collected 279,788 revision commits, 8,716 bug fixes, 1,715 bug locations, 73,225 syntactic dependencies, 203,867 history dependencies, and 32,773 semantic dependencies from 5 studied subjects: Caffe, Keras, Pytorch, TensorFlow, and Theano.

3.3 Model Construction

Dependency Structure Modeling. In our paper, for each form of dependency structure, we split it into a suite of overlapped subgraphs: *SGSet* according to its structural impact scope. We use each involved file as the leading file and formally define its impact subgraph: ISG_j as follows:

$$SGSet = \left\{ISG_j \mid j = 1, 2, ..., m\right\} \quad (8)$$

Table 1: Statistics of studied subjects. #Locations represents the number of bug locations: $|BL|$. #Syntactic, #History, #Semantic represents the number of collected syntactic, history, and semantic dependencies: $|E_{syn}|$, $|E_{his}|$, $|E_{sem}|$.

Subject	Version	Length of History (#Months)	#Commits	#Fixes	#Locations	#Syntactic	#History	#Semantic
Caffe [32]	1.0	9/2013 to 4/2022 (103)	12,633	632	259	45	1,651	606
Keras [7]	2.8.0	3/2015 to 4/2022 (85)	6,972	2,068	467	12,296	62,099	860
Pytorch [45]	1.11.0	1/2012 to 4/2022 (123)	97,902	75	119	21,244	33,651	9,472
TensorFlow [12]	2.8.0	11/2015 to 4/2022 (77)	134,056	250	180	37,162	106,295	21,711
Theano [13]	1.0.5	1/2008 to 4/2022 (171)	28,225	5,690	690	2,478	171	124
Sum	—	1/2008 to 4/2022 (171)	279,788	8,715	1,715	73,225	203,867	32,773

where m is the total number of impact subgraphs. Each impact subgraph (ISG_j) consists of two elements:

$$ISG_j = \left(f_j, suboridnate(f_j)\right) \tag{9}$$

where f_j represents the leading file and $suboridnate(f_j)$ represents the files that directly or indirectly depend on the leading file. In our paper, we implement the automatic splitting of dependency structures using the transitive closure function in the complex network analysis package: networkx [9]. Figure 4 presents a running example of dependency structure modeling in Pytorch. As presented, f_1 is the leading file. f_2 directly depends on f_1 and f_3-f_6 indirectly depend on f_1.

Bug Location Modeling. Based on the collected bug locations: BL, we first rank all the locations in terms of frequency and churn. We use $BLFre\_x\%ile$ to model the subset of bug locations within the top x_{th} percentile, ranked based on the number of times involved in bug fixes (bug frequency). We implement $BLFre\_x\%ile$ by iteratively counting the number of bug fixes: n for the bug location: bl_j in equation 5. We also use $BLChurn\_x\%ile$ to model the subset of bug locations within the top x_{th} percentile, ranked based on the number of lines of code spent on bug fixes (bug churn). We implement $BLChurn\_x\%ile$ by iteratively calculating the number of lines of code on bug fixes: $\sum_{i=1}^{n} churn(Fix_j, f_j)$ for the bug location: bl_j in equation 5.

3.4 Interplay Calculation

To investigate the correlation of different dependency structures with bug locations, we further calculate its interaction. The interplay calculation can be deemed as a set cover problem (SCP), which is a classical question in combinatorics and computer science shown to be one of Karp's 21 NP-complete problems [34]. In our paper, subgraphs: $SGSet$ can be analogized as the set used to cover. Bug locations, like $BLFre\_x\%ile$ or $BLChurn\_x\%ile$, can be analogized as the set to cover. As a heuristic algorithm, the greedy algorithm is efficient and easy to understand. Considering the scale of interactive computation in this paper, we design a greedy algorithm to solve this problem.

Algorithm 1 shows the procedure of our greedy algorithm. The input of this algorithm is a set of impact subgraphs: $SGSet$ and the target bug locations to be covered: $TargetSet$. The output is several subgraphs capturing the bug locations: $ResultSet$. Line 2 to 14 iteratively inspects each subgraph. Line 4 to 9 select the subgraph covering the most bug locations: $TargetSet$ from all the subgraphs: $SGSet$. After selecting, Line 11 removes the selected subgraph: $MaxSG$ and Line 13 removes the involved files of the selected subgraph: $MaxSG \cap TargetSet$. Line 15 finally returns the results: $ResultSet$.

Algorithm 1 GreedyCoverage(SGSet,TargetSet)

```
1:  ResultSet ← ∅
2:  while TargetSet ≠ ∅ do
3:      # find the subgraph covering TargetSet most
4:      MaxSG← ∅
5:      for ISG ∈ SGSet do
6:          if |ISG ∩ TargetSet| > |MaxSG ∩ TargetSet| then
7:              MaxSG ← ISG
8:          end if
9:      end for
10:     ResultSet.append(MaxSG)
11:     SGSet.remove(MaxSG)
12:     # remove involved files in the TargetSet
13:     TargetSet.remove(MaxSG ∩ TargetSet)
14: end while
15: return ResultSet
```

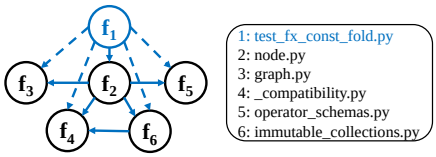

Figure 4: A running example of impact subgraph in Pytorch. test_fx_const_fold.py is the leading file. → represents direct dependencies. ⇢ represents indirect dependencies.

