MiFeMoDEP: A Hybrid Mixed Feature Model for Defect Prediction at Line Level

Team 11

The GitHub repository for this project can be found here.

Introduction:

Since its conception in 2002 *[1]*, 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.

The lack of a labeled Python defect prediction dataset was satisfied by the Defectors Dataset *[2]*, 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 *[3]* and *Bugsplorer [4]*.

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 *[5]*, and DeepLineDP *[6]*.

Related Works:

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) *[7]* 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.

DeepLineDP is a Deep Learning model working at the source code level, which uses a Hierarchical Attention Network (HAN) *[8]*. 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.

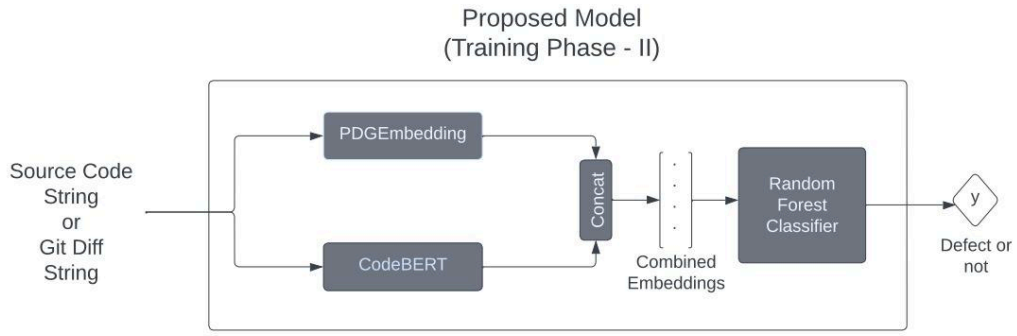
Bugsplorer is an improved version of DeepLineDP where two transformers *[9]* are used in place of the HAN to perform a similar function. The output is an `Lx2` vector, where `L` is the number of lines, and each line is mapped to two values to represent the probability of a bug. Similar to DeepLineDP, 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 *[10]* Dataset for Java. The transformers were replaced with RoBERTa *[11]* and CodeT5 *[12]* and RoBERTa was shown to have given better results.

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 *[13]*, which computes a node embedding. The node embedding and edge embeddings are passed to a Relational Graph Convolutional Network (RGCN) *[14]*, which predicts whether a file is defective or clean. The RGCN and PDG are fed to the graph interpreter GNNExplainer *[15]* to construct a minimal subgraph, on which SNA *[16]* methods are applied to achieve risks scores for nodes (code line), which are ranked. LineFlowDP has been experimented for Java projects, cross-project.

Methodology:

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 a dataset 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.



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

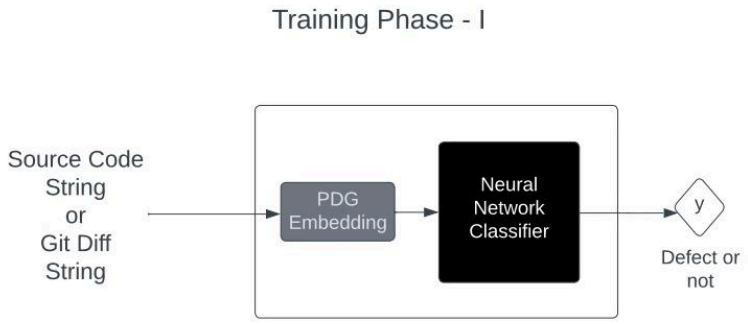
Source Code:

