

https://dspace.org/

Explainable Software Defect Prediction using Word Embeddings and Deep Learning

Phoshoko, Ernest

2025-10
University of the Witwatersrand, Johannesburg
https://hdl.handle.net/10539/49862

Phoshoko, Ernest. (2025). Explainable Software Defect Prediction using Word Embeddings and Deep Learning. [Master's dissertation, University of the Witwatersrand, Johannesburg]. WIReDSpace. https://hdl.handle.net/10539/49862

Downloaded from a DSpace repository. https://dspace.org/

Explainable Software Defect Prediction using Word Embeddings and Deep Learning

Phoshoko Ernest
Supervised by Prof Turgay Celik

A dissertation submitted to the Faculty of Engineering and the Built Environment, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science in Engineering.

Johannesburg, October 2025

Declaration

I declare that this dissertation is my own, unaided work, except where otherwise acknowledged. It is being submitted for the degree of Master of Science in Engineering to the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination to any other university.

Signed this 20th day of October 2025



Phoshoko Ernest

Supervised by Prof Turgay Celik

Abstract

Defects in the source code discovered during or after deployments require resources and effort to resolve. Researchers have developed techniques to tackle the issue of detecting source code defects at an earlier stage. In this study, we examined several convolutional neural network architectures to build a bug detection classifier for Java source code. We utilized Word2Vec, GloVe, and FastText to train embedding models that assist in embedding Java source code in a suitable format for our trained networks. We also explored a few explainable AI methods to provide visual heatmap explanations of the decisions made by our neural networks. With all our models trained under the same conditions and for up to 300 epochs, we identified two top-performing models: ResNet18, with an average accuracy of 77.22%, average precision of 79.17%, average recall of 76.50%, and average F1 score of 77.39% across all embeddings; and MobileNet, with an average accuracy of 84.59%, average precision of 86.04% average recall of 83.69%, and average F1 score of 84.84%. In this study, we used saliency maps, Grad-CAM, Score-CAM, and FIMF score-CAM to generate and analyze heatmaps of Java source code methods, allowing us to analyze and compare the features that the models focus on when making predictions.

Acknowledgements

I would like to thank my supervisor, Professor Turgay Celik, for all the guidance and advice throughout the research journey, from the proposal to the final compilation of the dissertation.

I would like to express my gratitude to my wife and mother for their unwavering support and encouragement throughout the process of completing my dissertation. Additionally, I want to thank my employers for providing me with the resources that made it possible for me to further my studies.

Contents

Declaration **i**

Abstract **ii**

Acknowledgements **iii**

Contents **iv**

List of Figures **viii**

List of Tables **xx**

1 Introduction **2**

1.1 Problem Statement 4

1.2 Research Questions 5

1.3 Research Aims and Objectives 5

1.4 Significance of the research 6

1.5 Summary 6

2 Background and Related Research **7**

2.1 Machine Learning 7

2.2 Artificial Neural Networks 8

2.3 Convolutional Neural Networks 9

2.3.1 Convolutional Layer 10

2.3.2 Pooling Layer 11

2.3.3 Dense Layer 11

2.4 Supervised learning 12

2.5 Abstract Syntax Trees 13

2.6 Methods for Encoding source code 14

2.6.1 Word2Vec 16

2.6.2 GloVe 18

2.6.3 FastText 19

2.6.4 Context2Vec 20

2.6.5 ELMo 20

2.7 Explainable AI 22

2.7.1 Saliency Map 22

2.7.2 Gradient-weighted Class Activation Mapping 23

2.7.3 ScoreCam 23

2.7.4 FIMF score-CAM 24

2.8 Classification Models 25

2.9 Related Research 31

2.10 Summary 34

3 Methodology **35**

3.1 DataSet 35

3.2 Methodology 36

3.2.1 Data Extraction 36

3.2.2 Word Embedding 37

3.2.3 DataSet Transformation 37

3.2.4 CodeXGLUE Dataset 39

3.2.5 Model Training 40

3.3 Classification Models Performance 40

3.4 Explainable AI Visualization 41

3.5 Visualization comparison metrics 44

3.5.1 Fraction of elements above relative threshold 44

3.5.2 Intersection over Union 45

3.6 Process flow 45

3.7 Summary 45

4 Results and Interpretation **47**

4.1 Convolutional Neural Networks Performance 47

4.1.1 1D Basemodel 47

4.1.2 2D Basemodel 51

4.1.3 AlexNet 54

4.1.4 VGG16 58

4.1.5 MobileNet 61

4.1.6 ResNet18 64

4.2 Model Embedding Analysis 68

4.3 Visualization Results 70

4.3.1 Intersection Over Union with Relative Thresholds 71

4.3.2 Fraction of Elements Above Threshold 75

4.3.3 Analysis of Data Threshold 79

4.3.4 Weights Mapping Percentage Thresholds 81

4.4 Results Analysis 85

4.4.1 RQ1 What are the current state-of-the-art methods in the embedded representation of the source code? 85

4.4.2 RQ2 What are the current state-of-the-art machine learning algorithms operating on embedded source code representation for defect detection? 85

4.4.3 RQ3 Which metrics can be used to assess and measure the performance of the machine learning algorithms in source code defect detection? 85

4.4.4 RQ4 From the current state-of-the-art defect detection methods, which one performs best based on these metrics? 86

4.5 Summary 86

5 Conclusion **88**

List of Figures

2.1 Simple one layered feed forward neural network 8

2.2 Simple one layered feed forward neural network [1] 9

2.3 Basic Convolution Network architecture [2] 9

2.4 Convolution layer computation with a kernal of size 3x3 [3] 10

2.5 Max pooling computation [3] 11

2.6 Illustration of a dense layer connected to a flattened output of a convolution layer [3] 12

2.7 Abstract Syntax Tree representation of the function code snippet in Listing 2.1 Figure above 14

2.8 Tokens representation of the code snippet in Listing 2.1. 15

2.9 CBOW and Skip-gram model architecture [4]. CBOW takes input of a one-hot encoded aggregated context vector passed to a neural network with one hidden layer to predict the target word, while Skip-gram takes a target word as input passed through a neural network with a single hidden layer to predict neighboring context words. 17

2.10 CBOW takes input of n-gram aggregated context vector passed to a neural network with one hidden layer to predict the target word [5]. 19

2.11 Context2Vec architecture takes input sentence words from both directions passed to a bidirectional LSTM to output embedding of the context word [6]. 21

2.12 ELMo takes input of n-gram word vectors which are fed through bidirectional LSTM to output a sequence of corresponding embeddings[5]. 21

2.13 Base convolutional network with input shape of 1225x1 26

2.14 Base convolutional network with input shape of 35x35 27

2.15 AlexNet architecture [7] 27

2.16 VGG16 architecture [8] 28

2.17 MobileNet architecture made up of 28 layers with a combination of depthwise separable and pointwise convolution layers [9] 30

2.18 Resnet18 architecture [10] 30

3.1 Example code snippets in the DataSet [11] showing buggy code snipped along with a fix of the code snipped 35

3.2 Represents a histogram of the token dataset lengths of each method from the processed dataset 37

3.3 Represents a histogram of the token dataset lengths of each method from the processed dataset with a filtered list for better visualization 37

3.4 Token representation of code snippet 3.2 38

3.5 GloVe embedded 1225x1 dimension representation of code snippet 3.2 38

3.6 GloVe embedded 35x35 dimension representation of code snippet 3.2 39

3.7 Represents a histogram of the token dataset lengths of each method from the processed CodeXGLUE dataset 39

3.8 Explainable AI Visualization methods for code snipped 3.3 generated by MobileNet with Word2Vec embedding,From left to right is Saliency map ,Grad-CAM,Score-CAM, FIMF score-CAM with a heatmap scale of 0 to 1.0 of intensities 42

3.9 Saliency map representation of a prediction made from a MobileNet with Word2Vec embedding of code snippet in Listing 3.3 42

3.10 Grad-CAM map representation of a prediction made from a MobileNet with Word2Vec embedding of code snippet in Listing 3.3 43

3.11 Score-CAM map representation of a prediction made from a MobileNet with Word2Vec embedding of code snippet in Listing 3.3 43

3.12 FIMF score-CAM map representation of a prediction made from a MobileNet with Word2Vec embedding of code snippet in Listing 3.3 44

3.13 Steps followed to generate visualization heatmaps for java method input to convolutional neural network models 45

4.1 1DBasemodel Word2Vec Model Accuracy vs Epoch 48

4.2 1DBasemodel Word2Vec Model Loss vs Epoch 48

4.3 1DBasemodel Word2Vec Model ROC, AUC score :0.523 48

4.4 1DBasemodel Word2Vec Confusion matrix 48

4.5 1DBasemodel Fasttext Model Accuracy vs Epoch 49

4.6 1DBasemodel Fasttext Model Loss vs Epoch 49

4.7 1DBasemodel Fasttext Model ROC, AUC score : 0.520 49

4.8 1DBasemodel Fasttext Confusion matrix 49

4.9 1DBasemodel GloVe Model Accuracy vs Epoch 49

4.10 1DBasemodel GloVe Model Loss vs Epoch 49

4.11 1DBasemodel GloVe Model ROC, AUC score : 0.522 50

4.12 1DBasemodel GloVe Confusion matrix 50

4.13 2D Basemodel Word2Vec Model Accuracy vs Epoch 51

4.14 2D Basemodel Word2Vec Model Loss vs Epoch 51

4.15 2D Basemodel Word2Vec Model ROC, AUC score :0.528 52

4.16 2D Basemodel Word2Vec Confusion matrix 52

4.17 2D Basemodel Fasttext Model Accuracy vs Epoch 52

4.18 2D Basemodel Fasttext Model Loss vs Epoch 52

4.19 2D Basemodel Fasttext Model ROC, AUC score : 0.530 52

4.20 2D Basemodel Fasttext Confusion matrix 52

4.21 2D Basemodel GloVe Model Accuracy vs Epoch 53

4.22 2D Basemodel GloVe Model Loss vs Epoch 53

4.23 2D Basemodel GloVe Model ROC, AUC score : 0.525 53

4.24 2D Basemodel GloVe Confusion matrix 53

4.25 AlexNet Word2Vec Model Accuracy vs Epoch 55

4.26 AlexNet Word2Vec Model Loss vs Epoch 55

4.27 AlexNet Word2Vec Model ROC, AUC score :0.611 55

4.28 AlexNet Word2Vec Confusion matrix 55

4.29 AlexNet Fasttext Model Accuracy vs Epoch 56

4.30 AlexNet Fasttext Model Loss vs Epoch 56

4.31 AlexNet Fasttext Model ROC, AUC score : 0.659 56

4.32 AlexNet Fasttext Confusion matrix 56

4.33 AlexNet GloVe Model Accuracy vs Epoch 56

4.34 AlexNet GloVe Model Loss vs Epoch 56

4.35 AlexNet GloVe Model ROC, AUC score : 0.637 57

4.36 AlexNet GloVe Confusion matrix 57

4.37 VGG16 Word2Vec Model Accuracy vs Epoch 58

4.38 VGG16 Word2Vec Model Loss vs Epoch 58

4.39 VGG16 Word2Vec Model ROC, AUC score :0.523 59

4.40 VGG16 Word2Vec Confusion matrix 59

4.41 VGG16 Fasttext Model Accuracy vs Epoch 59

4.42 VGG16 Fasttext Model Loss vs Epoch 59

4.43 VGG16 Fasttext Model ROC, AUC score : 0.521 59

4.44 VGG16 Fasttext Confusion matrix 59

4.45 VGG16 Glove Model Accuracy vs Epoch 60

4.46 VGG16 GloVe Model Loss vs Epoch 60

4.47 VGG16 GloVe Model ROC, AUC score : 0.521 60

4.48 VGG16 Glove Confusion matrix 60

4.49 MobileNet Word2Vec Model Accuracy vs Epoch 62

4.50 MobileNet Word2Vec Model Loss vs Epoch 62

4.51 MobileNet Word2Vec Model ROC, AUC score : 0.929 62

4.52 MobileNet Word2Vec Confusion matrix 62

4.53 MobileNet Fasttext Model Accuracy vs Epoch 62

4.54 MobileNet Fasttext Model Loss vs Epoch 62

4.55 MobileNet Fasttext Model ROC, AUC score :0.905 63

4.56 MobileNet Fasttext Confusion matrix 63

4.57 MobileNet GloVe Model Accuracy vs Epoch 63

4.58 MobileNet GloVe Model Loss vs Epoch 63

4.59 MobileNet GloVe Model ROC, AUC score : 0.895 63

4.60 MobileNet GloVe Confusion matrix 63

4.61 ResNet18 Word2Vec Model Accuracy vs Epoch 65

4.62 ResNet18 Word2Vec Model Loss vs Epoch 65

4.63 ResNet18 Word2Vec Model ROC, AUC score : 0.832 65

4.64 ResNet18 Word2Vec Confusion matrix 65

4.65 ResNet18 Fasttext Model Accuracy vs Epoch 66

4.66 ResNet18 Fasttext Model Loss vs Epoch 66

4.67 ResNet18 Fasttext Model ROC, AUC score : 0.871 66

4.68 ResNet18 Fasttext Confusion matrix 66

4.69 ResNet18 GloVe Model Accuracy vs Epoch 66

4.70 ResNet18 GloVe Model Loss vs Epoch 66

4.71 ResNet18 GloVe Model ROC, AUC score : 0.880 67

4.72 ResNet18 GloVe Confusion matrix 67

4.73 t-SNE visualizations of Word2vec embeddings applied on the models ResNet18 , MobileNet and AlexNet in the order from left to right . . 68

4.74 t-SNE visualizations of Fasttext embeddings applied on the models ResNet18 , MobileNet and AlexNet in the order from left to right . . 69

4.75 t-SNE visualizations of GloVe embeddings applied on the models ResNet18 , MobileNet and AlexNet in the order from left to right . . 70

4.76 Represents a histogram of the lengths of each method tokenized before padding it to length 1225 80

4.77 Represents a box and whisker plot of the lengths of each method tokenized before padding it to length 1225 80

4.78 Represents a histogram of the normalized lengths of each method tokenized before padding it to length 1225 80

4.79 Represents a box and whisker plot of the normalized lengths of each method tokenized before padding it to length 1225 80

4.80 Represents a histogram of the MobileNet Fraction of Elements above threshold 0.4 using Score-CAM and Word2Vec embedding 81

4.81 Represents a box and whisker plot of the MobileNet Fraction of Elements above threshold 0.4 using Score-CAM and Word2Vec embedding 81