4 EMPIRICAL STUDY

Based on the collected data, in this paper, we aim to explore the following three research questions:

RQ1: For these dependency structures in deep learning libraries, to what extent are they similar to each other?
The answer to this question will advance our understanding regarding to the overall differences of these three dependency structures in deep learning libraries.

RQ2: For these dependency structures in deep learning libraries, do they present similar performances on bug prediction?
The answer to this question will shed lights on their differences in terms of predicting bugs for deep learning libraries.

RQ3: Whether combinations of these dependency structures have the potential to improve bug prediction for deep learning libraries?
The answer to this question will provide suggestions regarding how various dependency structures can be better leveraged to predict bugs for deep learning libraries.

We run all the experiments on a 2.4GHz Intel Xeon-4210R server with 10 logical cores and 128GB of memory.

4.1 RQ1: Similarity Analysis

Approach. To answer RQ1, we measure the overall similarity among three forms of dependency structures: syntactic structure, history structure, and semantic structure in our studied subjects using *a2a* [37]. We measure the local similarity among these dependency structures using *cvg* [37]. The related concepts are listed as follows:

a2a [37] is a percentage score that measures the distance between two dependency structures: A and B, computed as follows:

$$a2a(A,B) = (1 - \frac{mto(A,B)}{aco(A) + aco(B)}) \times 100\% \quad (10)$$

where $mto(A,B)$ is the minimum number of operations needed to transform structure A into structure B. $aco(A)$ or $aco(B)$ is the number of operations needed to construct structure A or B from a "null" structure. The calculation details are available in [37]. *a2a* returns 0% if two structures are completely different. *a2a* returns 100% if two structures are same. The closer the two structures are, the higher the score.

cvg [37] measures the extent to which two structures' subgraphs overlap:

$$cvg(A,B) = \frac{simC(A,B)}{allC(A)} \times 100\% \quad (11)$$

$simC(A,B)$ returns the subset of subgraphs from A that have at least one "similar" subgraph in B. $allC(A)$ returns the set of all subgraphs in A. The calculation details are also available in [37]. For example, given two structures A and B, $cvg(A,B) = 70\%$ and $cvg(B,A) = 40\%$ mean that 70% of subgraphs in structure A still exist in structure B, while $100\% - cvg(B,A) = 60\%$ of the subgraphs in structure B have been added based on structure A.

Result. Table 2 demonstrates the similarity results as a square matrix, where rows and columns are labeled by the same set of three dependency structures: syntactic, history, and semantic structure in the same order. The cells contain the a2a and cvg scores. For instance, the cell (Caffe.Syntactic, History.a2a) is marked with "0.3%", meaning that the a2a score between syntactic structure and history structure in Caffe is 0.3%. In this matrix, the a2a result is symmetric and the cvg result is asymmetric according to its definitions. We highlight the greatest a2a and cvg scores in studied projects with a grey background color. As presented in Table 2, syntactic, history, and semantic structure present significant differences both on the overall and local similarity: the average a2a score between syntactic, history and semantic structure of studied projects is 0.8%-5.9%. The average avg score between syntactic, history and semantic structure of studied projects is 0.1%-4.9%.

Table 2: The similarities between syntactic structure, history structure and semantic structure.

		Syntactic		History		Semantic	
Subject	Type	a2a	cvg	a2a	cvg	a2a	cvg
Caffe	Syntactic	—	—	0.3%	3.3%	1.0%	<0.1%
	History	0.3%	0.4%	—	—	16.4%	0.8%
	Semantic	1.0%	<0.1%	16.4%	3.3%	—	—
Keras	Syntactic	—	—	5.7%	1.6%	1.9%	0.2%
	History	5.7%	1.3%	—	—	1.3%	0.9%
	Semantic	1.9%	1.1%	1.3%	11.0%	—	—
KerasPytorch	Syntactic	—	—	0.3%	2.4%	0.5%	0.2%
	History	0.3%	1.3%	—	—	2.9%	0.1%
	Semantic	0.5%	1.2%	2.9%	1.1%	—	—
TensorFlow	Syntactic	—	—	0.2%	1.1%	0.5%	0.3%
	History	0.2%	0.3%	—	—	2.5%	0.1%
	Semantic	0.5%	0.6%	2.5%	1.6%	—	—
Theano	Syntactic	—	—	0.4%	0.9%	0.2%	<0.1%
	History	0.4%	1.8%	—	—	6.5%	1.2%
	Semantic	0.2%	<0.1%	6.5%	7.6%	—	—
Avg	Syntactic	—	—	1.4%	1.9%	0.8%	0.1%
	History	1.4%	1.0%	—	—	5.9%	0.6%
	Semantic	0.8%	0.6%	5.9%	4.9%	—	—

Answer to RQ1: Comparing syntactic, history, and semantic structures, only about 6% of these structures in deep learning libraries are similar.

Implications. The result of RQ1 reveals that syntactic, history, and semantic dependencies produce drastically diverse structures. Compared with similarity results of previously studied software systems [14, 20, 64], these dependency structures in deep learning libraries are substantially less similar. This implies that their effectiveness should be varied as well when used to predict bugs. It is imperative to explore their influences on bug locations.

4.2 RQ2: Coverage Analysis

Approach. To answer RQ2, we first define a specific subset of bug locations as $TargetSet$, which can be either bug locations frequently involved in bug fixing: $BLFre\_x\%ile$, or bug locations that are most expensive to fix: $BLChurn\_x\%ile$. We employ multiple types of $TargetSet$, from $BLFre\_10\%ile$ to $BLChurn\_100\%ile$ (BL), and from $BLChurn\_10\%ile$ to $BLChurn\_100\%ile$ (BL). For each $TargetSet$, we use our tool: Depend4BL to locate its corresponding interactions as the coverage result: $ResultSet$ and calculate its precision and recall score: $CPrecision$ and $CRecall$ as follows:

$$\begin{aligned} CPrecision &= \frac{|ResultSet \cap TargetSet|}{|ResultSet|} \times 100\% \\ CRecall &= \frac{|ResultSet \cap TargetSet|}{|TargetSet|} \times 100\% \end{aligned} \quad (12)$$

For each subset of dependency structures and each subset of bug locations: $TargetSet$ from each subject, we employ Depend4BL and calculate the precision and recall scores: $CPrecision$ and $CRecall$.