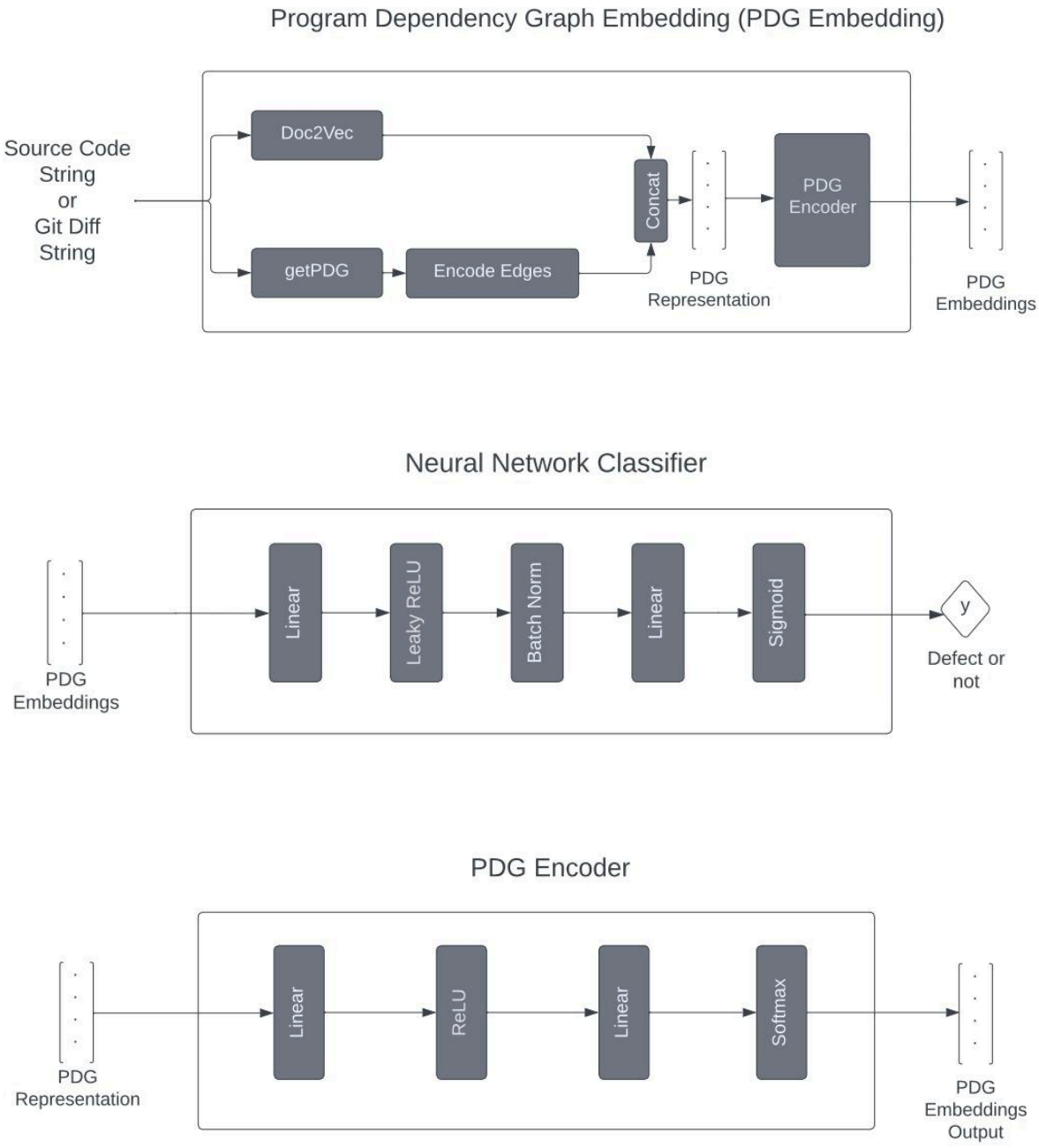
We use CodeBERT *[17]* to encode the semantic information of a source code file. Since CodeBERT only accepts a maximum length of 512 tokens at once, we divide the source code tokens into batches of 512, where batch b_i and batch b_{i+1} have 64 tokens in common to preserve the semantic information.
A primitive PDG is extracted using the AST of the code, and node embeddings are learned using Doc2Vec. A simple neural network encodes the node and edge embeddings. The CodeBERT and PDG embeddings are concatenated and passed to a Random Forest Classifier to predict whether a file is defective or clean.