1 1DBasemodel Fasttext Model Accuracy vs Epoch 98

2 1DBasemodel Fasttext earlystop Model Accuracy vs Epoch 98

3 1DBasemodel Fasttext Model Loss vs Epoch 98

4 1DBasemodel Fasttext earlystop Model Loss vs Epoch 98

5 1DBasemodel Word2Vec Model Accuracy vs Epoch 98

6 1DBasemodel Word2Vec earlystop Model Accuracy vs Epoch 98

7 1DBasemodel Word2Vec Model Loss vs Epoch 99

8 1DBasemodel Word2Vec earlystop Model Loss vs Epoch 99

9 1DBasemodel GloVe Model Accuracy vs Epoch 99

10 1DBasemodel GloVe earlystop Model Accuracy vs Epoch 99

11 1DBasemodel GloVe Model Accuracy vs Epoch 99

12 1DBasemodel GloVe earlystop Model Loss vs Epoch 99

13 2D Basemodel Fasttext Model Accuracy vs Epoch 100

14 2D Basemodel Fasttext earlystop Model Accuracy vs Epoch 100

15 2D Basemodel Fasttext Model Loss vs Epoch 100

16 2D Basemodel Fasttext earlystop Model Loss vs Epoch 100

17 2D Basemodel Word2Vec Model Accuracy vs Epoch 100

18 2D Basemodel Word2Vec earlystop Model Accuracy vs Epoch 100

19 2D Basemodel Word2Vec Model Loss vs Epoch 101

20 2D Basemodel Word2Vec earlystop Model Loss vs Epoch 101

21 2D Basemodel GloVe Model Accuracy vs Epoch 101

22 2D Basemodel GloVe earlystop Model Accuracy vs Epoch 101

23 2D Basemodel GloVe Model Loss vs Epoch 101

24 2D Basemodel GloVe earlystop Model Loss vs Epoch 101

25 AlexNet Fasttext Model Accuracy vs Epoch 102

26 AlexNet Fasttext earlystop Model Accuracy vs Epoch 102

27 AlexNet Fasttext Model Loss vs Epoch 102

28 AlexNet Fasttext earlystop Model Loss vs Epoch 102

29 AlexNet Word2Vec Model Accuracy vs Epoch 102

30 AlexNet Word2Vec earlystop Model Accuracy vs Epoch 102

31 AlexNet Word2Vec Model Loss vs Epoch 103

32 AlexNet Word2Vec earlystop Model Loss vs Epoch 103

33 AlexNet GloVe Model Accuracy vs Epoch 103

34 AlexNet GloVe earlystop Model Accuracy vs Epoch 103

35 AlexNet GloVe Model Loss vs Epoch 103

36 AlexNet GloVe earlystop Model Loss vs Epoch 103

37 VGG16 Fasttext Model Accuracy vs Epoch 104

38 VGG16 Fasttext earlystop Model Accuracy vs Epoch 104

39 VGG16 Fasttext Model Loss vs Epoch 104

40 VGG16 Fasttext earlystop Model Loss vs Epoch 104

41 VGG16 Word2Vec Model Accuracy vs Epoch 104

42 VGG16 Word2Vec earlystop Model Accuracy vs Epoch 104

43 VGG16 Word2Vec Model Loss vs Epoch 105

44 VGG16 Word2Vec earlystop Model Loss vs Epoch 105

45 VGG16 GloVe Model Accuracy vs Epoch 105

46 VGG16 GloVe earlystop Model Accuracy vs Epoch 105

47 VGG16 GloVe Model Loss vs Epoch 105

48 VGG16 GloVe earlystop Model Loss vs Epoch 105

49 MobileNet Fasttext Model Accuracy vs Epoch 106

50 MobileNet Fasttext earlystop Model Accuracy vs Epoch 106

51 MobileNet Fasttext Model Loss vs Epoch 106

52 MobileNet Fasttext earlystop Model Loss vs Epoch 106

53 MobileNet Word2Vec Model Accuracy vs Epoch 106

54 MobileNet Word2Vec earlystop Model Accuracy vs Epoch 106

55 MobileNet Word2Vec Model Loss vs Epoch 107

56 MobileNet Word2Vec earlystop Model Loss vs Epoch 107

57 MobileNet GloVe Model Accuracy vs Epoch 107

58 MobileNet GloVe earlystop Model Accuracy vs Epoch 107

59 MobileNet GloVe Model Accuracy vs Epoch 107

60 MobileNet GloVe earlystop Model Loss vs Epoch 107

61 ResNet18 Fasttext Model Accuracy vs Epoch 108

62 ResNet18 Fasttext earlystop Model Accuracy vs Epoch 108

63 ResNet18 Fasttext Model Loss vs Epoch 108

64 ResNet18 Fasttext earlystop Model Loss vs Epoch 108

65 ResNet18 Word2Vec Model Accuracy vs Epoch 108

66 ResNet18 Word2Vec earlystop Model Accuracy vs Epoch 108

67 ResNet18 Word2Vec Model Loss vs Epoch 109

68 ResNet18 Word2Vec earlystop Model Loss vs Epoch 109

69 ResNet18 GloVe Model Accuracy vs Epoch 109

70 ResNet18 GloVe earlystop Model Accuracy vs Epoch 109

71 ResNet18 GloVe Model Loss vs Epoch 109

72 ResNet18 GloVe earlystop Model Loss vs Epoch 109

73 1DBasemodel Word2Vec Model Accuracy vs Epoch 110

74 1DBasemodel Word2Vec Model Loss vs Epoch 110

75 1DBasemodel Fasttext Model Accuracy vs Epoch 110

76 1DBasemodel Fasttext Model Loss vs Epoch 110

77 1DBasemodel GloVe Model Accuracy vs Epoch 110

78 1DBasemodel GloVe Model Loss vs Epoch 110

79 2D Basemodel Word2Vec Model Accuracy vs Epoch 111

80 2D Basemodel Word2Vec Model Loss vs Epoch 111

81 2D Basemodel Fasttext Model Accuracy vs Epoch 111

82 2D Basemodel Fasttext Model Loss vs Epoch 111

83 2D Basemodel GloVe Model Accuracy vs Epoch 111

84 2D Basemodel GloVe Model Loss vs Epoch 111

85 AlexNet Word2Vec Model Accuracy vs Epoch 112

86 AlexNet Word2Vec Model Loss vs Epoch 112

87 AlexNet Fasttext Model Accuracy vs Epoch 112

88 AlexNet Fasttext Model Loss vs Epoch 112

89 AlexNet GloVe Model Accuracy vs Epoch 112

90 AlexNet GloVe Model Loss vs Epoch 112

91 VGG16 Word2Vec Model Accuracy vs Epoch 113

92 VGG16 Word2Vec Model Loss vs Epoch 113

93 VGG16 Fasttext Model Accuracy vs Epoch 113

94 VGG16 Fasttext Model Loss vs Epoch 113

95 VGG16 Glove Model Accuracy vs Epoch 113

96 VGG16 GloVe Model Loss vs Epoch 113

97 MobileNet Word2Vec Model Accuracy vs Epoch 114

98 MobileNet Word2Vec Model Loss vs Epoch 114

99 MobileNet Fasttext Model Accuracy vs Epoch 114

100 MobileNet Fasttext Model Loss vs Epoch 114

101 MobileNet GloVe Model Accuracy vs Epoch 114

102 MobileNet GloVe Model Loss vs Epoch 114

103 ResNet18 Word2Vec Model Accuracy vs Epoch 115

104 ResNet18 Word2Vec Model Loss vs Epoch 115

105 ResNet18 Fasttext Model Accuracy vs Epoch 115

106 ResNet18 Fasttext Model Loss vs Epoch 115

107 ResNet18 GloVe Model Accuracy vs Epoch 115

108 ResNet18 GloVe Model Loss vs Epoch 115

109 Explainable AI Visualization methods for code snipped 1 generated by AlexNet with Word2Vec embedding,From left to right is Saliency map ,Grad-CAM,Score-CAM, FIMF score-CAM with a heatmap scale of 0 to 1.0 of intensities 116

110 Saliency map representation of a prediction made from a AlexNet with Word2Vec embedding of code snippet in Listing 1 117

111 Grad-CAM map representation of a prediction made from a AlexNet with Word2Vec embedding of code snippet in Listing 1 117

112 Score-CAM map representation of a prediction made from a AlexNet with Word2Vec embedding of code snippet in Listing 1 118

113 FIMF score-CAM map representation of a prediction made from a AlexNet with Word2Vec embedding of code snippet in Listing 1 . . . 118

114 Explainable AI Visualization methods for code snipped 1 generated by ResNet18 with Word2Vec embedding,From left to right is Saliency map ,Grad-CAM,Score-CAM, FIMF score-CAM with a heatmap scale of 0 to 1.0 of intensities 119

115 Saliency map representation of a prediction made from a ResNet18 with Word2Vec embedding of code snippet in Listing 1 119

116 Grad-CAM map representation of a prediction made from a ResNet18 with Word2Vec embedding of code snippet in Listing 1 120

117 Score-CAM map representation of a prediction made from a ResNet18 with Word2Vec embedding of code snippet in Listing 1 120

118 FIMF score-CAM map representation of a prediction made from a ResNet18 with Word2Vec embedding of code snippet in Listing 1 . . 121

List of Tables

4.1 1DBasemodel CNN Bug detection Models Performance trained with different embeddings 48

4.2 CodeXGLUE 1DBasemodel CNN Bug detection Models Performance trained with different embeddings 48

4.3 2D Basemodel CNN Bug detection Models Performance trained with different embeddings 51

4.4 CodeXGLUE 2D Basemodel CNN Bug detection Models Performance trained with different embeddings 51

4.5 AlexNet CNN Bug detection Models Performance trained with different embeddings 54

4.6 CodeXGLUE AlexNet CNN Bug detection Models Performance trained with different embeddings 55

4.7 VGG16 CNN Bug detection Models Performance trained with different embeddings 58

4.8 CodeXGLUE CNN Bug detection Models Performance trained with different embeddings 58

4.9 MobileNet CNN Bug detection Models Performance trained with different embeddings 61

4.10 CodeXGLUE MobileNet CNN Bug detection Models Performance trained with different embeddings 61

4.11 ResNet18 Word2Vec CNN Bug detection Models Performance 64

4.12 CodeXGLUE ResNet18 Word2Vec CNN Bug detection Models Performance 65

4.13 AlexNet CNN Saliency Map Intersection of Elements Above Threshold 71

4.14 AlexNet CNN GradCam Map Intersection of Elements Above Threshold 71

4.15 AlexNet CNN Scorecam Map Intersection of Elements Above Threshold 72

4.16 AlexNet CNN FasterScorecam Map Intersection of Elements Above Threshold 72

4.17 Mobilenet CNN Saliency Map Intersection of Elements Above Threshold 72

4.18 Mobilenet CNN GradCam Map Intersection of Elements Above Threshold 73

4.19 Mobilenet CNN Scorecam Map Intersection of Elements Above Threshold 73

4.20 Mobilenet CNN FasterScorecam Map Intersection of Elements Above Threshold 73

4.21 Resnet18 CNN Saliency Map Intersection of Elements Above Threshold 74

4.22 Resnet18 CNN GradCam Map Intersection of Elements Above Threshold 74

4.23 Resnet18 CNN Scorecam Map Intersection of Elements Above Threshold 74

4.24 Resnet18 CNN FasterScorecam Map Intersection of Elements Above Threshold 74

4.25 AlexNet CNN Saliency Map Fraction of Elements Above Threshold . 75

4.26 AlexNet CNN GradCam Map Fraction of Elements Above Threshold 75

4.27 AlexNet CNN Scorecam Map Fraction of Elements Above Threshold 76

4.28 AlexNet CNN FasterScorecam Map Fraction of Elements Above Threshold 76

4.29 Mobilenet CNN Saliency Map Fraction of Elements Above Threshold 76

4.30 Mobilenet CNN GradCam Map Fraction of Elements Above Threshold 77

4.31 Mobilenet CNN Scorecam Map Fraction of Elements Above Threshold 77

4.32 Mobilenet CNN FasterScorecam Map Fraction of Elements Above Threshold 77

4.33 Resnet18 CNN Saliency Map Fraction of Elements Above Threshold 78

4.34 Resnet18 CNN GradCam Map Fraction of Elements Above Threshold 78

4.35 Resnet18 CNN Scorecam Map Fraction of Elements Above Threshold 78

4.36 Resnet18 CNN FasterScorecam Map Fraction of Elements Above Threshold 78

4.37 AlexNet CNN Saliency Map Weights mapping percentage Threshold 81

4.38 AlexNet CNN GradCam Map Weights mapping percentage Threshold 82

4.39 AlexNet CNN Scorecam Map Weights mapping percentage Threshold 82

4.40 AlexNet CNN FasterScorecam Map Weights mapping percentage Threshold 82

4.41 Mobilenet CNN Saliency Map Weights mapping percentage Threshold 82

4.42 Mobilenet CNN GradCam Map Weights mapping percentage Threshold 82

4.43 Mobilenet CNN Scorecam Map Weights mapping percentage Threshold 83

4.44 Mobilenet CNN FasterScorecam Map Weights mapping percentage Threshold 83

4.45 Resnet18 CNN Saliency Map Weights mapping percentage Threshold 83

4.46 Resnet18 CNN GradCam Map Weights mapping percentage Threshold 83

4.47 Resnet18 CNN Scorecam Map Weights mapping percentage Threshold 84

4.48 Resnet18 CNN FasterScorecam Map Weights mapping percentage Threshold 84

Chapter 1

Introduction

A lot of time and effort has been invested in writing software to solve problems, make money or simplify processes. Some software impacts people's lives, ranging from personal information to financial effects, leading to a requirement to build secure, high-quality software with minimal defects. A significant amount of time must be invested in testing the software to avoid release with defects, studies indicate that around 25%-30% of the development time should be dedicated to this process [12][13]. Different testing processes can be set during the development cycle to catch defects before the software is deployed in a production environment, ranging from writing unit tests, function tests, integration tests, regression tests, and more [14].

Putting all the relevant testing processes in place is beneficial but, in some cases, not feasible. For example, unit testing each software component for all possible results might be close to impossible [15]. Some tiny bugs can still slip through into the production environment, despite all the tests, providing opportunities to research other methods we can use to catch some of the known bugs reported in public git repositories by leveraging machine learning. Combining the different testing processes with a machine learning model that can pick up some of the common bugs in your source code before deploying the software could help increase the software's stability and quality [16].

Machine learning algorithms have been researched and used for their ability to learn and recognize patterns. Some of the real-life technologies which successfully utilize machine learning are image recognition systems, speech recognition, recommendations systems, sentiment analysis, and fraud detection, to list a few. Studies in this field of software defect detection have been split into two main directions [17]. The traditional direction is manually designing ways to represent features effectively to detect defects. The other involves using different machine learning algorithms to

test their performance in detecting defects in source code [17]. This research will focus on utilizing machine learning-based algorithms to solve this task. One crucial requirement for machine learning algorithms is the input. How can we read source code and convert it into a usable format which can be fed into a machine-learning algorithm to create a model?

Several methods have been employed in various research to represent source code in a way that can be utilized as input to machine learning algorithms. A survey was conducted to examine different ways of representing and embedding source code [18]. Some of the methods surveyed were token embedding, function embeddings using abstract syntax trees, and binary code embeddings, among others [18].

We will explore various methods of representing source code and embeddings, and create models to detect bugs in source code. We aim to establish a baseline using token-based source code representation and embeddings, as these offer a more natural form of representation compared to syntax representation, which emphasizes structure.

Token embeddings have been utilized to implement, analyze, and enhance source code using natural language-based machine learning techniques. In a recent research study, token-based embedding was explored to create a machine learning model capable of repairing buggy source code. The results of the study were remarkably good [19].

One of the three options explored for comparing students' source codes on an online university platform was token-based embedding. This option was used to measure the correctness of the solution as well as to conduct statistical analysis on student coding patterns [20].

Abstract Syntax Trees have been used to represent source code for different tasks. Abstract Syntax Tree is a representation of the program level abstract syntax [21]. These trees have been used to detect plagiarism in source code to protect software copyright, and intellectual property [22]. Abstract Syntax Trees provide a starting point for representing plain source code into a consistent structural form. The tree would require some form of vectorized embedding to feed the structural form into some of the powerful machine learning algorithms like Artificial Neural Networks and their variants. Some of the researchers involved in encoding Abstract Syntax Trees [23] used an open-source package to read source code as tokens and built a mapping dictionary to convert tokens to integers which are stored up to form a vector representation. Fan et al. utilized the vector representation to train various machine

learning algorithms, including Random Forests, Deep Belief Networks, Convolutional Neural Networks, and Recurrent Neural Networks, for the purpose of detecting defects in source code.

Exploration of the application of convolutional neural networks to tackle the defect prediction challenge targeting Java source code has been researched [24]. Li et al. [24] used an open-source python package called javalang to parse the java based source code into Abstract Syntax Trees as tokens and built a mapping between the tokens and their integer representations to embed the token as integer vectors. Deep Learning to Find Bugs [25] focuses on using JavaScript code snippets to train a source code bug detector. The source code was then transformed into an Abstract Syntax Tree token with the help of Word2Vec [4] embedding. The token was converted into a vector representation suited to be used as input for machine learning algorithms. Word2Vec is a natural language processing model used for embedding words to vectors.

The use of these embedding techniques was not limited to only the task of defect detection. Nguyen et al. explored using Word2Vec to develop a tool to find similar APIs written in Java and C# with the same usage[26] . The source code is converted into an Abstract Syntax tree to build a sequence that was fed into Word2Vec for embedding.

1.1 Problem Statement

Software bugs can redirect time resources from developing new features to searching for bugs, which may cause the software to be unstable or even unavailable if left alone. A software source code defect detection can help catch and reduce the number of bugs that would have gone into the production environment and assist developers in spending more time on development than bug fixes. We will be focusing on developing models that can detect if a source code contains bugs using machine learning algorithms.

