MiFeMoDEP - A Hybrid Mixed Feature Model for Cross Project Defect Prediction

Chandradithya S Jonnalagadda
cs21b059@iittp.ac.in
IIT Tirupati
Tirupati, AP, India

A Shree Balaji
cs21b008@iittp.ac.in
IIT Tirupati
Tirupati, AP, India

KE Nanda Kishore
cs21b025@iittp.ac.in
IIT Tirupati
Tirupati, AP, India

Karthikeya Maruvada
cs21b033@iittp.ac.in
IIT Tirupati
Tirupati, AP, India

Chetan Moturi
cs21b017@iittp.ac.in
IIT Tirupati
Tirupati, AP, India

ABSTRACT

Since its conception in 2002 [2], Cross-Project Defect Prediction has been researched extensively in Object-Oriented Software such as Java and C applications. Considering Python's meteoric rise across domains, including Machine Learning, DevOps, and Robotics, we believe that it is prudent to extend this research to Python applications too. In the very few models that have been proposed and tested on Python, there are limitations – semantic or flow information (either one or sometimes both) were not being considered, which leads to a loss of context, which is important for bug prediction. To address the limitations, we propose MiFeMoDEP, a Hybrid Mixed Feature Model which uses the flow and semantic information of source code files or commits to first predict file-level buggyness. The classifier is then passed to a LIME explainer along with a line-level dataset, which we extracted from the given dataset, to calculate scores for lines, which are then ranked.

The implementation and the replications can be found in the GitHub repo here.

KEYWORDS

Cross-Project Defect Prediction for Python, Line-level Software defect prediction, Code Property Graph, CodeBERT, Graph Neural Network, LIME

1 INTRODUCTION

Mentor - A Eashaan Rao

Software bug prediction plays an important role in the building and maintenance of software systems, and resource allocation. It reduces costs and leads to a more efficient development by helping software developers and engineers prioritize sections of the software to work on. This focus on an area with a higher likelihood of bugs produces a more stable and reliable software system.

The lack of a labeled Python defect prediction dataset was an initial roadblock for Cross-Project Defection Models for Python. The Defectors Dataset [8] fulfilled this requirement, and being the largest and most reliable open-source dataset for Python Defect Prediction, it paved the way for the current state-of-the-art Defect Prediction Models for Python, namely, JITLine [10] and Bugsplorer [7].

Defect Prediction is done at two levels of granularity — file-level and line-level. Line-level Defect Prediction is carried out in two different scenarios — Just-In-Time (JIT), which is done on commits, and Source Code Level, which is done on individual source code files.

Along with the previously stated current state-of-the-art defect prediction models for Python, JITLine, and Bugsplorer, in this study, we go over the current state-of-the-art defect prediction models for Java, LineFlowDP [19], and DeepLineDP [11].

2 RELATED WORK

2.1 JITLINE

JITLine is a Machine Learning model working at commit-level, i.e., JIT predictions at line-level. The diff of a commit is preprocessed to extract code tokens, which are then encoded to a vector representation using the CountVectorizer implementation from scikit-learn. This representation is then passed as input to a Random Forest Classifier, predicting whether each file is buggy. Finally, a Local Interpretable Model-agnostic Explanations (LIME) [12] technique is used to provide scores for each line, and their defect probabilities are thereby ranked. A glaring drawback of this model is that the tokenization eliminates the model from learning any semantic information about the program, which severely limits its ability to predict bugs. Moreover, the model was trained within project (Within-Project Defect Prediction), and hence its reach has not been tested.

2.2 DeepLineDP

DeepLineDP is a Deep Learning model working at the source code level, which uses a Hierarchical Attention Network (HAN) [20]. The source code is tokenized, using its Abstract Syntax Tree to extract each token's syntactic information. The model maintains the order of the tokens, representing each file as a sequence of lines and each line as a sequence of tokens. This representation is passed to the HAN, where the flow is as follows. Tokens are encoded and passed to a token-level attention layer. The attention scores of each token are computed with respect to each line. The lines are encoded and then passed to a line-level attention layer, where a process similar to the previous is carried out. The output of the HAN is passed to a fully connected layer, which predicts buggy files. The attention scores are used to rank the lines in terms of defect probabilities. While DeepLineDP extracts the semantic information of a file, it doesn't take into account the flow information, which also factors into a good amount of the defects. This model has been tested on Java and is cross-project.