We intend to use LIME to rank the lines in Release 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 plus/minus characters that represent the addition or removal of a line are present.





First, the neural network that encodes the PDG is trained with another neural network classifier (Phase I). Transfer Learning *[18]* is performed where the weights of the PDG Encoder are transferred to MiFeMoDEP, and a Random Forest Classifier is trained using the concatenated CodeBERT and PDG embeddings (Phase II).

Experiments and 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.

It has to be noted that due to memory and computational limitations, MiFeMoDEP has been trained on a minimal dataset, on a subset of Defectors.

Model Name	AUC	Precision	Recall	F1 Score	Distance to Heaven	False Alarm Rate
JITLine	0.806	0.302	0.674	0.417	0.279	0.222
DeepLineDP	0.651	0.581	0.827	0.682	0.473	0.646
MiFeMoDEP Source Code	0.600	0.823	0.970	0.890	0.718	1.000
MiFeMoDEP JIT	0.712	0.920	1.000	0.958	0.709	1.000

Due to the aforementioned training limitations, the values given here are to be taken with a pinch of salt.

JITLine was trained on the complete dataset as it does not require as much memory as Deep Learning Models do.

DeepLineDP and MiFeMoDEP were trained on the exact same subset, and it can be seen that our model outperforms or at least is as good as the state-of-the-art model (Higher AUC, but distance-to-heaven is not ideal, which implies that the model isn't great → we need to train it on more data). Hence, we can make a strong claim that, given the training on the complete dataset, MiFeMoDEP can outperform all current state-of-the-art models.

Goals for Release 2:

1. Compute line-level rankings as part of MiFeMoDEP and compare them against benchmarks
2. Research the possibility of weighted concatenation of the features extracted (semantic and flow)
3. Extract and encode a full-fledged PDG (there is a severe lack of tools for Python in this domain)
4. Create an API/CLI for project integration
5. Semi-supervised learning to finetune the model, so labeled datasets aren't required (most helpful when you want to implement this model on your codebase, but don't have time to create a labeled dataset)

Team Contributions:

- Chandradithya J — CS21B059 — JITLine Replication, MiFeMoDEP Building, Report Writing
- K E Nanda Kishore — CS21B025 — Preprocessing for all models, Experimentation, README
- A Shree Balaji — CS21B008 — Preprocessing for all models, JITLIne Replication, Experimentation
- Chetan Moturi — CS21B017 — DeepLineDP Replication, Experimentation, PPT
- Karthikeya Maruvada — CS21B033 — DeepLineDP Replication, Experimentation, PPT

Everyone has participated in the literature review.

References:

[1] Assessing the applicability of fault-proneness models across object-oriented software projects

[2] Defectors: A Large, Diverse Python Dataset for Defect Prediction

[3] JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction

[4] Predicting Line-Level Defects by Capturing Code Contexts with Hierarchical Transformers

[5] LineFlowDP: A Deep Learning-Based Two-Phase Approach for Line-Level Defect Prediction

[6] DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction

[7] "Why Should I Trust You?": Explaining the Predictions of Any Classifier

[8] Hierarchical Attention Networks for Document Classification

[9] Attention Is All You Need

[10] Predicting Defective Lines Using a Model-Agnostic Technique

[11] RoBERTa: A Robustly Optimized BERT Pretraining Approach

[12] CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation

[13] Distributed Representations of Sentences and Documents

[14] Modeling Relational Data with Graph Convolutional Networks

[15] GNNExplainer: Generating Explanations for Graph Neural Networks

[16] Social Network Analysis

[17] CodeBERT: A Pre-Trained Model for Programming and Natural Languages

[18] A Survey on Transfer Learning