1.2 Research Questions

- What are the current state-of-the-art methods in the embedded representation of the source code?
- What are the current state-of-the-art machine learning algorithms operating on embedded source code representation for defect detection?
- Which metrics can be used to assess and measure the performance of the machine learning algorithms in source code defect detection?
- From the current state-of-the-art defect detection methods, which one performs best based on these metrics?

1.3 Research Aims and Objectives

To answer the research questions, this research aims to survey the available methods for embedding source code and machine learning algorithms used to detect defects in source code, understand the current state-of-the-art methods, and compare their performance.

We will follow the following objectives to achieve the aims of this research:

- Perform a comprehensive literature survey on the embedded representation of source code and machine learning algorithms used to detect defects in code.
- Identify and implement the most successful embedding techniques and machine learning algorithms used to detect defects in source code.
- Identify and implement the most commonly used measures to test the performance of the machine learning algorithms in source code defect detection.
- Compare and analyse the performances of the embedding techniques and machine learning algorithms used in source code defect detection.

1.4 Significance of the research

In most of the literature, machine learning methods and word embedding methods are being used to build models to detect defects in software source code. In the existing literature, there have been no references to using explainability methods such as Grad-CAM to identify the statements which contribute to the model marking the source code as defective. The use of explainability methods to target the statements on a function level containing defects found by our machine learning model will form part of our contribution. Along with exploring the word embedding model's ability to transform raw source code into representations that capture syntactic and semantic relationships among tokens, allowing the models to generalize beyond surface-level code structures, enhancing defect detection models' performance.

1.5 Summary

In this chapter, we explored the problem of software defects and their impact on productivity, along with the efforts taken to resolve the issue. One method discussed involves using machine learning to assist in detecting defects, enabling faster fixes. As part of this research, we will build on the problem of defect detection by implementing some of the state-of-the-art machine learning architectures and utilizing explainability algorithms to clarify the important features these models focus on.

The remaining chapters of the dissertation include the following:

- Background and Related Research
- Methodology
- Results and Interpretation
- Conclusion

Chapter 2

Background and Related Research

The previous chapter went through some research to find a solution for using machine learning to tackle defect detection in source code. This chapter will provide a brief background on the methods, processes, and algorithms used to build models to detect source code defects.

2.1 Machine Learning

Machine learning is a subset of Artificial Intelligence with a focus on algorithms to imitate the ability of humans to learn from experience in the form of data [27]. Some of the supervised and unsupervised learning algorithms that we will be looking at to tackle software defect prediction are based on these machine learning algorithms. Supervised learning is a subset of machine learning that learns a function to map input data to output data [28]. Classification and Regression are two primary types of supervised learning methods [29]. Unlike supervised learning, unsupervised learning aims to understand the relationship and patterns in the data without labeled data. The common types of unsupervised learning methods are Clustering[30] ,Dimensionality Reduction[31] Anomaly Detection Association Rule Learning [32] and Generative Models [33].

2.2 Artificial Neural Networks

This research will focus on machine learning methods based on Artificial Neural Networks due to their proven success in solving simple to complex problems. Artificial Neural Networks are inspired by the human brain's ability to process, compute and learn from observations. Neural networks consist of neurons connected with adjustable weighting functions to build a network to process input [34].

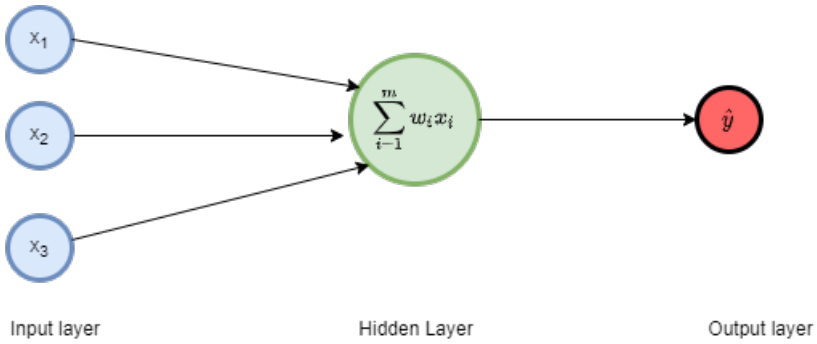


Figure 2.1: Simple one layered feed forward neural network

With each connection of the neurons, there is an associated weight. One common architecture is a feed-forward neural network, where each neuron computes a weighted sum of the inputs, with most cases neural networks having a non-linear activation function applied on the weighted sum [35].

Figure 2.1 represents a fully connected Artificial neural network with a single hidden layer with x_i being inputs to the network,w_i respective connection weights and $\hat{y}$ the network output. The model can be written as

$$\hat{y} = f\left(\sum_{i=1}^{m} w_i x_i\right) \tag{2.1}$$

with function f being the activation function. The study [36] explains Deep neural networks as an extension of the artificial neural network with the number of hidden layer being greater than one and depth of four and above.

Figure 2.2 represents a deep neural network with four layers: one input layer, three hidden layers, and one output layer. The network prediction is modeled by

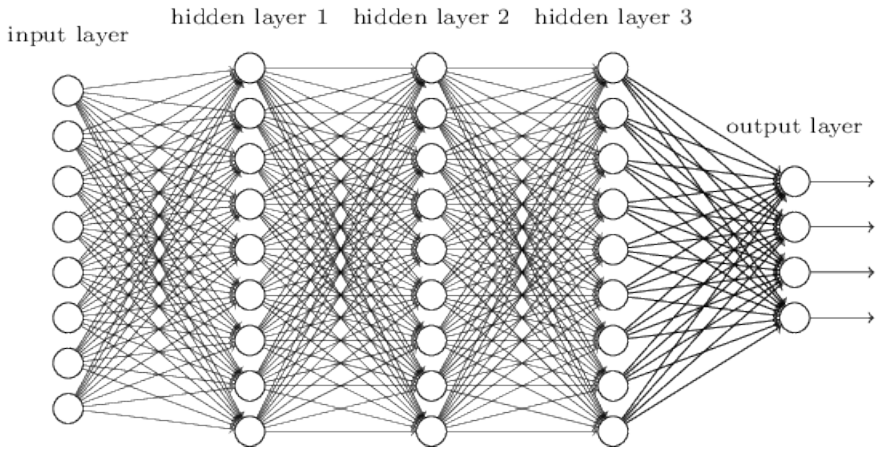


Figure 2.2: Simple one layered feed forward neural network [1]

$$\hat{y}^l = f\left(\sum_{i=1}^{m} w_i^l x_i^{l-1}\right) \tag{2.2}$$

with $l = 1, ..., n$ representing the layer and the hidden layers optimized through the backpropagation algorithm.

2.3 Convolutional Neural Networks

Convolutional Neural Networks are an expansion of deep artificial neural networks inspired by the brain's visual cortex. Convolution Networks were initially widely used for object detection, but studies have increased on the use of these networks for tasks like text recognition, action recognition, speech detection, and more [37].

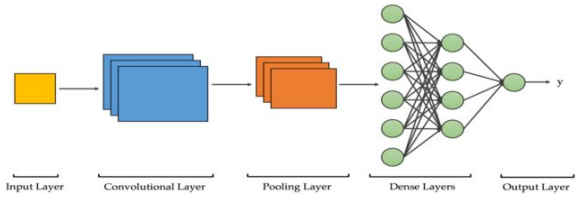


Figure 2.3: Basic Convolution Network architecture [2]

Figure 2.3 represents a basic convolutional neural network architecture with layers

that build basic units of the network. These are the input layer, convolutional layer, pooling layer, dense layer, and output layer.

2.3.1 Convolutional Layer

The convolutional layer is the core layer of a convolution neural network. It contains filters that learn and extract useful features from the input. Most of the computation is done on this layer by stacking feature maps to produce an output volume [37]. Just like the artificial neural networks, these layers are also trained using backpropagation.

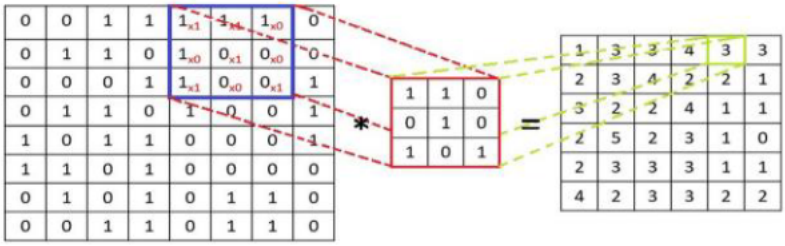


Figure 2.4: Convolution layer computation with a kernal of size 3x3 [3]

The convolution layer operation is given by element-wise multiplication between the input and kernel

$$Activation = Input * Kernal \tag{2.3}$$

$$Activation = \sum_{y=0}^{c} \left(\sum_{x=0}^{r} Input(x,y) * K(x,y) \right) \tag{2.4}$$

where c is the number of columns and r is the number of rows of the input.

2.3.2 Pooling Layer

Pooling layers are an essential component of convolutional neural networks, typically positioned after convolution layers. They are responsible for reducing input dimensions while retaining important information and reducing overfitting.

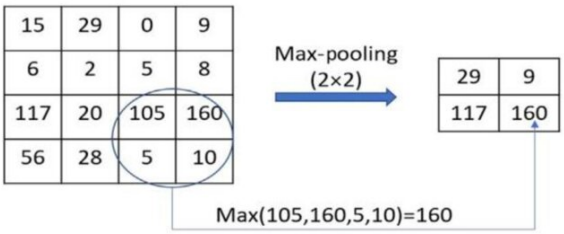


Figure 2.5: Max pooling computation [3]

$$Maxpool = M(m, n) \tag{2.5}$$

$$Maxpool = max_{x=0}^{m-1} m_{y=0}^{n-1} Activation(x, y) \tag{2.6}$$

where m,n are the pool dimensions

2.3.3 Dense Layer

The dense layer, also known as a fully connected layer, connects all neurons to the previous layer, which is flattened into a single one-dimensional vector, similar to a regular neural network. This layer feeds into the output layer, providing results based on the classification task.

$$dense = f\left(\sum_{i=1}^{m} w_i x_i + b_i\right) \tag{2.7}$$

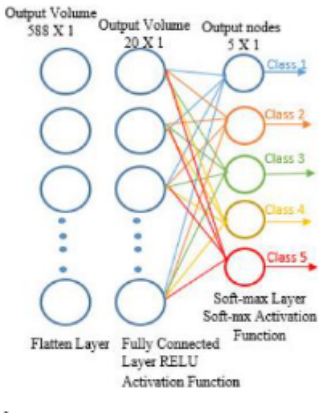


Figure 2.6: Illustration of a dense layer connected to a flattened output of a convolution layer [3]

where b_i represents the offset term.

2.4 Supervised learning

To develop our model, we will concentrate on supervised learning. This type of machine learning establishes the relationship between inputs and outputs in a labeled training dataset, enabling predictions based on new input. This process involves optimizing a loss function using a technique called Backpropagation.

Backpropagation uses gradient descent to adjust the neural network weights by following the following algorithm steps [1]:

1. Given a set of input data $(x, \hat{y})$
2. For each input x:
   - Feedforward computation for hidden layers

     $$z^{x,l} = w^l a^{x,l-1} + b^l$$

     where $a^{x,l} = f\left(\sum_{i=1}^{m} w_i^l x_i^{l-1} + b^l\right)$
   - Calculate Error vector at the output layer

     $$\delta^{x,n} = \frac{\partial J}{\partial a^{x,n}} f'(z^{x,n})$$

- Determine the error for every layer $\delta^{x,l}$

3. Update Weights and bias:

$$w^l = w^l - \eta \sum_x \delta^{x,l} \left(a^{x,l-1}\right)^T$$

2.5 Abstract Syntax Trees

Abstract Syntax Trees(ASTs) are trees aimed to represent the abstract syntactic structure of source code [38]. Compilers widely use these trees due to their ability to represent source code in a tree data structure, which is easier for them to understand. Abstract Syntax Trees are powerful representations that encapsulate the structural organization of source code, regardless of the programming language used [39]. By prioritizing the essential elements that define the code's architecture, ASTs effectively abstract away unnecessary details, such as parentheses in function calls.

```
1
2 function value(x){
3         let x = 42;
4         return x;
5 }
```

Listing 2.1: A code snippet

Code snippet in Listing 2.1 is a sample function that takes in a single parameter x and overrides the value of x with 42 and returns x. Figure 2.7 and Figure 2.8 contain the Abstract Syntax Tree representation of the code snippet in Listing 2.1 in the form of a JSON format and the same code in a token format representation. The token structure is a natural language representation of the code in a list object. The two representations were created using the Esprima[40] tool, which is a JavaScript-based tool hosted online. It can generate the tokenization and syntax trees from JavaScript source code.

```
{
  "type": "Program",
  "body": [
    {
      "type": "FunctionDeclaration",
      "id": {
        "type": "Identifier",
        "name": "value"
      },
      "params": [
        {
          "type": "Identifier",
          "name": "x"
        }
      ],
      "body": {
        "type": "BlockStatement",
        "body": [
          {
            "type": "VariableDeclaration",
            "declarations": [
              {
                "type": "VariableDeclarator",
                "id": {
                  "type": "Identifier",
                  "name": "x"
                },
                "init": {
                  "type": "Literal",
                  "value": 42,
                  "raw": "42"
                }
              }
            ],
            "kind": "let"
          },
          {
            "type": "ReturnStatement",
            "argument": {
              "type": "Identifier",
              "name": "x"
            }
          }
        ]
      },
      "generator": false,
      "expression": false,
      "async": false
    }
  ],
  "sourceType": "script"
}
```


Figure 2.7: Abstract Syntax Tree representation of the function code snippet in Listing 2.1 Figure above

2.6 Methods for Encoding source code

In order to use tokens or an abstract syntax tree as input for machine learning algorithms, we need a way to represent code tokens in a numerical vector form, which is where word embedding comes in. Word embedding involves representing words as numeric vectors. There are two ways to do this: either by manually creating an

```
[
    {
        "type": "Keyword",
        "value": "function"
    },
    {
        "type": "Identifier",
        "value": "value"
    },
    {
        "type": "Punctuator",
        "value": "("
    },
    {
        "type": "Identifier",
        "value": "x"
    },
    {
        "type": "Punctuator",
        "value": ")"
    },
    {
        "type": "Punctuator",
        "value": "{"
    },
    {
        "type": "Keyword",
        "value": "let"
    },
    {
        "type": "Identifier",
        "value": "x"
    },
    {
        "type": "Punctuator",
        "value": "="
    },
    {
        "type": "Numeric",
        "value": "42"
    },
    {
        "type": "Punctuator",
        "value": ";"
    },
    {
        "type": "Keyword",
        "value": "return"
    },
    {
        "type": "Identifier",
        "value": "x"
    },
    {
        "type": "Punctuator",
        "value": ";"
    },
    {
        "type": "Punctuator",
        "value": "}"
    }
]
```


Figure 2.8: Tokens representation of the code snippet in Listing 2.1.

embedding or by using machine learning or deep learning algorithms to learn the representations. In this way, we can effectively embed words in trees and utilize them for machine learning algorithms.

2.6.1 Word2Vec

One of the widely used models in word embedding is Word2Vec. Word2Vec is a two-model architecture proposed by Mikolov et al. [4] and has found many applications in natural language processing tasks. The purpose of word2vec is to compute a continuous vector representation of words from large data sets. Word2Vec presents a way of representing a word as a vector of numbers that can capture the meaning of the word and not just give it random values, producing vector embeddings similar to words that occur in a similar context. Word2Vec utilizes the neural network's ability to learn and model complex non-linear relationships given a specific dataset to produce the word embeddings. Word2vec provides two models Continuous Bag of Words (CBOW) and Skip-gram as shown in Figure 2.9. CBOW is a model that allows you to predict a target word from a given context of words. On the other hand, Skip-gram predicts the context words from a given target.

Continuous Bag of Words (CBOW)

The models purpose is to predict a target word provided a set of context words within a specific window size of a sentence. As shown in Figure 2.1 the model takes an input of context words to predict the center word w(t) in a window size of two, each context word is one-hot encoded vectors with the model optimizing the logarithmic loss function

$$-\frac{1}{T}\sum_{t=1}^{T} w_t.log(p(w_t)) + (1-w_t).log(1-p(w_t)) \tag{2.8}$$

and a softmax activation function was used in the final layer.