2.3 Bugsplorer

Bugsplorer is an improved version of DeepLineDP where two transformers [14] are used in place of the HAN to perform a similar function. The output is an Lx2 vector, where L is the number of lines, and each line is mapped to two values to represent the probability of a bug. Similar to DeepLineDP, it doesn't consider the flow information but captures the semantic information much better. Bugsplorer has been experimented on the Defectors Dataset for Python and on the LineDP [16] Dataset for Java. The transformers were replaced with RoBERTa [6] and CodeT5 [15] and RoBERTa was shown to have given better results.

2.4 LineFlowDP

LineFlowDP is a Deep Learning model working at the source code level, utilizing a given source code's Program Dependency Graph (PDG). Each node in the computed PDG, representing a line of code, is extended to preserve semantic information using data flow and control flow information. A vector representation is learned using Doc2Vec [5], which computes a node embedding. The node embedding and edge embeddings are passed to a Relational Graph Convolutional Network (RGCN) [13], which predicts whether a file is defective or clean. The RGCN and PDG are fed to the graph interpreter GNNExplainer [21] to construct a minimal subgraph, on which SNA [17] methods are applied to achieve risks scores for nodes (code line), which are ranked. LineFlowDP has been experimented for Java projects, cross-project.

3 PROPOSED MODEL: MIFEMODEP

We propose **MiFeMoDEP**, a Hybrid Mixed Feature Model for Defect Prediction. Building upon the ideas of the related works, we propose a file-level classification model, which is passed to LIME to rank the lines. Our model deviates from the others with a mixed feature pathway, where the semantic information and flow information are extracted independently and then mixed before passing it to a classifier.

We also built datasets for line-level testing from the Defectors Dataset. In the Defectors Dataset, each file or commit is given, and a list of defective lines is the target. This is in contrast to the regularly used dataset where lines are individual rows in the dataset.

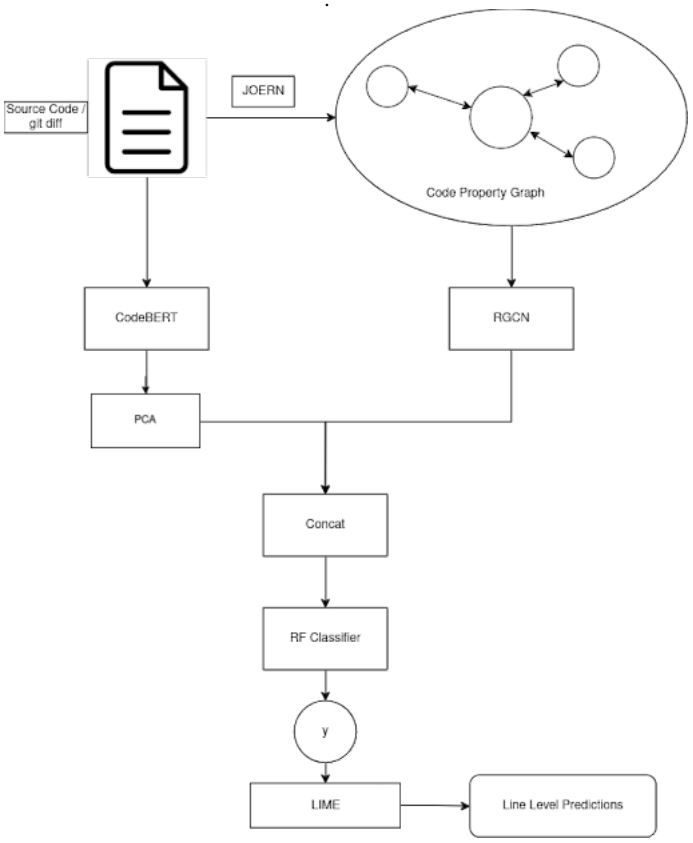
4 DESIGN AND DEVELOPMENT

Slightly different methodologies are proposed for source code and JIT defect prediction.