Result. Table 3 presents the average $CPrecision$ and $CRecall$ scores of studied subjects to cover from $BLFre\_10\%ile$ to $BLFre\_100\%ile$, and from $BLChurn\_10\%ile$ to $BLChurn\_100\%ile$ using syntactic,

Table 3: The average coverage results: *CPrecision* **and** *CRecall* **for various subsets of bug locations:** *BLFre_x%ile* **and** *BLChurn_x%ile* **in studied subjects.**

		BLFre_x%ile										
Type	Measure	10%ile	20%ile	30%ile	40%ile	50%ile	60%ile	70%ile	80%ile	90%ile	100%ile	Avg
Syntactic	CPrecision	5.3%	4.7%	7.0%	5.6%	6.7%	6.7%	8.6%	8.6%	7.3%	8.2%	6.9%
Syntactic	CRecall	38.8%$^+$	35.1%	34.9%	33.0%	29.8%	29.3%	29.8%	28.8%	28.7%	29.8%	31.8%
History	CPrecision	42.8%$^+$	37.1%$^+$	21.6%	21.9%	21.3%	22.5%	25.4%	26.0%	27.0%	28.2%	27.4%
History	CRecall	22.4%	36.7%$^+$	38.3%$^+$	41.2%$^+$	41.5%$^+$	41.6%$^+$	42.5%$^+$	42.2%$^+$	41.4%$^+$	41.8%$^+$	39.0%$^+$
Semantic	CPrecision	35.1%	23.6%	29.4%$^+$	31.2%$^+$	31.3%$^+$	32.2%$^+$	37.2%$^+$	39.7%$^+$	42.9%$^+$	41.5%$^+$	34.4%$^+$
Semantic	CRecall	4.9%	8.4%	10.0%	9.2%	9.2%	9.3%	9.6%	9.0%	9.1%	10.3%	8.9%
		BLChurn_x%ile										
Type	Measure	10%ile	20%ile	30%ile	40%ile	50%ile	60%ile	70%ile	80%ile	90%ile	100%ile	Avg
Syntactic	CPrecision	5.6%	5.7%	5.7%	9.0%	6.8%	7.9%	7.8%	8.7%	7.7%	8.2%	7.3%
Syntactic	CRecall	28.7%$^+$	30.4%	29.9%	29.8%	29.6%	28.5%	28.5%	28.5%	29.7%	29.8%	29.3%
History	CPrecision	20.6%	16.6%	17.9%	19.4%	19.4%	22.2%	23.8%	26.0%	27.8%	28.2%	22.2%
History	CRecall	28.7%$^+$	32.8%$^+$	34.9%$^+$	36.0%$^+$	37.4%$^+$	39.3%$^+$	40.6%$^+$	41.6%$^+$	42.3%$^+$	41.8%$^+$	37.6%$^+$
Semantic	CPrecision	31.8%$^+$	25.1%$^+$	25.5%$^+$	30.6%$^+$	31.0%$^+$	32.6%$^+$	34.3%$^+$	37.9%$^+$	38.7%$^+$	41.5%$^+$	32.9%$^+$
Semantic	CRecall	8.6%	6.3%	6.9%	8.3%	9.2%	8.2%	9.1%	9.8%	10.3%	10.3%	8.7%