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}} \tag{2.9}$$

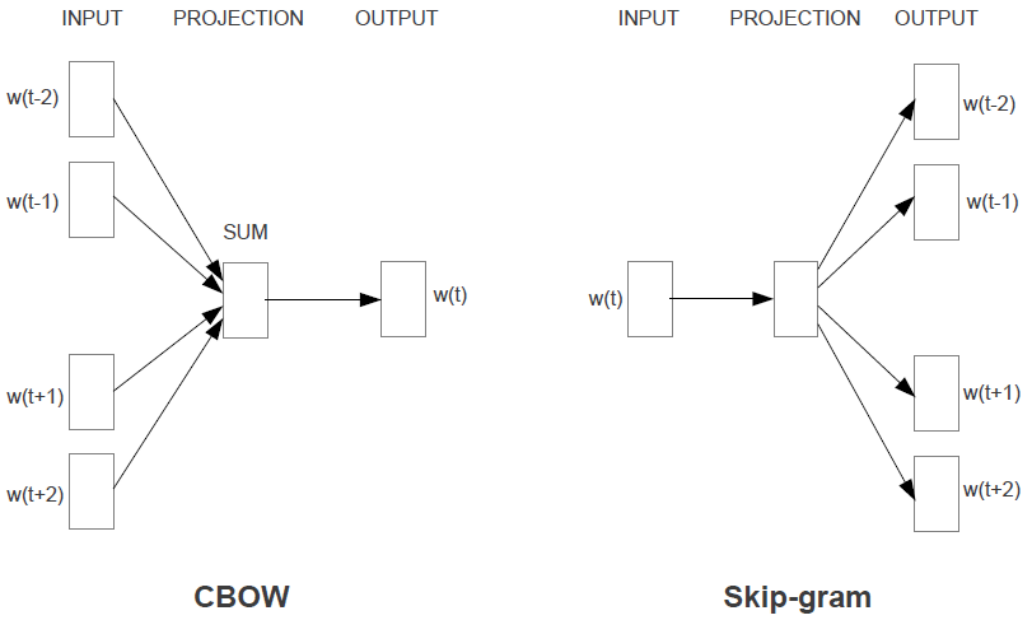


Figure 2.9: CBOW and Skip-gram model architecture [4]. CBOW takes input of a one-hot encoded aggregated context vector passed to a neural network with one hidden layer to predict the target word, while Skip-gram takes a target word as input passed through a neural network with a single hidden layer to predict neighboring context words.

Skip Gram

Skip Gram on the other hand follows the same architecture as CBOW as shown in Figure 2.1 with the difference being the architecture is flipped. The purpose of the model is to predict the context given a word. The model maximizes the average log probability

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0} logp(w_{t+j}|w_t) \tag{2.10}$$

where c is the size of the training context and $p(w_{t+j}|w_t)$ is defined using the softmax function

$$p(w_O|w_I) = \frac{exp(v'^{\top}_{w_O} v_{w_I})}{\sum_{w=1}^{W} exp(v'^{\top}_{w_w} v_{w_I})} \tag{2.11}$$

where v_w and v'_w are the input and output vector representations of w, and W is the number of words in the vocabulary [41].

2.6.2 GloVe

Global Vectors (GloVe) [42] is an extension of the Word2Vec embedding model. While Word2Vec focuses on the local context of the corpus used to train the model, GloVe considers the corpus's global statistical information using a co-occurrence matrix constructed with the count of several times two words occurred in a similar context in the corpus to calculate the probability of co-occurrence.

The model is modeled as a least squares problem to optimize the cost function

$$J = \sum_{i,j=1}^{V} f(X_{ij})(w_i^\top \widetilde{w}_j + b_i + \widetilde{b}_j - logX_{ij})^2 \tag{2.12}$$

where V is the size of the vocabulary, the model is developed by creating a function that represents the ratio of co-occurrence probabilities.

$$F(w_i, w_j, \widetilde{w}_k) = \frac{P_{ik}}{P_{jk}} \tag{2.13}$$

where P_{ik} is the probability that word k appears in the context of word i similar to P_{jk}. To encode the information present in the ratio F is represented as vector differences. And to align with the ratio being a scalar a dot product is taken between the parameters $w_i - w_j \quad and \quad \widetilde{w}_k$ which are vectors

$$F(w_i - w_j, \widetilde{w}_k) = \frac{P_{ik}}{P_{jk}} \tag{2.14}$$

$$F((w_i - w_j)^\top \widetilde{w}_k) = \frac{P_{ik}}{P_{jk}} \tag{2.15}$$

using the property of homomorphism 2.15 is rewritten as

$$F((w_i - w_j)^\top \widetilde{w}_k) = \frac{F(w_i^\top \widetilde{w}_k)}{F(w_j^\top \widetilde{w}_k)} \tag{2.16}$$

From 2.15 and 2.16, we get

$$F(w_i^\top \widetilde{w}_k) = P_{ik} = \frac{X_{ik}}{X_i} \tag{2.17}$$

replacing F with exponential function and applying log on both sides we get

$$w_i^\top \widetilde{w}_k = log(P_{ik}) = log(X_{ik}) - log(X_i) \tag{2.18}$$

$log(X_i)$ is then absorbed into a bias b_i and an additional bias $\widetilde{b}_k$ is added to restore symmetry [42].

$$w_i^\top \widetilde{w}_k + b_i + \widetilde{b}_k = log(X_{ik}) \tag{2.19}$$

applying least square on Equation 2.19 we get the loss function J.

2.6.3 FastText

FastText [43] is another model that extends on the Word2Vec Skip-gram and CBOW model. The FastText model is built to try to combat one of the limitations of Skip-gram, which ignores the morphology of a word by having one vector representation of each word, representing each word as a bag of character n-grams to enrich the model with subword information. FastText has the added benefit of obtaining vector representations of words out of the corpus.

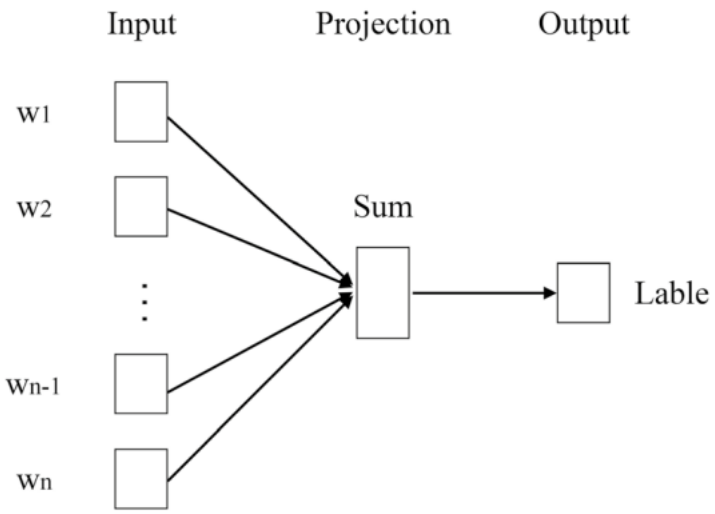


Figure 2.10: CBOW takes input of n-gram aggregated context vector passed to a neural network with one hidden layer to predict the target word [5].

The model in Figure 2.10 takes an input of n-grams $w_1, w_2, ..., w_{n-1}, w_n$ of a word and the embedding is calculated by averaging all the embeddings of the n-gram.An example of a n-gram for the word "example" is $< ex, exa, xam, amp, mpl, ple, le >$.

2.6.4 Context2Vec

Context2Vec [44] is a Neural Network model based on the Word2Vec Continuous Bag of Words (CBOW) architecture, with the difference being Context2Vec uses a bidirectional LSTM instead of Word2Vec's CBOW average word representation.

Figure 2.11 illustrates the architecture of Context2Vec, which consists of a bidirectional LSTM that takes an input sentence $w_{1:n}$ and computes the context embedding of the target word w_i using vector concatenation[6]

$$biLSTM(w_{1:n}, i) = leftLSTM(l_{1:i-1}) \oplus rightLSTM(r_{n:i+1})$$

the output is fed into a Multi Layer Perceptron.

2.6.5 ELMo

Embeddings from Language Models (ElMo) is a state-of-the-art model for representing words in the form of vectors capturing characteristics of the words and how they are used across different contexts [45]. ElMo not only provides a constant representation of the word, but it also captures the meaning of the word given a context, representing words with different meanings as different vectors. ElMo embedding vectors representations are derived using bidirectional Long Short Term Memory (LSTM) networks combined with language models [45]. ElMo takes advantage of the ability of LSTMs to persist information to capture the context in which the words were used.

The bidirectional model computes the forward pass

$$p(w_1, w_2, ..., w_N) = \prod_{k=1}^{N} p(w_k | w_1, w_2, ..., w_{k-1})$$

and backward pass as

$$p(w_1, w_2, ..., w_N) = \prod_{k=1}^{N} p(w_k | w_{k+1}, w_{k+2}, ..., t_N)$$

each layer of the LSTM computes a context representation $\overrightarrow{h}_{k,j}^{LM}$ where $j = 1, 2, ..., L$,

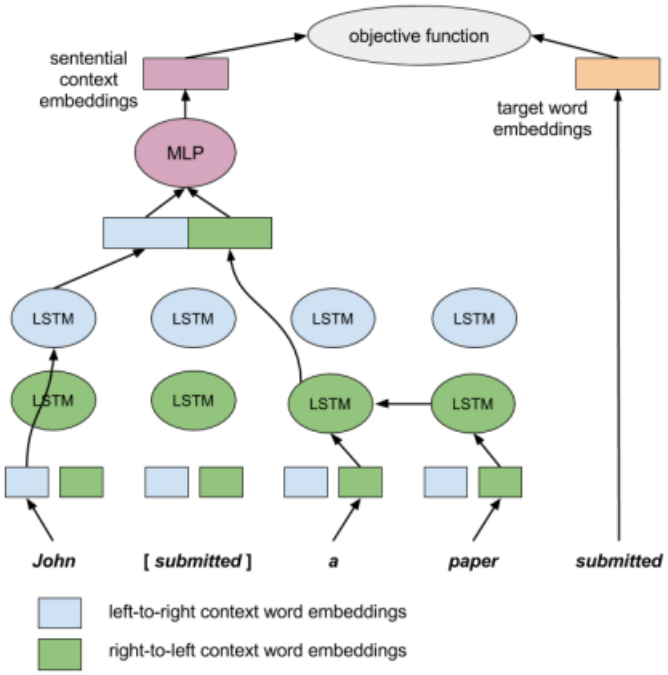


Figure 2.11: Context2Vec architecture takes input sentence words from both directions passed to a bidirectional LSTM to output embedding of the context word [6].

the maximum likelihood of the model is calculated by combining the two passes

$$\sum_{k=1}^{N} = (logp(w_k|w_1, w_2, ..., w_{k-1}) + logp(w_k|w_{k+1}, w_{k+2}, ..., t_N))$$

Provided a sequence of N tokens $w_1, w_2, ..., w_N$ ELMo computes each token representation by summing the intermediate layers of the bidirectional LSTM [5]

$$R_k = \{x_k^{LM}, \overrightarrow{h}_{k,j}^{LM}, \overleftarrow{h}_{k,j}^{LM} | j = 1, 2, ..., L\} = \{h_{k,j}^{LM} | j = 0, 1, ..., L\}$$

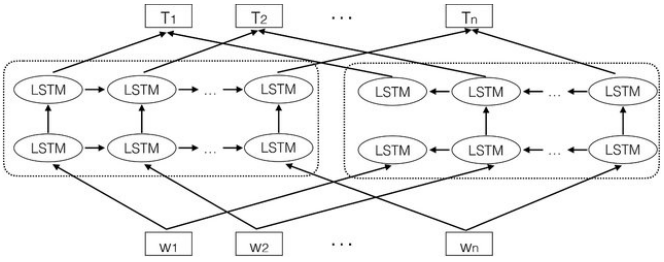


Figure 2.12: ELMo takes input of n-gram word vectors which are fed through bidirectional LSTM to output a sequence of corresponding embeddings[5].

2.7 Explainable AI

The use of black-box deep learning models is rapidly increasing in many domains, thanks to their ability to solve complex problems by learning from provided datasets. However, these deep learning models face challenges because they are less transparent and less explainable.

Explainable AI is a field of research that focuses on how black-box deep learning models can be made transparent and explainable. Explainable AI adds dimensions that make it possible to show which features are most dominant, along with the correlation among the features and why the model predicted the specific outcome[46].

We will be examining various visual explainable AI models such as saliency maps, Gradient-weighted Class Activation Mapping (Grad-CAM), Score-CAM, and Fast Score-CAM as they are designed for convolutional neural networks and focus on highlighting important regions of the input data to provide visual explanations. We will compare the results from each model applied to a few machine learning convolutional neural network architectures.

2.7.1 Saliency Map

Saliency maps are a type of visual explanation that show which input features were used by a machine learning model to make its output decisions. The saliency map is a 2D matrix that matches the size of the analyzed image. Each pixel in the map represents the importance of the corresponding pixel in the input image[47]. This map helps identify the crucial areas of the input that impact the model's final decision.

Saliency maps are among the first methods for attribute visualization, rooted in the principle of backpropagation. The importance of saliency map attribution is calculated by

$$w = \left.\frac{\delta S_c}{\delta I}\right|_{I_0}$$

with I representing the input and $S_c(I)$ is the class prediction score motivated by a

non-linear approximation model

$$S_c(I) \approx w^\top I + b$$

2.7.2 Gradient-weighted Class Activation Mapping

The Class Activation Maps (CAM) method is one of the earliest techniques in the field of visual explainability methods. It involves using global average pooling in convolutional neural networks to generate class activation maps, highlighting important regions by computing a weighted sum of the feature activation values [48].

Gradient-weighted Class Activation Mapping (Grad-CAM) is an enhancement of CAM. Grad-CAM utilizes the gradient information that comes into the final convolutional layer of the CNN to allocate importance values to each neuron for a specific decision of interest [49]. Grad-CAM is used to offer visual explanations of the decisions made by CNN-based models without altering the architecture or re-training the model.

Grad-Cam produces a localization map of important regions of the input provided to a convolutional neural network. The importance is given by the weighted combination of convolutional layer feature maps A^k with the global average pooling weights α_k^c, and finally, a ReLU function is applied to keep only the positive contributions.

$$L^c_{grad-cam} = ReLU(\sum_k \alpha_k^c A^k)$$

where

$$\alpha_k^c = \frac{1}{Z} \sum_i \sum_j \frac{\delta y^c}{\delta A_{ij}^k}$$

i and j represent the width and height dimension and y^c is the prediction class score

2.7.3 ScoreCam

Score-CAM is based on class activation mapping. Unlike previous approaches based on class activation mapping, Score-CAM removes the need for gradients by determining

the weight of each activation map through its forward passing score on the target class [50]. The final result is achieved by linearly combining weights and activation maps. Score-CAM was created to bridge the gap between perturbation-based and CAM-based methods.

Perturbation-based approaches work by perturbing the original input and observing the change in the model's prediction to determine the relationship between the model and the perturbed region [51].

Score-Cam represents the importance of each activation map using Channel-wise Increase of Confidence(CIC). The CIC is calculated by

$$C(A_l^k) = f(XoH_l^k) - f(X_b)$$

where

$$H_l^k = s(Up(A_l^k))$$

with A_l^k representing the k-th channel of layer l , $f(X)$ is the results of the convolutional neural network with input X.Up() is a function that upsamples while s() is a normalization function. Similar to Grad-Cam ReLU faction is applied to keep only the positive contributions resulting in

$$L_{score-cam}^c = ReLU(\sum_k \alpha_k^c A_l^k)$$

where

$$\alpha_k^c = C(A_l^k)$$

2.7.4 FIMF score-CAM

The Fast score-CAM based on local multi-feature integration, is an improvement on the Score-CAM method called FIMF Score-CAM, it integrates multiple features quickly. It directly calculates the feature map's weight using the feature template and integrates various semantic features of the local space to generate a more comprehensive and interpretable heatmap for the object of interest[52].