4.1 Source Code

We use CodeBERT [3] to encode the semantic information of a source code file. Since CodeBERT only accepts a maximum length of 512 tokens at once, we divide the source code tokens into batches of 512, where batch bi and batch bi+1 have 64 tokens in common to preserve the semantic information.

The CodeBERT embeddings are passed to a Principal Component Analysis Model [4] to decompose them to a size of 128. This technique has been used to allow the model to use a reduced but equivalent set of features to run without much hardware limitations.



MiFeMoDeP

JOERN [1] is used to extract a Code Property Graph (CPG) [18] of the source code, which helps glean the flow information of the code. The CPG's nodes contain a DATA dictionary, which is converted to a string to train a Doc2Vec model, to get node embeddings. JOERN currenty supports 20 different edge types. We add one more edge type to accommodate any ones not supported by JOERN.

The encoded nodes and edges are then passed to a Relational Graph Convolutional Model (RGCN) to encode the flow information. RGCNs are used when the edges in the graph are of different types, i.e., the graph is a relational graph.

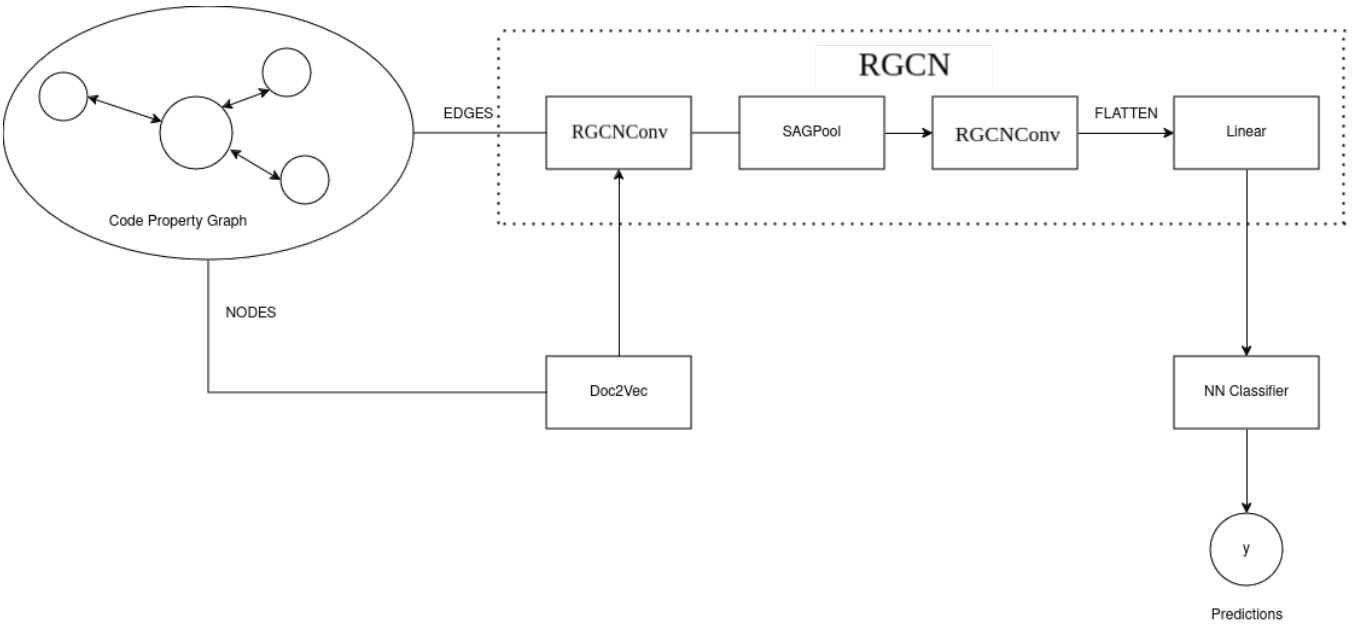
The CodeBERT embeddings and the CPG embeddings are concatenated and passed to a Random Forest Classifier to predict whether the file is buggy or not.