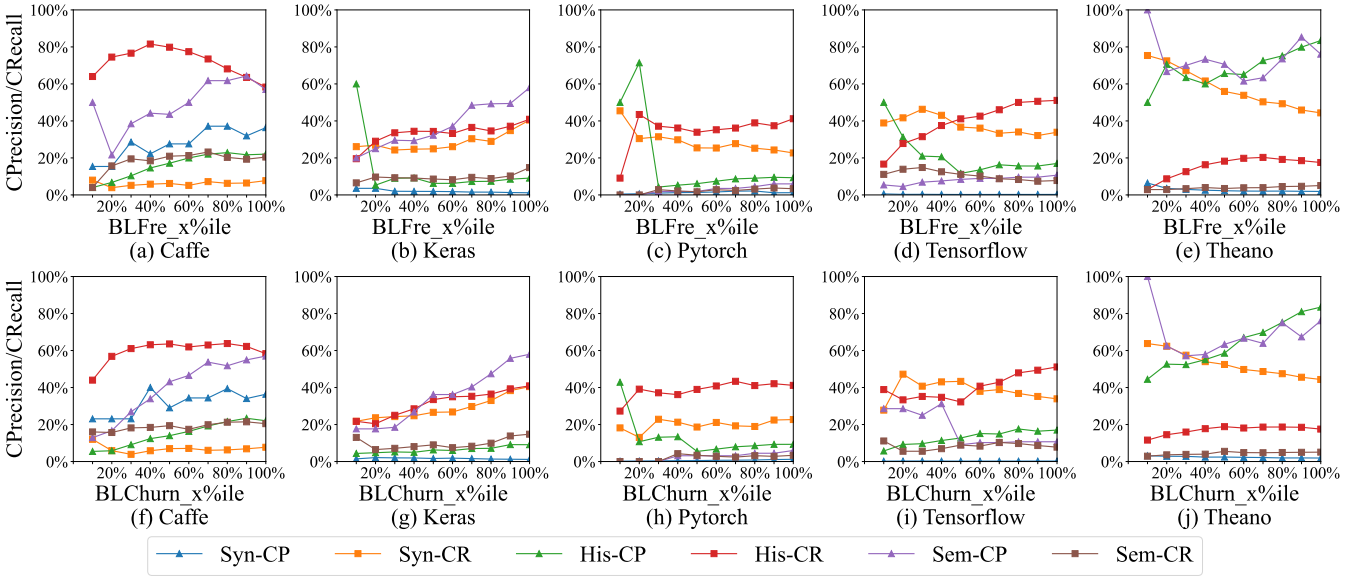

Figure 5: The coverage results: *CPrecision* **and** *CRecall* **for various subsets of bug locations:** *BLFre_x%ile* **and** *BLChurn_x%ile* **in studied subjects: Caffe, Keras, Pytorch, TensorFlow, Theano.**

history, and semantic structure respectively. We highlight the greatest CPrecision and CRecall scores with a grey background color and a + mark for each bug location. Figure 5 depicts the coverage results for each subjects, where the x-axis represents *BLFre_x%ile* or *BLChurn_x%ile* and the y-axis represents *CPrecision* or *CRecall*. As presented in Table 3 and Figure 5, we observed that bug locations are covered differently in terms of syntactic, historical, and semantic structure. The results in Table 3 and Figure 5 demonstrate that the semantic structure presents the lowest CRecall score: 8.9% in BLFre_x%ile and 8.7% in BLChurn_x%ile but the highest CPrecision score: 34.4% in BLFre_x%ile and 32.9% in BLChurn_x%ile on average. By contrast, the history structure presents the highest CRecall score: 39.0% in BLFre_x%ile and 37.6% in BLChurn_x%ile but moderate CPrecision score on average. The syntactic structure also presents a moderate CRecall score and the lowest CPrecision score: 39.0% in BLFre_x%ile and 37.6% in BLChurn_x%ile on average.

Answer to RQ2: Syntactic, history, and semantic structures capture different subsets of bug locations in deep learning libraries. The semantic structure captures the least locations but with the highest efficiency. By contrast, the history structure captures the most locations with moderate efficiency. The syntactic structure also captures moderate locations but with the lowest efficiency. Each structure has its advantages and drawbacks.

Implications. The result of RQ2 demonstrates that the syntactic, history, and semantic structures are significantly varied in terms of capturing bug locations in deep learning libraries. Compared with coverage results of previously studied software systems [14, 20, 64], the characteristics of bug locations covered by each structure vary significantly as well. The indication is that when performing bug prediction for deep learning libraries, these dependency structures should be considered and compared separately. We need a comprehensive understanding regarding to the influence of dependency structure interactions on bug locations.

4.3 RQ3: Combination Analysis

Approach. To answer RQ3, we exhaustively investigate the intersection and union of syntactic, history, and semantic structure. In total, we generate 8 combinations of these dependency structures, measure their interaction with three representative *TargetSets*: *BLFre_10%ile*, *BLChurn_10%ile*, and *BL*, and make a systematic comparison.

The 8 combinations are listed as follows: Syn ∩ His, Syn ∩ Sem, His ∩ Sem, Syn ∩ His ∩ Sem, Syn ∪ His, Syn ∪ Sem, His ∪ Sem, and Syn ∪ His ∪ Sem, where Syn, His and Sem represent the abbreviation of syntactic, history, and semantic structure. The notations, ∪ and ∩, represent the union operator and intersection operator.

For each combination, we use Depend4BL to calculate interactions with *BLFre_10%ile*, *BLChurn_10%ile* and *BL*. In addition to *CPrecision* and *CRecall*, we use the following two metrics to measure the results:

BFIR, following the work of Mo et al. [39], measures the increase of the average bug frequency of each bug location with the combination over bug locations without the combination, defined as:

$$BFIR = (\frac{fre\,(ResultSet)}{|ResultSet|} \times \frac{|BL - ResultSet|}{fre\,(BL - ResultSet)} - 1) \times 100\% \quad (13)$$

where $|ResultSet|$ represents the number of bug locations in combination. $|BL - ResultSet|$ represents the number of bug locations not in combination. $fre\,(ResultSet)$ represents the sum of bug frequencies for bug locations in $ResultSet$. $fre\,(BL - ResultSet)$ represents the sum of bug frequencies for bug locations not in $ResultSet$.

BCIR, similarly, measures the increase of the average bug churn of each bug location with the combination over bug locations without the combination, defined as:

$$BCIR = (\frac{churn\,(ResultSet)}{|ResultSet|} \times \frac{|BL - ResultSet|}{churn\,(BL - ResultSet)} - 1) \times 100\% \quad (14)$$

where $churn\,(ResultSet)$ represents the sum of bug churn for bug locations in $ResultSet$. $churn\,(BL - ResultSet)$ represents the sum of bug churn for bug locations not in $ResultSet$.

We use an example to illustrate *BFIR* and *BCIR*. For a *BL* with three bug locations: $\{A, B, C\}$, its involved bug frequencies and churn are listed as: (3, 100), (2, 200) and (1, 300). Given a *ResultSet*: $\{A, B\}$, to calculate *BFR*, the result of $fre\,(ResultSet)$ is 5 and the result of $fre\,(BL - ResultSet)$ is 1 respectively. Thus, the result of *BFIR* should be ((5/2)*(1/1)-1) = 150%, meaning that the average bug frequency within the *ResultSet* is higher than the average frequency in $BL - ResultSet$. The result of BCIR should be ((300/2)*(1/300)-1) = -50%, meaning that the average bug churn within the *ResultSet* is lower than the average frequency in $BL - ResultSet$.