Fast score-CAM seeks to minimize the computation time of the original score-CAM by avoiding the recalculation of feature extraction while still taking into account the impact of specific regions on the convolutional neural network's output. To integrate the feature maps to obtain information about relevant regions. A denotes the feature

map from the target convolution layer $A = \{A^1, A^2, ..., A^n\}$, and A^k is a selected feature map of the target convolution layer which is used to calculate the template T^k.

$$T^k = \begin{cases} 1 & A^k_{ij} > 0 \\ 0 & A^k_{ij} \leq 0 \end{cases}$$

the CIC is the calculated as

$$C(A^k_l) = f(T^k o A^m) - f(X_b), m \in \{1, ..., n\}$$

$$L^c_{FIMFScore-CAM} = ReLU(\sum_k \alpha^c_k A^k_l)$$

where

$$\alpha^c_k = C(A^k_l)$$

2.8 Classification Models

We will be looking at a couple of convolutional neural network architectures for our classification models. These models will enable us to train models capable of predicting whether a code snippet has a defect.

We will be using these models for our experiment :

- base 1D convolutional neural network
- base 2D convolutional neural network
- AlexNet
- VGG16
- Mobilenet
- Resnet18

1D and 2D base convolutional network

The two architectures share the same structure: an input layer, two convolution layers, two pooling layers, and two dense layers. The key difference lies in the input

layer: the 1D model uses a base input layer with a vector length of 1225, while the 2D model's input is reshaped into a 35x35 shape to mimic an image input.

1D base convolutional network

Layer (type)	Output Shape
conv2d_9 (Conv2D)	(None, 1225, 1, 32)
max_pooling2d_6 (MaxPooling2D)	(None, 1225, 1, 32)
conv2d_10 (Conv2D)	(None, 1225, 1, 64)
max_pooling2d_7 (MaxPooling2D)	(None, 1225, 1, 64)
conv2d_11 (Conv2D)	(None, 1225, 1, 128)
flatten_3 (Flatten)	(None, 156800)
dense_6 (Dense)	(None, 128)
dense_7 (Dense)	(None, 2)

Figure 2.13: Base convolutional network with input shape of 1225x1

The base 1D convolutional network as shown in Figure 2.13 is composed of :

- Convolution layer, maxpool layer ,Convolution layer, maxpool layer
- Two fully connected dense layers
- Output layer contains softmax activation while the rest contain Relu activation

2D base convolutional network

The base 2 D convolutional network as shown in Figure 2.14 is composed of :

- Convolution layer, maxpool layer ,Convolution layer, maxpool layer
- Two fully connected dense layers

Layer (type)	Output Shape
conv2d_12 (Conv2D)	(None, 35, 35, 32)
max_pooling2d_8 (MaxPooling2D)	(None, 17, 17, 32)
conv2d_13 (Conv2D)	(None, 15, 15, 64)
max_pooling2d_9 (MaxPooling2D)	(None, 7, 7, 64)
conv2d_14 (Conv2D)	(None, 5, 5, 128)
flatten_4 (Flatten)	(None, 3200)
dense_8 (Dense)	(None, 128)
dense_9 (Dense)	(None, 2)

Figure 2.14: Base convolutional network with input shape of 35x35

- Output layer contains softmax activation while the rest contain Relu activation

AlexNet

AlexNet is a type of convolutional neural network architecture that performed well in one of the ImageNet Large-Scale Visual Recognition Challenge competitions [53]. Its high-performance results were attributed to the depth of the model's layers. However, this depth also resulted in expensive computation, which was made possible with the help of Graphics Processing Units (GPUs).

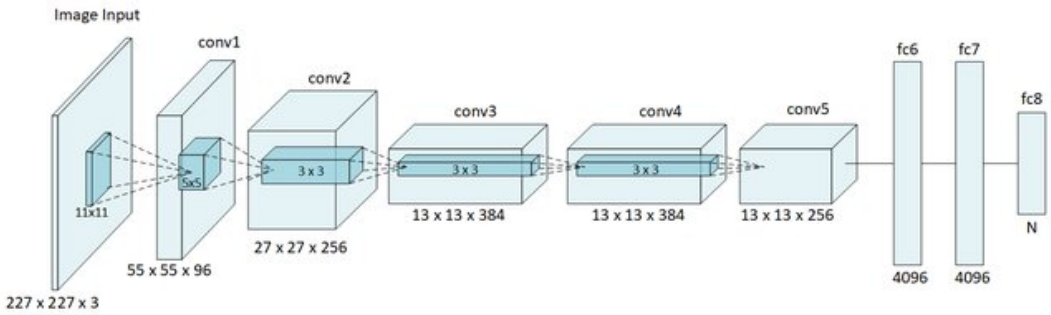


Figure 2.15: AlexNet architecture [7]

The AlexNet architecture as shown in Figure 2.15 is composed of:

- One convolution layer,One maxpool layer,One convolution layer
- Three convolution layers , One maxpool layer ,Three dense fully connected layers
- Output layer contains softmax activation while the rest contain Relu activation

VGG16

VGG is another type of convolutional neural network architecture that achieved high performance in the ImageNet Large-Scale Visual Recognition Challenge competitions. The research explored performance improvements from an architecture with increased depth using very small (3×3) convolution filters [54].

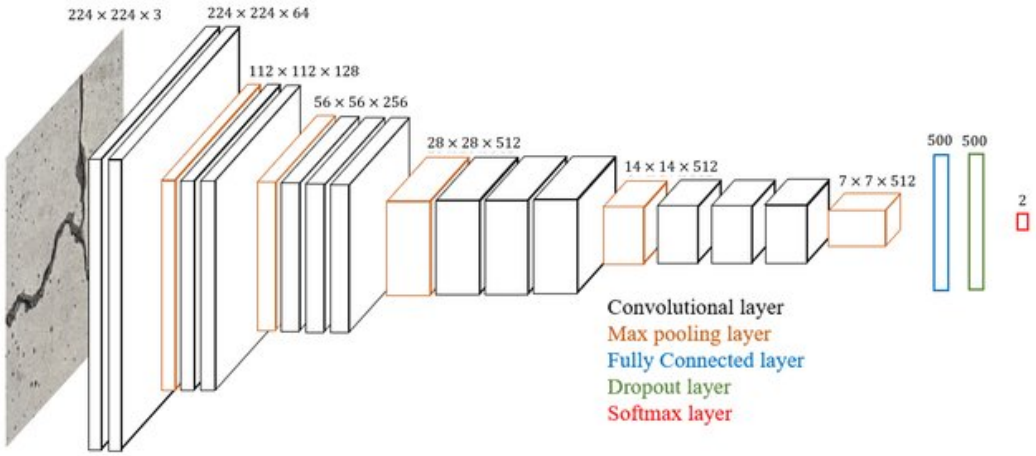


Figure 2.16: VGG16 architecture [8]

The VGG16 architecture as shown in Figure 2.16 is composed of:

- Two convolution layers,One maxpool layer,Two convolution layers
- One maxpool layer , Three convolution layers , One maxpool layer
- Three convolution layers , One maxpool layer , Three convolution layers
- One maxpool layer , Three dense fully connected layers
- Final layer contains softmax activation while the rest contain Relu activation

Mobilenet

MobileNet is a simple yet efficient and not very computationally intensive convolutional neural network for mobile vision applications. MobileNets are based on a streamlined architecture that uses depthwise separable convolutions to construct lightweight deep neural networks[9]. Depthwise separable convolutions are a type of convolution that factorizes the process into two operations: depthwise convolution and pointwise convolution. In depthwise convolution, a single filter is applied independently to each input channel. Then, pointwise convolution combines the outputs from the depthwise convolution across all channels.

The Mobilenet architecture as shown in Figure 2.17 is composed of :

- Convolution layer for the first layer with a stride 2
- Main body composed of Stacked depthwise separable conv blocks
  - depthwise conv
  - batchnorm + ReLU
  - pointwise conv
  - batchnorm + ReLU
- Downsampling layer using stride-2 depthwise convolutions
- Final Layers
  - Global average pooling
  - fully connected layer
  - Softmax activation

Resnet18

A residual neural network is a type of deep learning architecture. In this architecture, the weight layers learn residual functions in relation to the layer inputs. This architecture aims to tackle the problem of degradation, which occurs when the

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Figure 2.17: MobileNet architecture made up of 28 layers with a combination of depthwise separable and pointwise convolution layers [9]

performance of a neural network on training data saturates and then starts to decline as the network's depth increases [55].

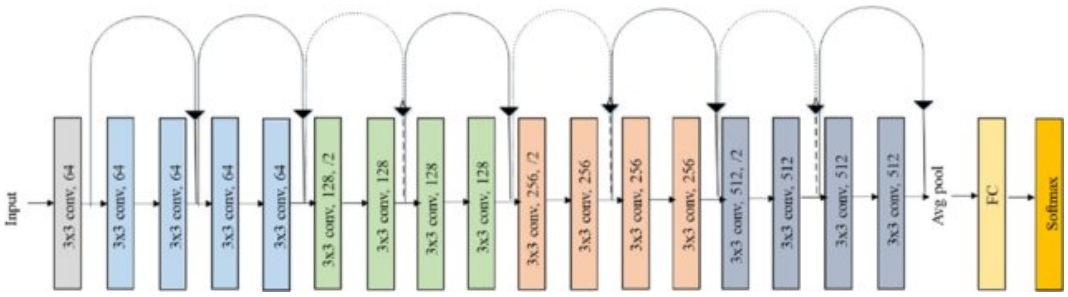


Figure 2.18: Resnet18 architecture [10]

The Resnet18 architecture as shown in Figure 2.18 is composed of :

- Resnet 18 architecture is made up of 18 layers
- With four layers of identical convolution layers

- Final layer contains softmax activation while the rest contain Relu activation

2.9 Related Research

Convolutional Neural Networks (CNNs) are a popular machine learning model used for various tasks, including image recognition, speech recognition, and natural language processing (NLP). In our research, we have defined our problem statement as a form of NLP, where we utilize machine learning to determine whether a snippet of code written in a human-readable language contains defects. In this section, we will examine related studies conducted in this field.

Natural language processing is one of the important areas of research in artificial intelligence, as it aims to connect machine language to human-understandable language. NLP has had a huge impact on systems that understand natural languages and produce speech-to-text, along with systems that generate natural languages such as text-to-speech [56].

Research has demonstrated the capabilities of Natural Language Processing (NLP) in various applications. One notable use is converting voice to text, which assists physically challenged individuals who are interested in coding [57]. In addition, studies have focused on the use of NLP to summarize articles and meeting notes, addressing the challenge of reading long documents to extract key insights or conclusions [58][59].

To contribute to the software failure bug reporting research and reduce the time required for manual bug identification, Tang et al. [60] proposes a feature representation method that combines Latent Dirichlet Allocation (LDA) and Word2Vec. This approach aims to enhance the performance of various machine learning models by improving data representation, which is critical for the input layer.

Tang et al.[60] utilizes LDA to model the topics within bug reports. The results of this modeling yield term frequency (TF) calculations, which serve as a statistical measure of the importance of specific words. Word2Vec is then employed to create word embedding vectors, which are subsequently fed into a convolutional neural network to train a bug-reporting model.

Ngoc et al. conducted a study on the impact of different word embedding methods for software vulnerability detection. The study compared Word2Vec, GloVe, FastText,

and pre-trained GloVe embeddings in combination with four neural network architectures: Bi-LSTM, Hierarchical Attention Network, Text-CNN, and DNN, trained on both synthetic and real-world datasets [61].

JianJie and Le propose using Bidirectional Encoder Representations from Transformers (BERT) in conjunction with ensemble learning to address the task of software code defect detection[62]. This method utilizes Abstract Syntax Trees (AST) and code tokens as input to the BERT embedding model, along with multiple neural network architectures Graph Convolutional Networks (GCN), Convolutional Neural Networks (CNN), and Bidirectional Long Short-term Memory (BiLISM). These networks extract features from the embeddings, which are then passed to a fully connected layer for prediction.

Cross-project defect prediction (CPDP) aims to enhance the quality assurance of software projects by identifying defects in the code[63]. This paper proposes using a Simplified Abstract Syntax Tree (S-AST) to represent source code, alongside leveraging word2vec to create embeddings for the abstract syntax tree tokens (ast-token2vec). For the prediction, a bi-directional long short-term memory (BLSTM) neural network is employed to learn the features of the tokenized tree and make predictions.

The Surface Lexical and Semantic Correlation Matching based on Convolutional Neural Networks (SLS-CNN) [64]framework was proposed to address the gap between source code files and bug reports. This framework leverages the semantic correlation between the two to generate features for a deep neural network. It employs Doc2Vec to train embeddings for the bug reports and Word2Vec to create embeddings for the source code. Additionally, it utilizes the Hadamard product to compute the correlation features between the embeddings.

To address defect prediction, the paper introduces a framework called Defect Prediction via Attention Mechanism[65]. This framework extracts vectors from an Abstract Syntax Tree and employs a conventional approach to map each word to a vector, generating embeddings. These vectors are then input into a Recurrent Neural Network (RNN) to learn semantic features. The learned features are subsequently passed to attention layers, which establish relationships among them, and finally, these features are sent to a fully connected layer to make a prediction.

To address the loss of structural information that occurs when converting Abstract Syntax Trees (ASTs) into integer vectors, the paper proposes the use of three-based encoding with hybrid granularity for defect prediction [66]. A tree-based

word2vec continuous bag-of-words model is utilized to train an embedding for the AST. This embedding is then converted into matrices, which are subsequently fed into a convolutional neural network to extract additional features for the prediction.

As the adoption of black-box models rises, research into explainable AI is also gaining momentum to help clarify aspects of these complex models. In an effort to explain predictions made by object detection models, the paper introduces Detector Randomized Input Sampling for Explanation (D-RISE)[67]. This method produces saliency maps for object detectors, effectively explaining both the localization and classification components of the detection process.

The paper aims to understand and interpret the latent representations of high-level concepts in generative models. It introduces a method for obtaining concept saliency maps [68] to assess the importance of input features in deep generative models, which are a subset of unsupervised learning.

As an attempt to solve the lack of interpretability of existing remote sensing image detection models and the challenges of feature extraction between different networks, Yin et al. proposes an object detection and interpretation model based on gradient-weighted class activation mapping and reinforcement learning to increase detection accuracy [69]. The solution involves the use of the ResNet model for feature extraction, Grad-CAM to generate focus regions, and reinforcement learning to optimize the region generation network.

This study aims to determine the most effective explainable AI method between Grad-CAM and Score-CAM for remote sensing datasets. It evaluates performance using three metrics: Remove and Debias (ROAD), Softmax Information Curve (SIC), and Infidelity[70]. Tasneem and Islam used ROAD to assess the impact of highlighted regions on predictions, SIC to measure how well explanations prioritize informative areas, and Infidelity to evaluate how closely the explanations align with the model's actual behavior. The focus is on understanding which method displays greater reliability and accuracy, utilizing the pre-trained ImageNet model ResNet and training it on the UC-Merced and EuroSAT dataset for classification tasks.

Soomro et al. proposes a method that combines the strengths of two explainable AI techniques, GradCAM++ and Score-CAM, to enhance the interpretability and localization of convolutional neural networks [71]. This approach uses Grad-CAM to generate a coarse heatmap that highlights important regions and using Score-CAM to refine those regions further. The experiments were conducted on the ImageNet dataset using the ResNet, VGG16, and AlexNet pre-trained models.

Yang et al. conducted an analytical study on eleven convolutional neural network(CNN) architectures to classify lung CT images as COVID-19 positive or non-COVID-19. This research used relatively small data sets and evaluated a range of models, including traditional CNNs such as VGG and ResNet, lightweight frameworks such as MobileNet, and contemporary scaling techniques such as EfficientNet [72]. In addition, the study incorporated visualization methods such as saliency maps and Grad-CAM to enhance the interpretability of the model predictions.

The research paper by Giammarco et al. focuses on the diagnosis of colon cancer using explainable AI and proposes a method to detect cancer cells in colon tissues[73]. It examines several state-of-the-art architectures for image classification, including ResNet50, DenseNet, VGG19, Standard CNN, Inception-V3, EfficientNet, and MobileNet. Additionally, the study utilizes techniques such as Grad-CAM, Score-CAM, and FastScore-CAM to generate heatmaps highlighting regions of interest.