To pass to LIME, we select the files in the dataset, where the predicted and ground truth values of the buggyness of the file is TRUE. A LimeTabularExplainer is created with the training features, and each file in the test dataset, along with the trained Random Forest Classifer are passed to it to extract an explanation.

This explanation has scores for each line, which are then used to rank the lines based on their buggyness.

4.2 JIT

The only difference for JIT defect prediction is that the input diff to the model is preprocessed such that only the source code and the



Training the RCGN

plus/minus characters that represent the addition or removal of a line are present.

First, the RGCN that encodes the CPG is trained with another neural network classifier (Phase I). Transfer Learning [9] is performed where the weights of the RGCN are transferred to MiFeMoDEP, and a Random Forest Classifier is trained using the concatenated CodeBERT and CPG embeddings (Phase II).

5 EXPERIMENTS & RESULTS

JITLine and DeepLineDP have been replicated and tested on the Defectors Dataset, with the hyperparameters all preserved from their original implementations. Bugsplorer has not been replicated due to the fact that we found their explanation of the inputs and outputs to different modules confusing at best and contradictory at worst. In addition to that, the complete code wasn't made publicly available to replicate it. LineFlowDP was made publicly available as recently as February 23, 2024, and hence also wasn't replicated and doesn't have complete code online.

Thanks to the resources we received after the project's initial release, we were able to train MiFeMoDEP much more comprehensively, though the complete dataset was not used due to time limitations.

The following are the metrics calculated: Top-10-Accuracy, Recall@20%Effort, Effort@20%LOC and IFA (Initial False Alarm).

Top-10-Accuracy gives the number of times the where the correct label is among the top 10 labels. A high value is preferred.

Effort@20%LOC measures the amount of effort (measured as LOC) that developers have to spend to find the actual 20% defect-introducing commits divided by the total changed LOC of the whole testing dataset. A low value of Effort@20%Recall indicates that the developers will spend a little amount of effort to find the 20% actual defect-introducing commits.

Recall@20%Effort is the inverse of Effort@20%LOC.

Initial False Alarm measures the number of clean lines that developers need to inspect until the first defective line is found for each file. A low IFA value indicates that few clean lines are ranked at the top, while a high IFA value indicates that developers will spend unnecessary effort on clean lines.

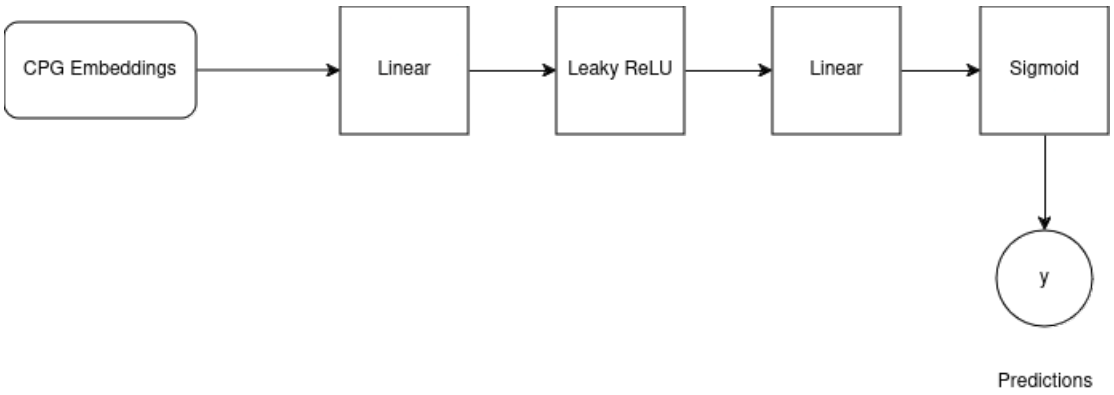
MiFeMoDEP achieves a higher median Top-10-Accuracy, and very similar Recall@20%Effort and Effort@20%LOC, indicating that our model is performing atleast as good as the state-of-the-art models. We achieve a lower IFA which is also very promising. We strongly believe that, given the time, MiFeMoDEP has the potential to perform better than the current SOTA models.