Result. Table 4 presents the combination results. The column: "Type" presents each combination. For each combination, other columns present the average results of 5 studied subjects to cover *BLFre_10%ile*, *BLChurn_10%ile* and *BL* respectively. For intersections and unions of combinations, we highlight the greatest scores with a grey background color and a + mark respectively. If a combination covers few bug locations, we mark (CPrecision, CRecall, BFIR, BCIR) as (<0.1%, <0.1%, NULL, NULL).

The intersection of syntactic, history, and semantic structure capture few bug locations with most frequencies and churn: BLFre_10%ile and BLChurn_10%ile, and a small proportion of all bug locations: 5.1%, showing fewer maintenance costs than other locations in frequency and churn: -73.4% and -90.2%. The intersection of syntactic and semantic structure (Syn ∩ Sem) shows a high CPrecision score: 83.3%, 64.3% and 86.4% to cover bug locations: BLFre_10%ile, BLChurn_10%ile, and BL but a low CRecall score: 3.6%, 5.8% and 4.9%, consuming more frequency and churn than other locations. Other intersections have relatively lower CRecall scores over various bug locations as well.

The union of syntactic, history and semantic structures captures the most bug locations: 63.5% of all the bug locations, as well as the most severe bug locations: 62.5% in BLFre_10%ile and 57.5% in BLChurn_10%ile. For unions of two structures, the union of history and semantic structure (Syn ∪ His) has the highest CPrecision scores: 27.1%, 22.0% and 24.9% in BLFre_10%ile, BLChurn_10%ile and BL respectively. The union of syntactic and history structure (Syn ∪ His) covers bug locations with more churn, whereas the union of syntactic and semantic structure (His ∪ Sem) covers bug locations with more frequencies. One possible explanation is that there are considerable variations between locations with the highest frequencies and those with the highest churn.

Answer to RQ3: Combinations of syntactic, history, and semantic structure have the potential to effectively improve bug prediction performance: unions of all structures cover more bug locations, and combinations involving semantic structure capture severe bug locations more efficiently.

Implications. The result of RQ3 is unexpected. The intersection of three structures captures few bug locations, whereas their union captures the most bug locations, demonstrating that these dependency structures are independent and complementary. We also observe that combinations with semantic structure improve the coverage efficiency. This inspires us to design flexible strategies of structure combinations to enhance the bug prediction performance.

5 APPLICATION

This section discusses the follow-up research motivated by our study, including the benchmark, toolkit, and insights on future directions.

Benchmarks of correlations between dependency structures and bug locations in deep learning libraries. Our study outputs a comprehensive dataset of correlations between dependency structures and bug locations, including 279,788 revision commits,

Table 4: The combination results of syntactic, history and semantic structures. Syn, His, and Sem are abbreviations for syntactic, history, and semantic.

	BLFre_10%ile				BLChurn_10%ile				BL			
Combinations	CPrecision	CRecall	BFIR	BCIR	CPrecision	CRecall	BFIR	BCIR	CPrecision	CRecall	BFIR	BCIR
Syn ∩ His	<0.1%	<0.1%	NULL	NULL	<0.1%	<0.1%	NULL	NULL	80.7%	5.1%	-34.0%	-62.8%
Syn ∩ Sem	83.3%$^+$	3.6%	1020.1%$^+$	963.2%$^+$	64.3%$^+$	5.8%$^+$	44.8%$^+$	24.1%$^+$	86.4%	4.9%$^+$	165.7%$^+$	141.3%$^+$
His ∩ Sem	18.2%	8.0%$^+$	-46.0%	-50.5%	25.0%	2.2%	-79.5%	-68.4%	87.8%$^+$	4.6%	-34.5%	-57.1%
Syn ∩ His ∩ Sem	<0.1%	<0.1%	NULL	NULL	<0.1%	<0.1%	NULL	NULL	72.7%	5.1%	-73.4%	-90.2%
Syn ∪ His	3.6%	57.7%	95.4%	78.8%$^+$	2.4%	58.7%$^+$	59.9%	44.4%	5.9%	62.0%	34.4%	21.8%$^+$
Syn ∪ Sem	6.1%	41.1%	77.4%	28.4%	4.2%	33.4%	114.6%$^+$	-3.2%	11.5%	34.7%	36.1%$^+$	-3.7%
His ∪ Sem	27.1%$^+$	26.6%	101.4%$^+$	77.2%	22.0%$^+$	34.2%	-24.8%	50.9%$^+$	24.9%$^+$	45.6%	-12.0%	-7.4%
Syn ∪ His ∪ Sem	3.1%	62.5%$^+$	87.8%	74.7%	2.7%	57.5%	54.5%	40.4%	5.2%	63.5%$^+$	31.6%	20.7%

Table 5: The run-time performance of Depend4BL.

Tool	Caffe	Keras	Pytorch	TensorFlow	Theano
Depend4BL	0.07s	1.25s	14.24s	56.65s	0.16s
Titan [59]	0.07s	5.18s	24.11s	80.66s	0.43s
ACDC [55]	0.19s	5.57s	23.86s	80.41s	0.36s

8,715 bug fixes,1,715 bug locations, 73,225 syntactic dependencies, 203,867 history dependencies and 32,773 semantic dependencies of the top 5 open-source deep learning libraries: Caffe, Keras, Pytorch, TensorFlow, and Theano on Github in terms of stars and forks. This dataset also contains the interaction between various dependencies structures and bug locations for each subject. We believe this dataset can (1) provide an effective and reliable basis for detecting various forms of dependency structures and involved bug locations in deep learning libraries; (2) support the research on bug prediction techniques for deep learning libraries.