2.10 Summary

In this chapter, we provide an in-depth explanation of machine learning and the diverse architectures that have developed from it, along with its foundational elements and the mathematics involved in various computations. We present convolutional neural networks (CNNs) and several architectures they have inspired. We also examine different methods for representing source code and techniques that facilitate embedding these representations in a form suitable for machine learning algorithms, particularly CNNs. Furthermore, we investigate various explainability techniques, including visual heatmaps, to enhance our model transparency. Additionally, we introduce specific convolutional neural network architectures that will be utilized in our research experiments. Lastly, we review existing studies that are relevant to our research area. We acknowledge the work conducted on convolutional neural networks and explainable AI across various fields. However, there is a noticeable gap in the application of explainable AI to source code defect detection.

Chapter 3

Methodology

This research aims to explore different methods and techniques for representing and embedding source code into a format suitable for machine learning algorithms. It will also investigate how to integrate explainable AI to visualize and obtain statistical information about the trained machine learning models.

3.1 DataSet

The data set used in this study is a large java dataset generated by mining millions of open-source projects with multiple bug fixes available in their respective github repositories generated by the author Tufano et al. [11]. The data is sourced from a public GitHub archive, spanning from 2011 to 2017. By analyzing commit messages, a dataset was created that highlights commits with identifiers such as bug, fix, solve, and issues. For each identified commit, a versions of the source code both before and after the fix was collected and filtered for duplicates. The process resulted in a dataset comprising 787,178 Java files that include the buggy methods alongside their corresponding fixed versions.

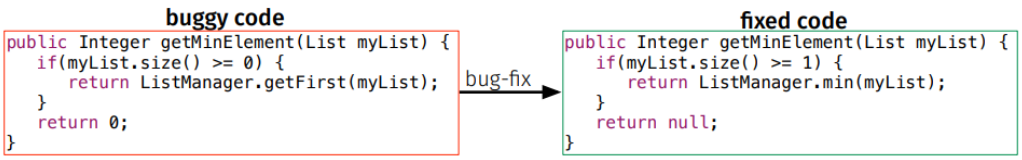


Figure 3.1: Example code snippets in the DataSet [11] showing buggy code snipped along with a fix of the code snipped

To complement the primary dataset, we will also incorporate the refined CodeXGLUE [74] dataset by Tufano et al.. The data set comprises method-level Java bug fix pairs where the methods are normalized by abstracting most variable names, method names, and some of the literals into placeholders to reduce vocabulary size, but in turn also losing some semantics linked to the abstracted variables.

```
1
2 public boolean METHOD_1 ( java.lang.String name ) {
3 TYPE_1 VAR_1 = TYPE_1 . METHOD_2 ( VAR_2 ) ;
4 return ( ! ( METHOD_3 ( name ) ) ) && ( VAR_1 . contains ( name ) ) ;
5 }
```

Listing 3.1: Example abstracted java code snippet from the dataset

3.2 Methodology

3.2.1 Data Extraction

We create an [N, 2] array that contains pairs of methods: the buggy method and its fixed version for each Java project, with N representing the number of java files. Using this array, we navigate through each project directory and load the Java code as strings, which we then store in a list. This list serves as our dataset for training our models.

Next, we convert the source code of each method to tokens. This crucial step is the backbone of our process, as the tokens will be used to train word embedding models along with the classification models, contributing significantly to the overall success of our project.

We limited our dataset to methods with a specific token length to improve model learning and reduce noise that could be introduced through padding to ensure uniform data input lengths. Furthermore, our choice was supported by the analysis of our datasets, as shown in Figures 3.2 and 3.3, which revealed that the majority of our dataset instances fall within the range of 0 to 1000, making this sampling both practical and representative. The choice to use a maximum length of 1225 addresses some outliers in our dataset and creates a complete shape for our 2D reshape.

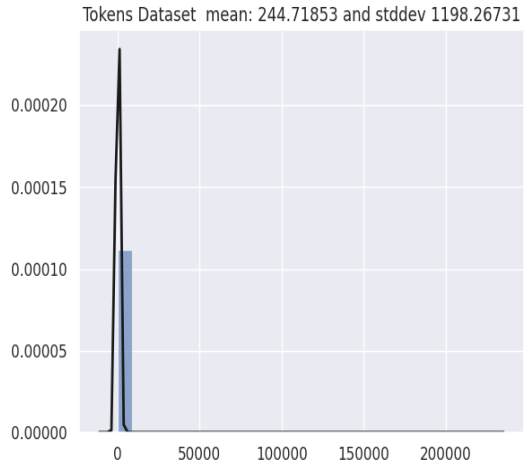


Figure 3.2: Represents a histogram of the token dataset lengths of each method from the processed dataset

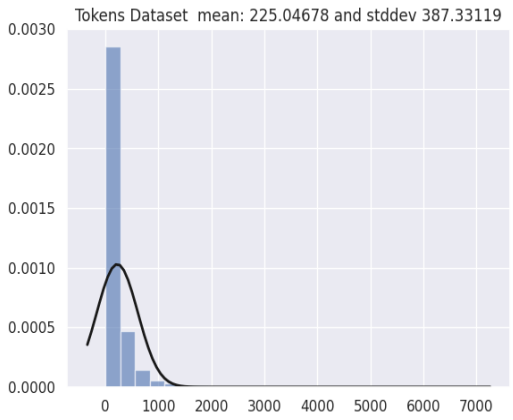


Figure 3.3: Represents a histogram of the token dataset lengths of each method from the processed dataset with a filtered list for better visualization

3.2.2 Word Embedding

We will use Word2Vec, FastText, and GloVe as our word embedding models to generate embedded token vectors. These models will be trained on pre-processed token datasets. We will embed a subset of the dataset generating vectors with a maximum length of 1225. For methods with tokens less than the maximum length, we will pad them with 0.

For the embedding models, we used Python libraries to implement and train our embedding for our data. For the embedding dimensions, we used a low dimension of 1x1 dimension embedding to run our initial experiment using a low dimension for better initial analysis. For Word2vec and Fasttext, we used the Gensim python library [75], for Glove implementation, we used the Mittens python library [76].

3.2.3 DataSet Transformation

We create two datasets derived from our various architectures for training our defect detection model. After embedding, one dataset is transformed into a 1225x1 dimension array for each input, while the other is reshaped into a 35x35 format.

Code snippet 3.2 shows a Java method from our dataset. Figure 3.4 illustrates the code tokens from the Java snippet. Figure 3.5 displays the GloVe embedding of the Java snippet in a 1225x1 array format, while Figure 3.6 shows the GloVe embedding transformed into a 35x35 array.

```
1
2 @java.lang.Override
3 public void visitString(long resultID, java.lang.String value) {
4     strings.put(resultID, value);
5 }
```

Listing 3.2: Example java method code snippet from training dataset

```
['@',
 'java.lang.Override',
 'public',
 'void',
 'visitString',
 '(',
 'long',
 'resultID',
 ',',
 'java.lang.String',
 'value',
 ')',
 '{',
 'strings.put',
 '(',
 'resultID',
 ',',
 'value',
 ')',
 ';',
 '}']
```

Figure 3.4: Token representation of code snippet 3.2

```
[[-0.00728832]
 [-0.00582588]
 [-0.01656189]
 ...
 [ 0.        ]
 [ 0.        ]
 [ 0.        ]]
```

Figure 3.5: GloVe embedded 1225x1 dimension representation of code snippet 3.2

```
[[-0.00728832 -0.00582588 -0.01656189 ...  0.         0.
  0.        ]
 [ 0.         0.         0.         ...  0.         0.
  0.        ]
 [ 0.         0.         0.         ...  0.         0.
  0.        ]
 ...
 [ 0.         0.         0.         ...  0.         0.
  0.        ]
 [ 0.         0.         0.         ...  0.         0.
  0.        ]
 [ 0.         0.         0.         ...  0.         0.
  0.        ]]
```

Figure 3.6: GloVe embedded 35x35 dimension representation of code snippet 3.2

3.2.4 CodeXGLUE Dataset

The same methodology will be followed in with the CodeXGLUE dataset as listed in Subsections 3.2.1,3.2.2 and 3.2.3, with the difference in the input dimensions based on the structure and size of the dataset as can be seen in Figure 3.7 which shows how the dataset is represented in terms of token length which is why the input dimension is set to maximum length of 121 to cover for all the dataset.

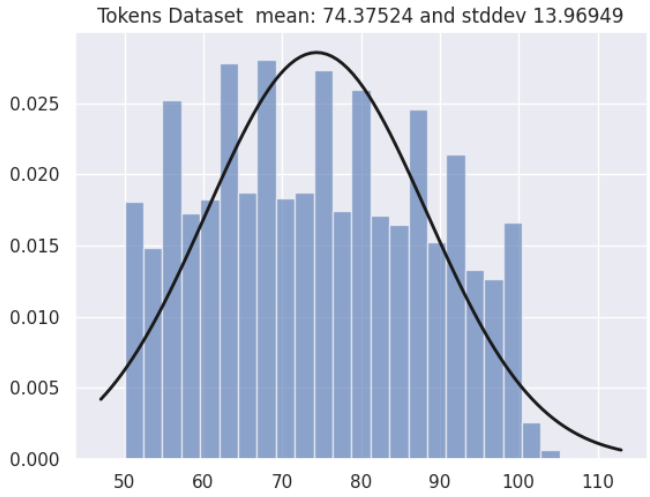


Figure 3.7: Represents a histogram of the token dataset lengths of each method from the processed CodeXGLUE dataset

3.2.5 Model Training

All of our models listed in Chapter 2.8 are implemented using the Python Keras library, with the dataset split into 80% training and 20% validation. All the models are trained for a maximum of 300 epochs using the backpropagation algorithm and a seed value of 10 to ensure reproducibility of the results. To ensure a fair comparison across the model configurations, each configuration is given equal training exposure, allowing us to assess their true learning capacities under controlled conditions. The hyperparameters used for training all of the models where a batch size of 1024 to balance gpu utilization and training speed based on the large size of our data, we also used the Keras library Adadelta optimizer with a learning rate of 0.001 for a more stability on our large dataset we also used a modelcheckpoint monitoring validation accuracy to prevent overfitting and save best performing model. For the training infrastructure, we used various runtimes from Google Colab.

3.3 Classification Models Performance

To evaluate the effectiveness of our classification models, we will be using at the following metrics:

- $Recall = \frac{TP}{TP+FN}$
- $Precision = \frac{TP}{TP+FP}$
- $F_score = \frac{2*Recall*Precision}{Recall+Precision}$

Where TP = True Positives, FP = False Positives, FN = False Negatives, TN = True Negatives.

3.4 Explainable AI Visualization

For our visualizations, we will utilize the methods outlined in Chapter 2.7. The code snippet 3.3 will illustrate the corresponding heatmaps generated using MobileNet input with Word2Vec embedding to highlight the significance of each input through Saliency maps, Grad-CAM, Score-CAM, and FIMF score-CAM, as demonstrated in Figure 114. Figure 115 shows a Saliency map with the tokens embedded in the image. In contrast, Figure 116 presents a Grad-CAM heatmap with embedded tokens, Figure 117 displays Score-CAM heatmap with tokens embedded, and Figure 118 shows the FIMF score-CAM heatmap with tokens embedded in the image. To create the heatmap visualizations, we processed the results from the explainability methods along with the original input, making sure to remove any padding beyond the appropriate dimensions relative to the original input size. We then applied a function to reshape the data into dimensions of 15x15, 25x25, and 35x35 for better visualization.

```
1
2 @java.lang.Override
3 public void handle(javafx.scene.input.MouseEvent mouseEvent) {
4     picture.setDescription(editDescTxt.getText());
5     descTxt.setText(editDescTxt.getText());
6     content.getChildren().remove(vbox);
7     content.getChildren().add(5, descEditBtn);
8     IHM.utils.Dialogs.showInformationDialog( " Description modifiee
          avec success !" );
9 }
```

Listing 3.3: Example java code snippet from training dataset

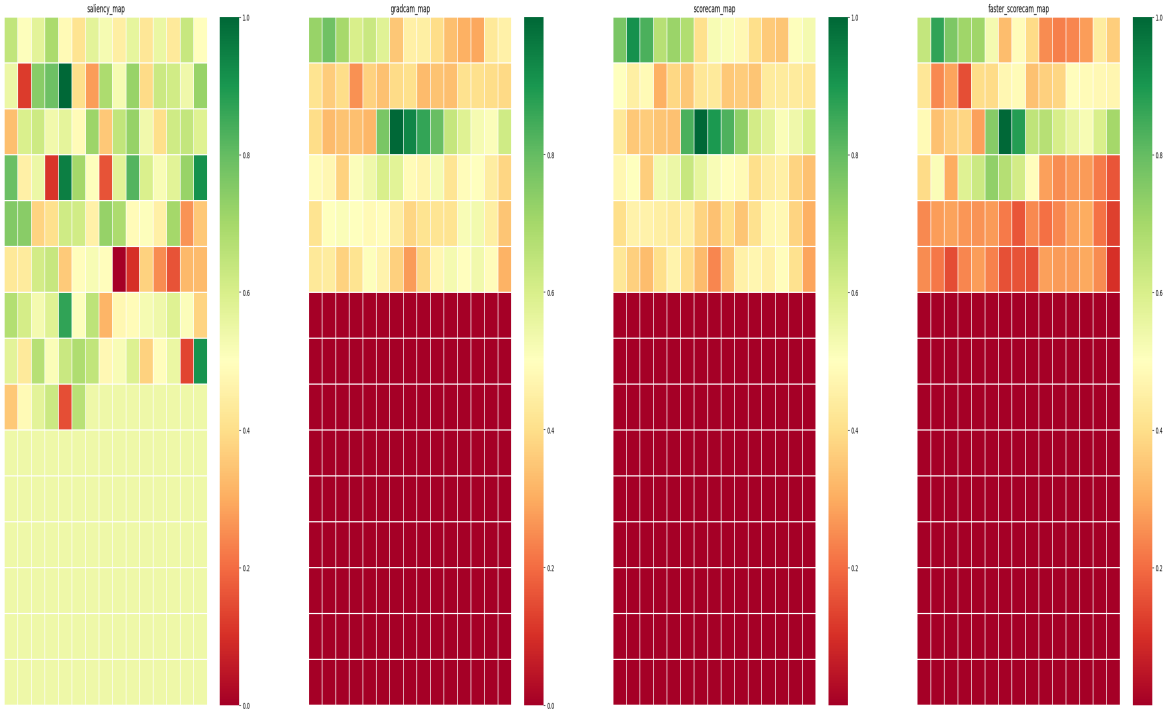


Figure 3.8: Explainable AI Visualization methods for code snipped 3.3 generated by MobileNet with Word2Vec embedding,From left to right is Saliency map ,Grad-CAM,Score-CAM, FIMF score-CAM with a heatmap scale of 0 to 1.0 of intensities