The only drawback for our model is that the inference time is greater compared to JITLine and DeepLineDP since the CPG must be generated.

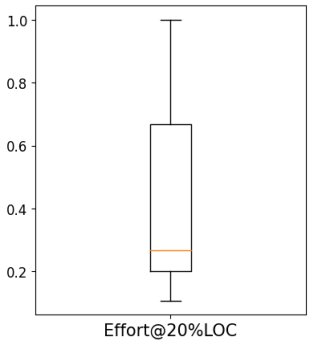
6 CONCLUSION AND FUTURE WORK

In this report, we proposed MiFeMoDEP, a Hybrid Mixed Feature Model for predicting defects at line-level. We then conducted our experimental study by replicating JITLine and DeepLineDP for the Defectors Dataset, and using them as our benchmarks. As discussed in Section 5, MiFeMoDEP performs atleast as good as the benchmarks.

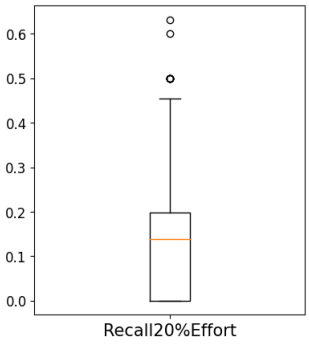
In the future, we aim to develop a Command Line Interface (CLI) for MiFeMoDEP, to allow a smooth integration for software developers and engineers. We also aim to develop a version of MiFeMoDEP which uses Semi-Supervised Learning. This will allow developers to finetune MiFeMoDEP without needing a labeled dataset.