Dependency structure-centric bug analysis tool for deep learning libraries. Our study outputs a dependency structure-centric bug analysis tool: Depend4BL for understanding bug locations from dependency structures in deep learning libraries. We extract the core component of our analysis framework and implement Depend4BL. It has three steps, including data collection, model construction, and interplay calculation, which are described in Section 3; Table 5 presents the run-time performance. We observed that Depend4BL consume less time than state-of-the-art reverse engineering tools: Titan [59] and ACDC [56]. The output of Depend4BL can be further summarized as patterns for developers to detect code smell or extracted as machine learning features for bug prediction of deep learning libraries.

Useful insights on within-project bug prediction techniques for deep learning libraries. The result of RQ1-3 presents that when analyzing bugs in deep learning libraries, the syntactic, history and semantic structures should be combined together, which provides useful insights and hints on bug prediction techniques. We take the union of syntactic, history and semantic structures as input to construct bug predictor for deep learning libraries and evaluate its performance to validate our findings. Following the work of Qu et al. [47], we leverage a state-of-the-art network embedding technique: Node2Vec [25] to automatically extract features from structures for bug prediction. Obviously, this is a binary classification problem, and we choose 8 representative classifiers from traditional machine learning and deep learning algorithms respectively, including Decision Tree (DT), Naive Bayes (NB), Support Vector Machine (SVM), Logistic Regression (LR), Random Forest (RF), Extreme Gradient Boosting (XGB), Convolutional Neural Network (CNN), Long Short Term Memory Recurrent Neural Network (LSTM). For each subject, we use the pre-release history to predict post-release bugs, and divide the revision history as: 80% for training and 20% for testing. For the training set, we use SMOTE [16] technique to solve imbalance problem, repeat the 10-fold cross-validation 10 times (10×10) to reduce the bias, and use the grid search strategies to automatically tune the hyper-parameters of classifiers [53]. For the testing set, we evaluate the performance of classifiers using AUC, Accuracy and F1-measure, which are frequently used in the evaluation of bug prediction techniques. Table 6 presents the results of these classifiers on studied subjects. We highlight the greatest scores and observed that the results are promising achieving nearly over 90% scores on all evaluated metrics and studied subjects. The results can further be improved and extended in cross-project bug prediction and fine-grained bug prediction at the line/method level.

6 THREATS TO VALIDITY

In this section, we discuss the threats to validity and limitation of our study.

Internal threats. First, we only investigated the correlation, not the causality, between dependency structures and bug locations in deep learning libraries. We believe the presence of some bugs may be caused by changes in dependency structures. We will further explore this causality relationship in our future work. Second, we heuristically set thresholds for determining history and semantic structures according to previous studies [20]. It is unclear whether these threshold settings will generalize our studied subjects. We also manually inspected and confirmed generated dependency structures for studied projects. Third, we collect commits, history structures, and bug fixes for each subject, from the first to the latest release. Prior research [58] suggested that if only recent history is used, the result could be different. To validate our work, we recalculate the data, extracting history structures and bug fixes from each subject's most recent 3 years of revision history. The results showed that specific bug frequency and churn ranking orders are different, but the general conclusions are exactly the same. Finally, our tool: Depend4BL may produce wrong results. To reduce this threat, we made our tool open source and continue to repair bugs that have been disclosed.

Table 6: Performance of bug prediction on five subjects using the union of dependency structures with various classifiers.

Classifier	Measure	Caffe	Keras	Pytorch	TensorFlow	Theano
DT [50]	AUC	85.62%	76.54%	88.06%	86.76%	81.58%
	Accuracy	85.71%	75.50%	88.07%	86.76%	81.72%
	F1-Measure	86.00%	75.38%	88.10%	86.80%	81.03%
NB [49]	AUC	80.96%	74.24%	82.82%	72.11%	81.87%
	Accuracy	81.43%	73.37%	82.76%	72.03%	81.53%
	F1-Measure	83.54%	74.88%	81.66%	74.13%	81.76%
SVM [43]	AUC	93.91%$^+$	83.90%	85.58%	89.88%	85.53%
	Accuracy	94.37%$^+$	84.42%	85.53%	89.86%	85.54%
	F1-Measure	95.12%$^+$	86.70%	85.56%	90.74%	85.42%
LR [28]	AUC	90.52%	75.63%	86.96%	75.79%	88.49%
	Accuracy	90.14%	76.38%	86.95%	75.80%	88.54%
	F1-Measure	90.91%	79.83%	86.93%	73.99%	87.67%
RF [44]	AUC	91.41%	87.24%	88.62%	81.18%	83.64%
	Accuracy	91.55%	86.93%	88.63%	81.18%	83.63%
	F1-Measure	92.50%	86.60%	88.66%	81.85%	83.67%
XGB [18]	AUC	93.08%	87.39%	89.67%$^+$	91.51%$^+$	90.10%$^+$
	Accuracy	92.86%	86.93%	89.67%$^+$	91.51%$^+$	90.09%$^+$
	F1-Measure	92.96%	86.87%	89.67%$^+$	91.51%$^+$	90.11%$^+$
CNN [23]	AUC	92.90%	91.71%$^+$	84.27%	82.90%	84.58%
	Accuracy	92.96%	91.55%$^+$	84.37%	82.96%	84.29%
	F1-Measure	93.33%	91.89%$^+$	85.00%	82.54%	83.55%
LSTM [19]	AUC	91.33%	82.07%	87.74%	84.71%	85.14%
	Accuracy	90.14%	81.91%	87.74%	84.76%	85.54%
	F1-Measure	91.57%	83.49%	87.77%	85.01%	84.89%