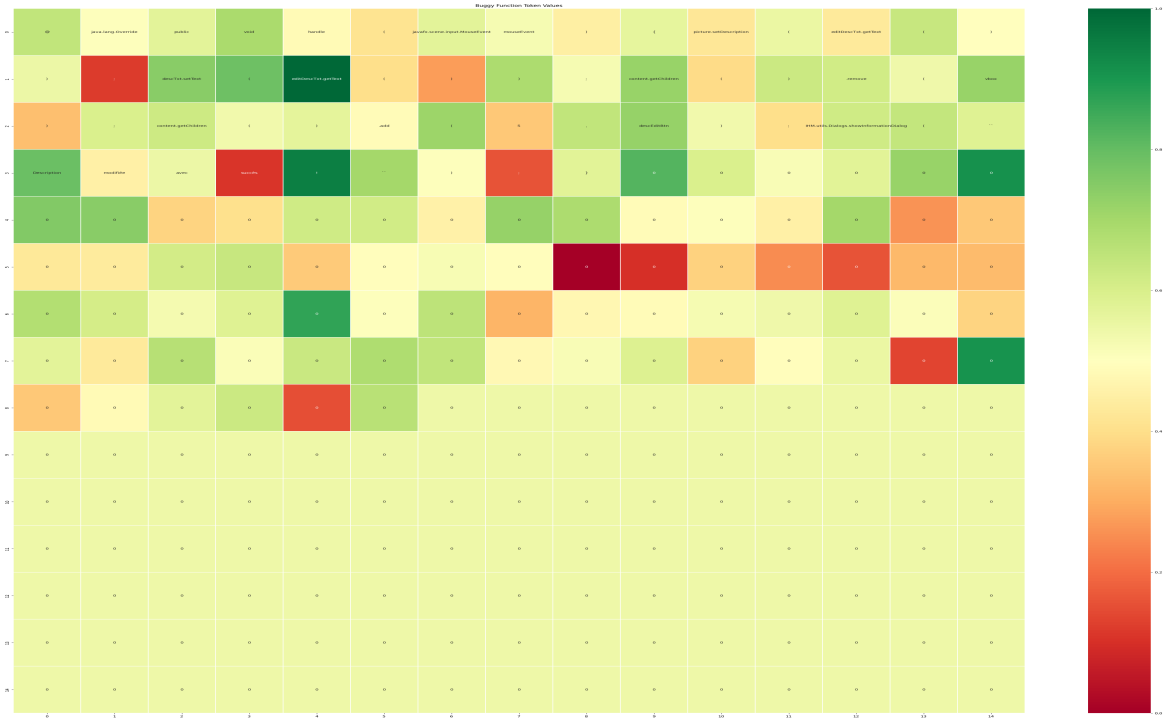


Figure 3.9: Saliency map representation of a prediction made from a MobileNet with Word2Vec embedding of code snippet in Listing 3.3

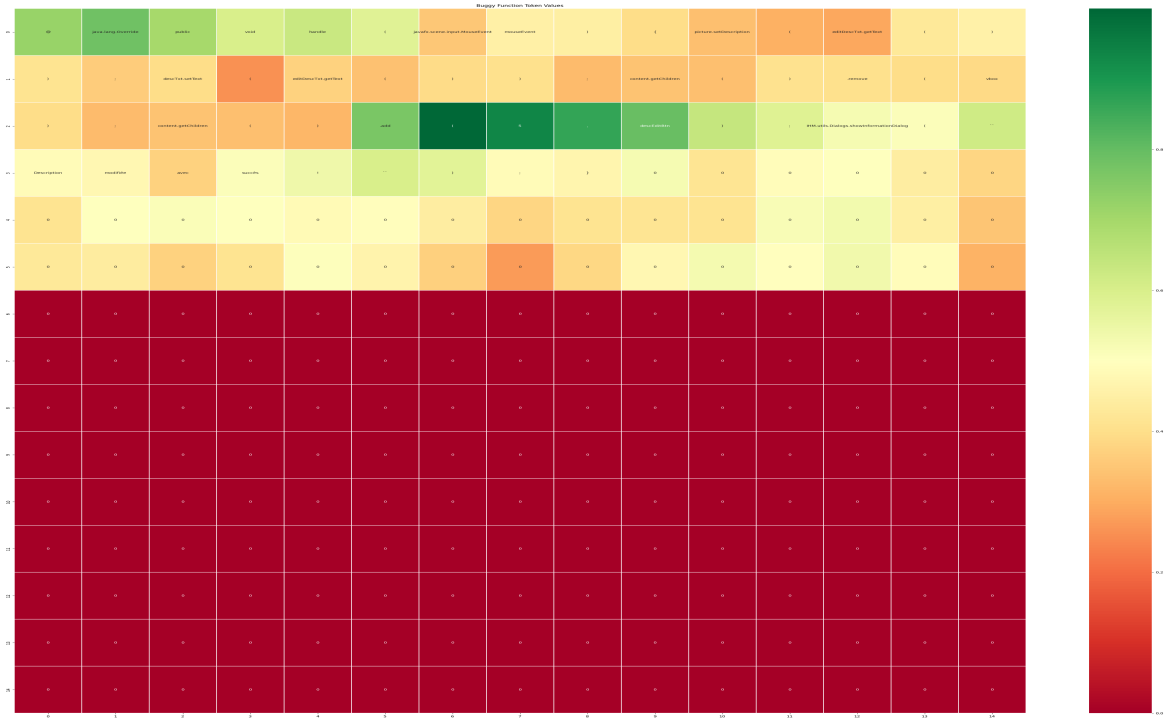


Figure 3.10: Grad-CAM map representation of a prediction made from a MobileNet with Word2Vec embedding of code snippet in Listing 3.3

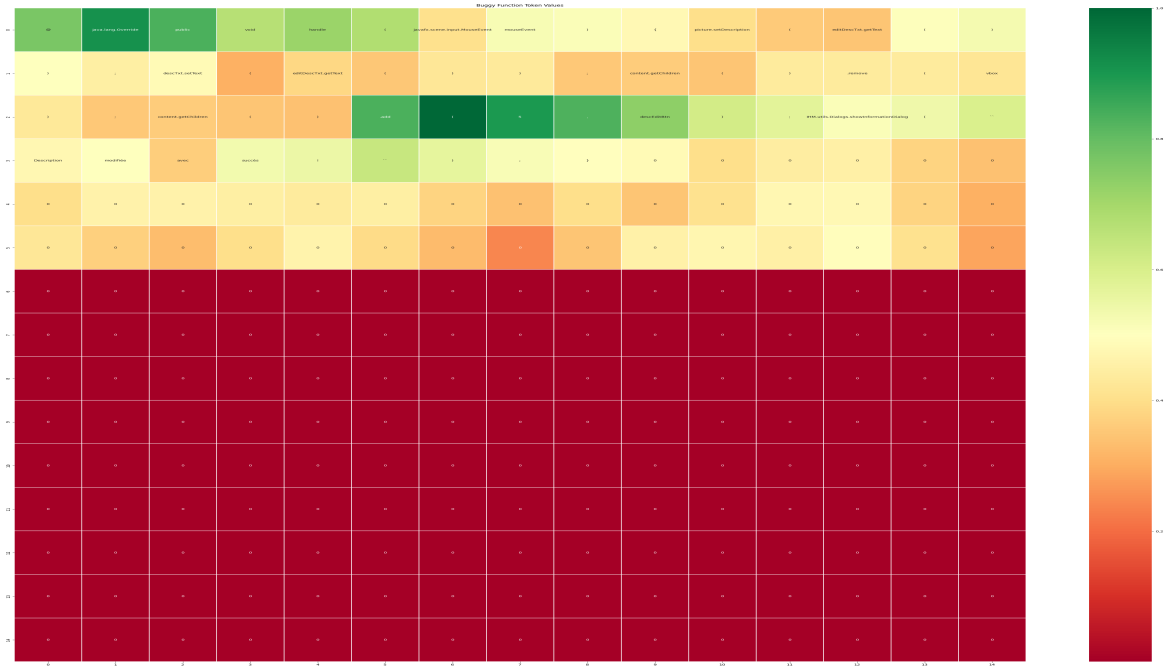


Figure 3.11: Score-CAM map representation of a prediction made from a MobileNet with Word2Vec embedding of code snippet in Listing 3.3

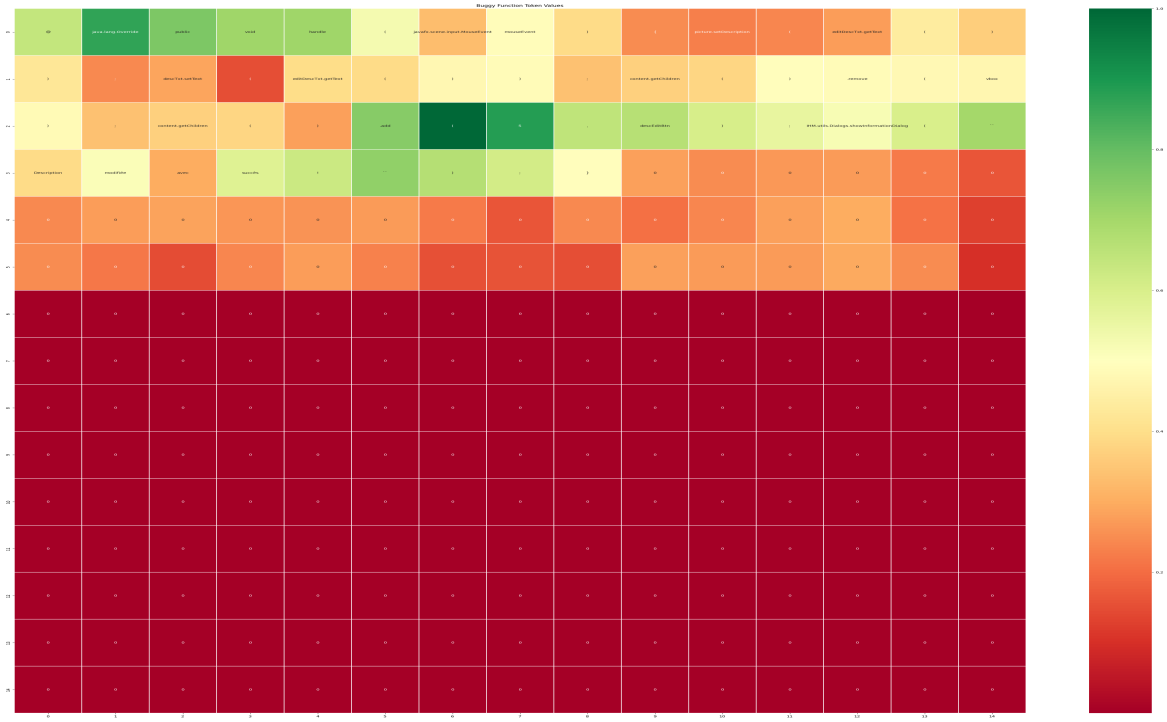


Figure 3.12: FIMF score-CAM map representation of a prediction made from a MobileNet with Word2Vec embedding of code snippet in Listing 3.3

3.5 Visualization comparison metrics

To augment our analysis of the various explainable ai models we will be using the two comparison metrics introduced by Górski et al. [77].

3.5.1 Fraction of elements above relative threshold

The fraction of elements above the relative threshold $F(v, t)$is used to calculate the amount of attention the CNN pays to its input. This fraction is defined as the number of elements in a vector v that exceed a given relative threshold t, multiplied by the maximum value of the vector and divided by the length of the vector, where v is a heatmap produced by an explainable AI model.

3.5.2 Intersection over Union

Intersection over Union (IoU) $I(v_1, v_2, t_1, t_2)$ helps compare the predictions of two different models based on the same input. It measures whether the two models focus on the same parts of the input when generating their predictions. The metric takes an explainable AI model heatmap of two inputs (v_1 and v_2), binarizing them using relative thresholds ($t_1$and t_2), and then calculating the standard intersection over union. This metric quantifies the extent of overlap between the words deemed important for prediction by each model.

3.6 Process flow

The Figure 3.13 represents the process followed to generate results. First, pre-processing converts the source code method snippet into tokens. Next, the tokens are converted into embeddings using our selected word embedding models. Then, the classification step includes the six models listed above Section 2.8. The final step is a visualization layer using the following explainability methods: saliency map, grad-cam, score cam and fast cam.

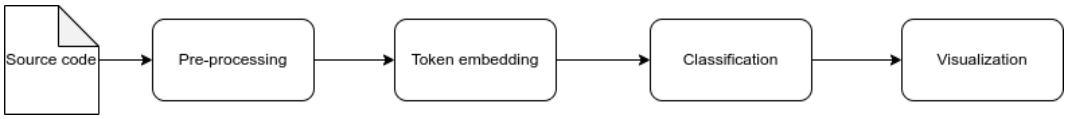


Figure 3.13: Steps followed to generate visualization heatmaps for java method input to convolutional neural network models

3.7 Summary

In this chapter, we explored the dataset used in this research, detailing the exact processes we followed to extract and transform the raw data to meet our specific requirements. We present the various input dimensions that will be used to train our convolutional neural network architectures. Additionally, we outline the evaluation

methods that will assess the effectiveness of these models. Furthermore, we examined a range of explainable AI visualizations designed to enhance the interpretability of our results, providing valuable insights into the decision-making processes of the neural networks and contributing to a deeper understanding of their functionality within the context of our research. The methodology choices help shape the insights generated by this research by integrating pre-processing, evaluation, and interpretability methods to create a framework that not only evaluates the model performance but also interrogates how the models arrive at particular predictions. This frame of reference positions the results not simply as outputs of computation, but as contributions to a deeper understanding of neural network behavior in the context of our research.

Chapter 4

Results and Interpretation

In this section, we will present the results obtained from training the model described in Section 2.8 on the main Java dataset and the complementary CodeXGLUE dataset [11]. Each model will be trained using data embedded with our selected embedding models: Word2Vec, FastText, and GloVe. For each model, we will evaluate its performance based on accuracy, precision, recall, F1 score, and AUC score. Additionally, we will analyze the results generated by visualization techniques, including saliency maps, Grad-CAM, Score-CAM, and FIMF Score-CAM.

4.1 Convolutional Neural Networks Performance

4.1.1 1D Basemodel

This section contains model performance results from training our 1D Basemodel on our primary dataset, as well as the CodeXGLUE dataset. The models are trained until the maximum epoch of 300, with the results of the models trained till convergence based on the validation accuracy, moved to Appendix A for the CodeXGLUE and Appendix B for the primary dataset.

1DBasemodel	Accuracy	Precision	Recall	F1 Score
Word2Vec	0.5169	0.4031	0.5225	0.4551
Fasttext	0.5004	0.9985	0.5004	0.6667
GloVe	0.5165	0.4105	0.5216	0.4594

Table 4.1: 1DBasemodel CNN Bug detection Models Performance trained with different embeddings

1DBasemodel	Accuracy	Precision	Recall	F1 Score
Word2Vec	0.5208	0.5141	0.5211	0.5176
Fasttext	0.5207	0.4930	0.5219	0.5070
GloVe	0.5169	0.7695	0.5112	0.6143

Table 4.2: CodeXGLUE 1DBasemodel CNN Bug detection Models Performance trained with different embeddings