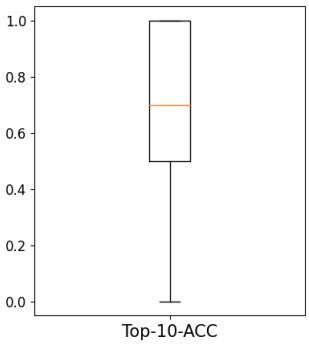
Neural Network Classifier Architecture



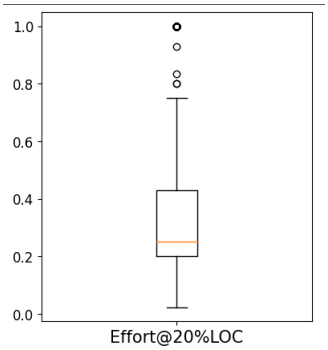
MiFeMoDEP Effort20LOC



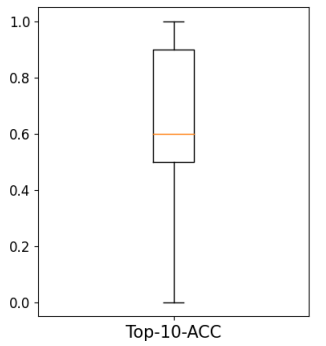
JITLine Recall20effort



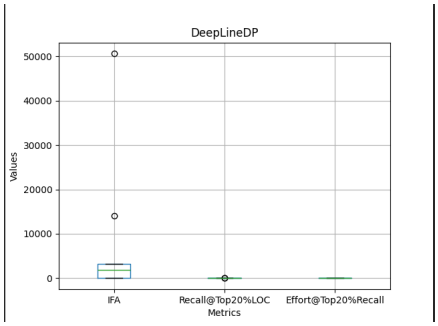
MiFeMoDEP Top 10 Accuracy



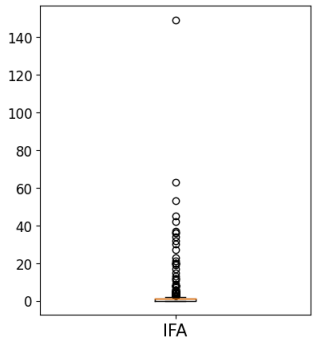
JITLine Effort20LOC



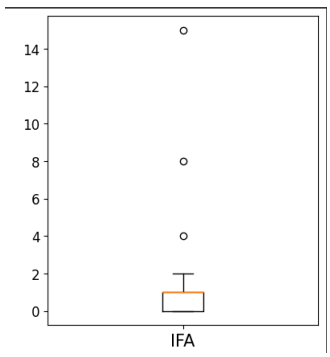
JITLine Top 10 Accuracy



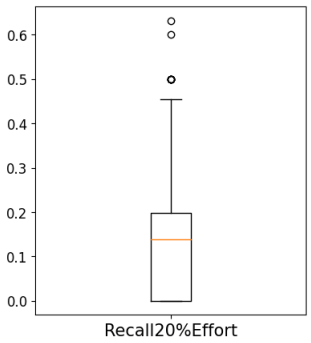
DeepLineDP Results



JITLine IFA



MiFeMoDEP IFA



MiFeMoDeP Recall20effort

REFERENCES

[1] 2024. https://joern.io/ [Online; accessed 24-April-2024].

[2] L.C. Briand, W.L. Melo, and J. Wust. 2002. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Transactions on Software Engineering* 28, 7 (2002), 706–720. https://doi.org/10.1109/TSE.2002.1019484

[3] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL]

[4] Ian Jolliffe and Jorge Cadima. 2016. Principal component analysis: A review and recent developments. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 374 (04 2016), 20150202. https://doi.org/10.1098/rsta.2015.0202

[5] Quoc V. Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. arXiv:1405.4053 [cs.CL]

[6] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv:1907.11692 [cs.CL]

[7] Parvez Mahbub and Mohammad Masudur Rahman. 2023. Predicting Line-Level Defects by Capturing Code Contexts with Hierarchical Transformers. arXiv:2312.11889 [cs.SE]

[8] Parvez Mahbub, Ohiduzzaman Shuvo, and Mohammad Masudur Rahman. 2023. Defectors: A Large, Diverse Python Dataset for Defect Prediction. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE. https://doi.org/10.1109/msr59073.2023.00085

[9] Sinno Jialin Pan and Qiang Yang. 2010. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering* 22, 10 (2010), 1345–1359. https://doi.org/10.1109/TKDE.2009.191

[10] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2021. JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction. arXiv:2103.07068 [cs.SE]

[11] Chanathip Pornprasit and Chakkrit Kla Tantithamthavorn. 2023. DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction. *IEEE Transactions on Software Engineering* 49, 1 (2023), 84–98. https://doi.org/10.1109/TSE.2022.3144348

[12] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. arXiv:1602.04938 [cs.LG]

[13] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2017. Modeling Relational Data with Graph Convolutional Networks. arXiv:1703.06103 [stat.ML]

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL]

[15] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. arXiv:2109.00859 [cs.CL]

[16] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto. 2022. Predicting Defective Lines Using a Model-Agnostic Technique. *IEEE Transactions on Software Engineering* 48, 05 (may 2022), 1480–1496. https://doi.org/10.1109/TSE.2020.3023177

[17] Wikipedia contributors. 2024. Social network analysis — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Social_network_analysis&oldid=1208902947 [Online; accessed 24-April-2024].

[18] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. 590–604. https://doi.org/10.1109/SP.2014.44

[19] Fengyu Yang, Fa Zhong, Guangdong Zeng, Peng Xiao, and Wei Zheng. 2024. LineFlowDP: A Deep Learning-Based Two-Phase Approach for Line-Level Defect Prediction. *Empirical Software Engineering* 29, 2 (23 Feb 2024), 50. https://doi.org/10.1007/s10664-023-10439-z

[20] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. 2016. Hierarchical Attention Networks for Document Classification. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Kevin Knight, Ani Nenkova, and Owen Rambow (Eds.). Association for Computational Linguistics, San Diego, California, 1480–1489. https://doi.org/10.18653/v1/N16-1174

[21] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. GNNExplainer: Generating Explanations for Graph Neural Networks. arXiv:1903.03894 [cs.LG]