External Threats. The first threat comes from the quality of bug fixes. Previous studies [26, 27] pointed out that a bug fix may not be committed to fix bugs but to finish other tasks. To reduce this threat, we manually collect expected bug fixes and further leverage related methods like untangling changes [26] to remove noise. The second threat comes from the chosen subjects. We intensively studied the top 5 open-source deep learning libraries on Github in terms of stars and forks. It is still uncertain whether our findings will generalize to other open-source or industrial deep learning libraries. Studying more subjects is our ongoing work. The third threat comes from the imbalance problem of bug data, which is a common challenge in machine learning and bug prediction. However, our objective is to investigate dependency structures in deep learning libraries. To mitigate this risk, we employ the sampling technique: SMOTE [16] in our bug prediction in Section 5. Our ongoing work is to employ advanced sampling techniques to address the imbalance problem.
Verifiability. To ensure the replicability of our study, we make the tools and the data publicly available [3]. We also run statistical tests to ensure that our measurements are statistically significant.

7 RELATED WORK

In this section, we compare our work with related research.

Dependency Structure-based Bug Prediction. Leveraging various dependency structures in bug prediction has been widely studied [15, 24, 42, 51, 57, 64]. Selby et al. [51] first extracted metrics from syntactic structures to predict bugs in source files. Zimmermann et al. [64] reported that network measures for syntactic structures are useful for constructing good bug predictors. Furthermore, Cataldo et al. [15] derived density metrics from history structures for defect prediction. Lin et al. [57] used deep neural networks to automatically learn features from semantic structures to improve bug prediction. Cui et al. [20] conducted a comparative study of various dependency structures in Apache open source projects. Compared with these previous works, our study targets for validating the correlation between dependency structures and bug locations in deep learning libraries, which hasn't been systematically investigated yet. The results may benefit designing good bug predictors for deep learning libraries.

Bug Analysis on Deep Learning Systems. There is also rich literature about bug analysis on deep learning systems. Zhang et al. [62] investigated bugs of TensorFlow and classified them into 7 root causes and 4 symptoms. Thung et al. [54] studied bug characteristics in three deep learning systems: Mahout, Lucene, and OpenNLP, including bug frequency, bug type, bug severity, bug impact, fix duration, and fix effort. Humbatova et al. [29] summarized a taxonomy of bugs in deep learning systems. Islam et al. [30, 31] further conducted an empirical study of bugs on deep neural networks and summarized their bug fixing patterns. In particular, Shen et al. [52] focused on bugs in deep learning compilers. Yan et al. [60] studied numerical bugs in deep learning systems. Zhang et al. [61] investigated bugs in Microsoft's deep learning jobs. In comparison to previous bug analysis studies on deep learning systems, our study starts with deep learning libraries, which are the foundation of most deep learning systems. Our work focus on understanding deep learning library bugs from the perspective of dependency structures, as well as forecasting these bugs at an early stage by utilizing dependency structures.

8 CONCLUSION

In this paper, we presented our systematic study on the correlation between dependency structures and bug locations in deep learning libraries. We conducted our study on the top 5 open-source deep learning libraries on Github in terms of stars and forks, involving 279,788 revision commits and 8,715 bug fixes. Supported by Depend4BL, for each subject, we gathered three forms of dependency structures: syntactic, history and semantic, and calculated their interactions with bug locations. We investigated three research questions based on these collected data. The results demonstrated the independence and complementary nature of three dependency structures in deep learning libraries, as well as their significant association with bug locations and drastically different impacts. We also presented a suite of qualitative and quantitative findings that shed light on bug prediction techniques for deep learning libraries.

ACKNOWLEDGEMENT

This work was supported by National Natural Science Foundation of China (61902288, 61972300, U21B2015), Strategic Priority Research Program of Chinese Academy of Science(XDC05040100), National Key Research and Development Program of China under Grant (2019YFB1406404), Fundamental Research Funds for the Central Universities (XJS220311).

REFERENCES

[1] 2022. *a775e0c.* https://github.com/tensorflow/tensorflow/commit/a775e0c
[2] 2022. *ANTLR.* https://github.com/antlr/antlr4

[3] 2022. *Benchmark and Toolkit.* https://anonymous.4open.science/r/ESEM22-Data-038D
[4] 2022. *Depends.* https://github.com/multilang-depends/depends
[5] 2022. *Git.* https://git-scm.com
[6] 2022. *GumTree.* https://github.com/GumTreeDiff/gumtree
[7] 2022. *KERAS.* https://keras.io/
[8] 2022. *List of self-driving car fatalities.* https://en.wikipedia.org/wiki/Self-driving_car#cite_note-15
[9] 2022. *Networkx.* https://networkx.org
[10] 2022. *SVN.* https://subversion.apache.org
[11] 2022. *Uber is giving up on self-driving cars in California after deadly crash.* https://www.vice.com/en_us/article/9kga85/uber-is-giving-up-on-self-driving-cars-in-california-after-deadly-crash
[12] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: A System for {Large-Scale} Machine Learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.
[13] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* (2016), arXiv–1605.
[14] Gabriele Bavota, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. 2013. An empirical study on the developers' perception of software coupling. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 692–701.
[15] Marcelo Cataldo, Audris Mockus, Jeffrey A Roberts, and James D Herbsleb. 2009. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering* 35, 6 (2009), 864–878.
[16] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.
[17] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. 2015. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE international conference on computer vision*. 2722–2730.
[18] Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, et al. 2015. Xgboost: extreme gradient boosting. *R package version 0.4-2* 1, 4 (2015), 1–4.
[19] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).
[20] Di Cui, Ting Liu, Yuanfang Cai, Qinghua Zheng, Qiong Feng, Wuxia Jin, Jiaqi Guo, and Yu Qu. 2019. Investigating the impact of multiple dependency structures on software defects. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 584–595.
[21] Daniel Alencar Da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. 2016. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering* 43, 7 (2016), 641–657.
[22] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-scale analysis of framework-specific exceptions in Android apps. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 408–419.
[23] Ross Girshick. 2015. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*. 1440–1448.
[24] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. 2000. Predicting fault incidence using software change history. *IEEE Transactions on software engineering* 26, 7 (2000), 653–661.
[25] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 855–864.
[26] Kim Herzig. 2013. The Impact of Tangled Code Changes. In *Working Conference on Mining Software Repositories*.
[27] Kim Herzig, Sascha Just, and Andreas Zeller. 2013. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *2013 35th international conference on software engineering (ICSE)*. IEEE, 392–401.
[28] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. 2013. *Applied logistic regression*. Vol. 398. John Wiley & Sons.
[29] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1110–1121.
[30] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 510–520.
[31] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing deep neural networks: Fix patterns and challenges. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1135–1146.
[32] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. 675–678.
[33] N. Kambhatla. 2004. Combining lexical, syntactic, and semantic features with maximum entropy models for information extraction. In *Annual Meeting of Association of Computational Linguistics, 2004*.
[34] Richard M Karp. 1975. On the computational complexity of combinatorial problems. *Networks* 5, 1 (1975), 45–68.
[35] Donghwa Kim, Deokseong Seo, Suhyoun Cho, and Pilsung Kang. 2019. Multi-co-training for document classification using various document representations: TF–IDF, LDA, and Doc2Vec. *Information Sciences* 477 (2019), 15–29.
[36] JH Lau and T Baldwin. 2019. An Empirical Evaluation of doc2vec with Practical Insights into Document Embedding Generation, July 2016.
[37] Duc Minh Le, Pooyan Behnamghader, Joshua Garcia, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. 2015. An empirical study of architectural change in open-source software systems. In *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 235–245.
[38] Siqi Liu, Sidong Liu, Weidong Cai, Sonia Pujol, Ron Kikinis, and Dagan Feng. 2014. Early diagnosis of Alzheimer's disease with deep learning. In *2014 IEEE 11th international symposium on biomedical imaging (ISBI)*. IEEE, 1015–1018.
[39] Ran Mo, Yuanfang Cai, Rick Kazman, and Lu Xiao. 2015. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*. IEEE, 51–60.
[40] Ran Mo and Mengya Zhan. 2019. History coupling space: A new model to represent evolutionary relations. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 126–133.
[41] Ruihui Mu and Xiaoqin Zeng. 2019. A review of deep learning research. *KSII Transactions on Internet and Information Systems (TIIS)* 13, 4 (2019), 1738–1764.
[42] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*. ACM, 452–461.
[43] William S Noble. 2006. What is a support vector machine? *Nature biotechnology* 24, 12 (2006), 1565–1567.
[44] Mahesh Pal. 2005. Random forest classifier for remote sensing classification. *International journal of remote sensing* 26, 1 (2005), 217–222.
[45] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).
[46] Yu Qu, Xiaohong Guan, Qinghua Zheng, Ting Liu, Lidan Wang, Yuqiao Hou, and Zijiang Yang. 2015. Exploring community structure of software Call Graph and its applications in class cohesion measurement. *Journal of Systems and Software* 108 (2015), 193–210.
[47] Yu Qu, Ting Liu, Jianlei Chi, Yangxu Jin, Di Cui, Ancheng He, and Qinghua Zheng. 2018. node2defect: using network embedding to improve software defect prediction. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 844–849.
[48] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. https://arxiv.org/abs/1908.10084
[49] Irina Rish et al. 2001. An empirical study of the naive Bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, Vol. 3. 41–46.
[50] S Rasoul Safavian and David Landgrebe. 1991. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics* 21, 3 (1991), 660–674.
[51] Richard W. Selby and Victor R. Basili. 1991. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering* 17, 2 (1991), 141–152.
[52] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 968–980.
[53] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2018. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering* 45, 7 (2018), 683–711.
[54] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 271–280.
[55] Vassilios Tzerpos and Richard C Holt. 2000. Accd: an algorithm for comprehension-driven clustering. In *Proceedings Seventh Working Conference on Reverse Engineering*. IEEE, 258–267.
[56] Vassilios Tzerpos and Richard C Holt. 2000. ACDC: An Algorithm for Comprehension-Driven Clustering.. In *wcre*. 258–267.

[57] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Ieee/acm International Conference on Software Engineering*. 297–308.

[58] Sunny Wong and Yuanfang Cai. 2011. Generalizing evolutionary coupling with stochastic dependencies. In *Ieee/acm International Conference on Automated Software Engineering*. 293–302.

[59] Lu Xiao, Yuanfang Cai, and Rick Kazman. 2014. Titan: A toolset that connects software architecture with quality analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 763–766.

[60] Ming Yan, Junjie Chen, Xiangyu Zhang, Lin Tan, Gan Wang, and Zan Wang. 2021. Exposing numerical bugs in deep learning via gradient back-propagation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 627–638.

[61] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An empirical study on program failures of deep learning jobs. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1159–1170.

[62] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 129–140.

[63] Hao Zhong and Zhendong Su. 2015. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 913–923.

[64] Thomas Zimmermann and Nachiappan Nagappan. 2008. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*. 531–540.