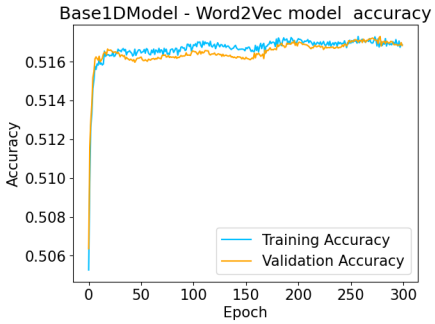


Figure 4.1: 1DBasemodel Word2Vec Model Accuracy vs Epoch

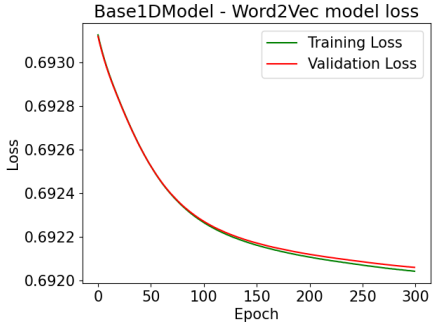


Figure 4.2: 1DBasemodel Word2Vec Model Loss vs Epoch

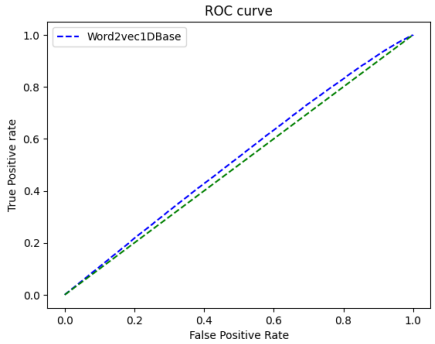


Figure 4.3: 1DBasemodel Word2Vec Model ROC, AUC score :0.523

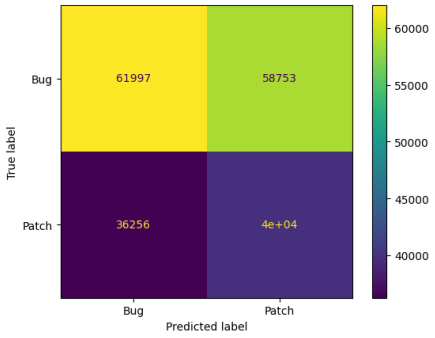


Figure 4.4: 1DBasemodel Word2Vec Confusion matrix

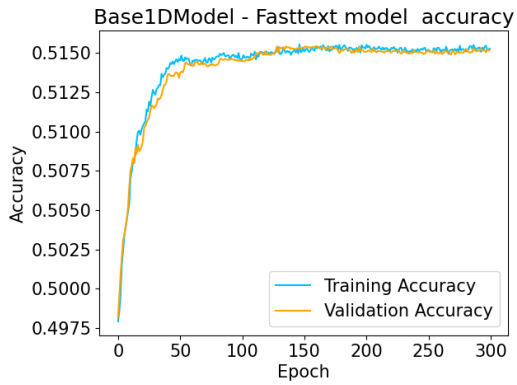


Figure 4.5: 1DBasemodel Fasttext Model Accuracy vs Epoch

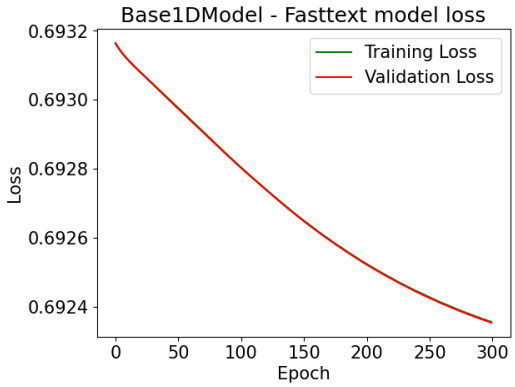


Figure 4.6: 1DBasemodel Fasttext Model Loss vs Epoch

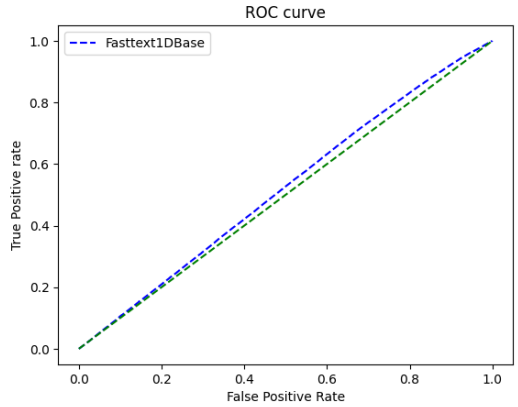


Figure 4.7: 1DBasemodel Fasttext Model ROC, AUC score : 0.520

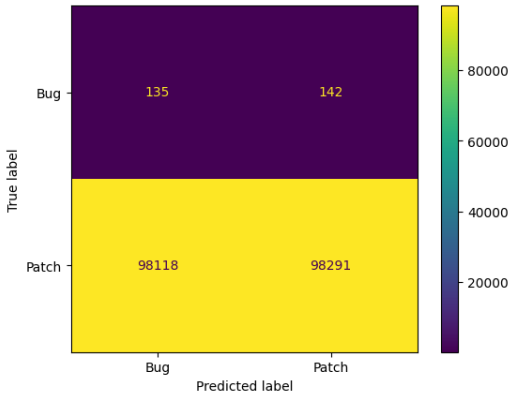


Figure 4.8: 1DBasemodel Fasttext Confusion matrix

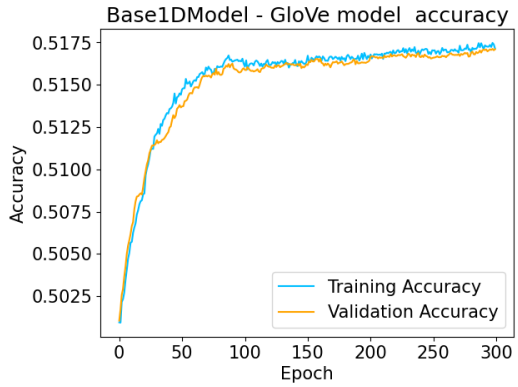


Figure 4.9: 1DBasemodel GloVe Model Accuracy vs Epoch

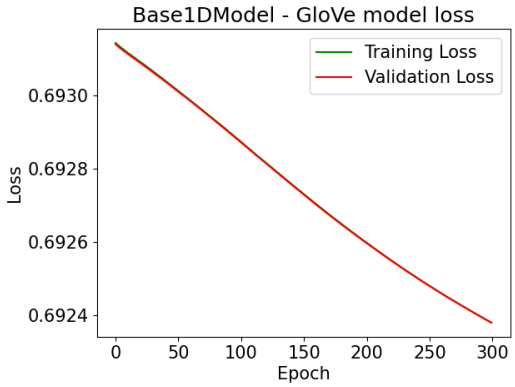


Figure 4.10: 1DBasemodel GloVe Model Loss vs Epoch

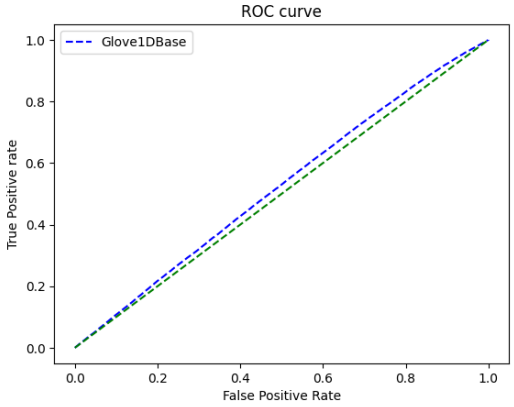


Figure 4.11: 1DBasemodel GloVe Model ROC, AUC score : 0.522

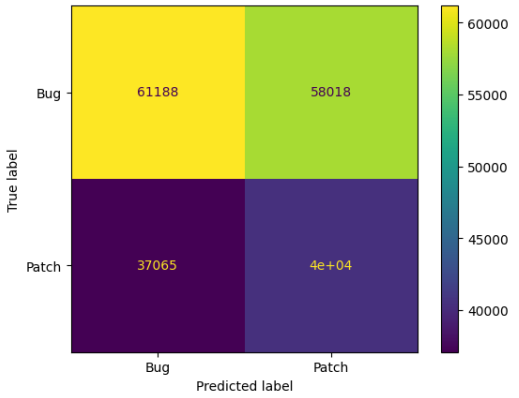


Figure 4.12: 1DBasemodel GloVe Confusion matrix

Results Interpretation

The 1D Basemodel as referred in Section 2.8 is a simple convolutional neural network with the input dimension of 1x1225. The models were able to achieve an average Accuracy of 0.5112, average Precision of 0.6040, average Recall of 0.5148 and average F1 Score of 0.5270. Looking at the results of the Accuracy vs Epoch Graph in Figures 4.1,4.5,4.9 we notice that all three models struggle to improve past the training accuracy of 0.52 percent. The models also achieved an average AUC score of 0.521 with all three performing similar to random choice models as shown in the Figures 4.11, 4.7,4.3.

The architecture does not appear to be well-suited for learning from the dataset. Even when different embeddings were provided, the models consistently performed poorly. The dataset is balanced in that each buggy method corresponds to its respective fixed code. During our training experiments, increasing the number of epochs did not improve performance, the model's training accuracy stagnated between 0.51 and 0.52.

Similar performance is observed with the same architecture trained on the CodeXGLUE dataset. The architecture struggles to learn the dataset with a 300-epoch training interval. Notably, training with early stopping on validation accuracy results in Word2vec and FastText embedding models converging at 25 epochs. In contrast, the GloVe model went up to 250 epochs.

4.1.2 2D Basemodel

This section contains model performance results from training our 2D Basemodel on our primary dataset, as well as the CodeXGLUE dataset. The models are trained until the maximum epoch of 300, with the results of the models trained till convergence based on the validation accuracy, moved to Appendix A for the CodeXGLUE and Appendix B for the primary dataset.

Basemodel	Accuracy	Precision	Recall	F1 Score
Word2Vec	0.5193	0.4098	0.5253	0.4604
Fasttext	0.5201	0.4830	0.5223	0.5019
GloVe	0.5166	0.5346	0.5165	0.5254

Table 4.3: 2D Basemodel CNN Bug detection Models Performance trained with different embeddings

Basemodel	Accuracy	Precision	Recall	F1 Score
Word2Vec	0.5211	0.5568	0.5197	0.5376
Fasttext	0.5219	0.5369	0.5212	0.5289
GloVe	0.5160	0.5253	0.5157	0.5205

Table 4.4: CodeXGLUE 2D Basemodel CNN Bug detection Models Performance trained with different embeddings

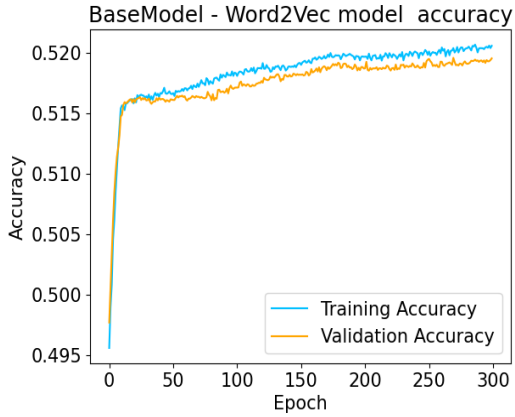


Figure 4.13: 2D Basemodel Word2Vec Model Accuracy vs Epoch

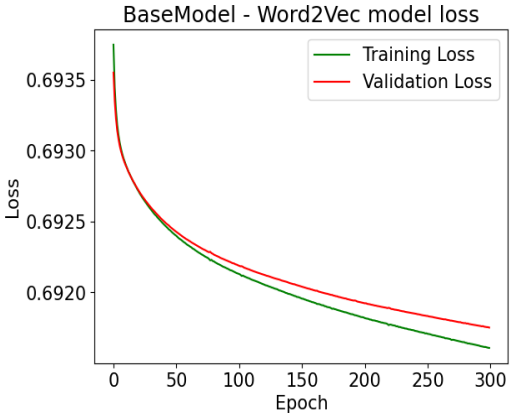


Figure 4.14: 2D Basemodel Word2Vec Model Loss vs Epoch

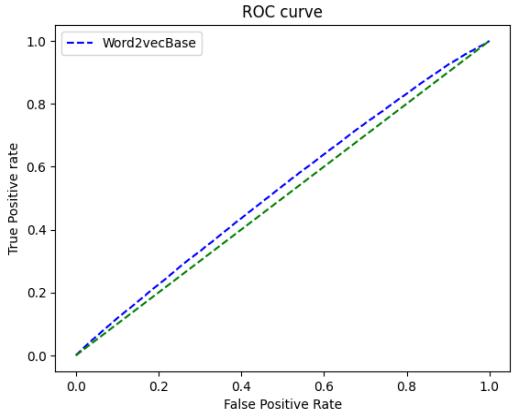


Figure 4.15: 2D Basemodel Word2Vec Model ROC, AUC score :0.528

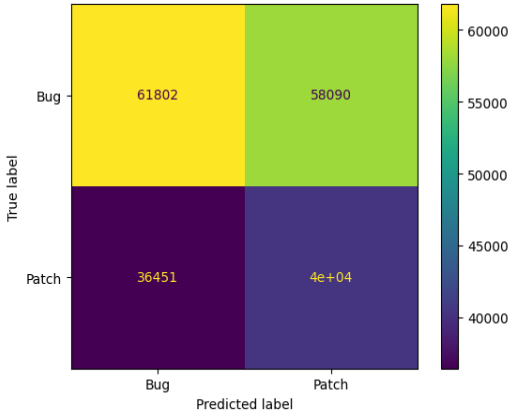


Figure 4.16: 2D Basemodel Word2Vec Confusion matrix

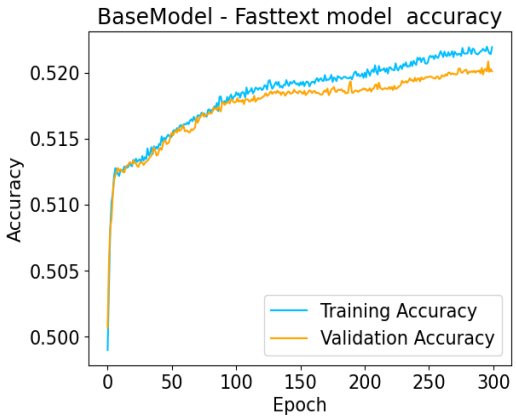


Figure 4.17: 2D Basemodel Fasttext Model Accuracy vs Epoch

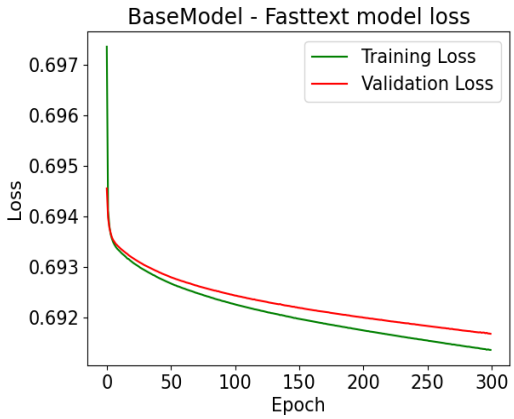


Figure 4.18: 2D Basemodel Fasttext Model Loss vs Epoch

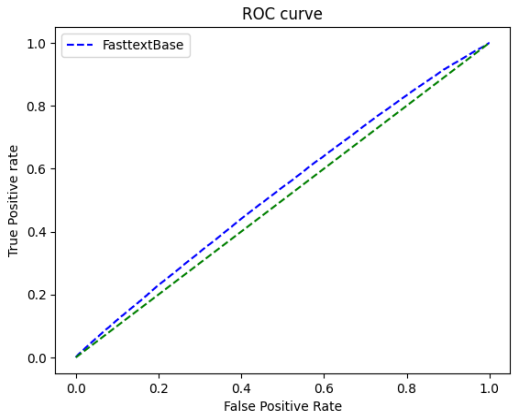


Figure 4.19: 2D Basemodel Fasttext Model ROC, AUC score : 0.530

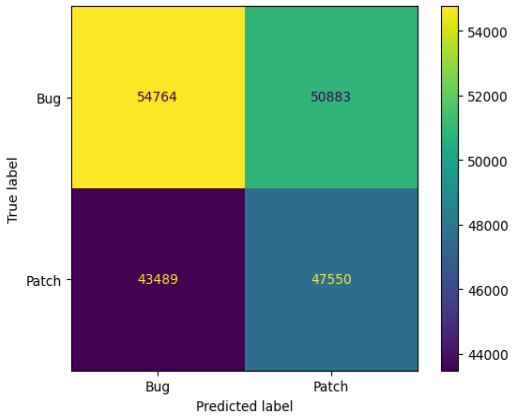


Figure 4.20: 2D Basemodel Fasttext Confusion matrix

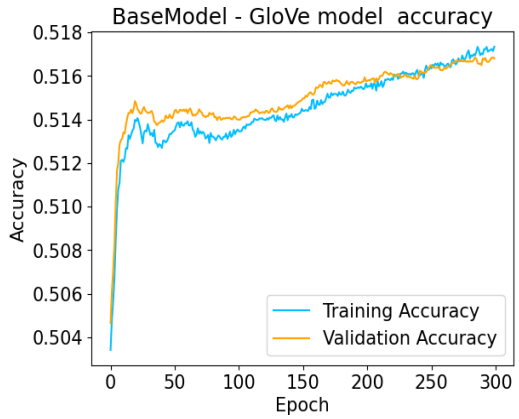


Figure 4.21: 2D Basemodel GloVe Model Accuracy vs Epoch

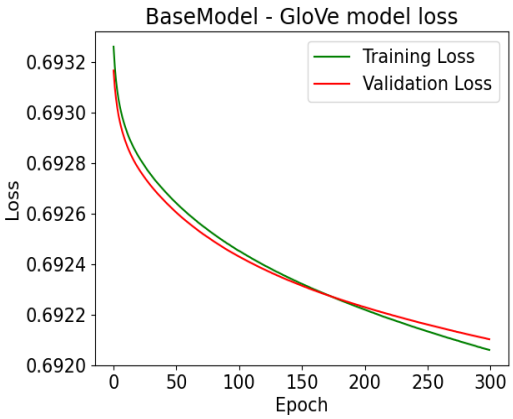


Figure 4.22: 2D Basemodel GloVe Model Loss vs Epoch

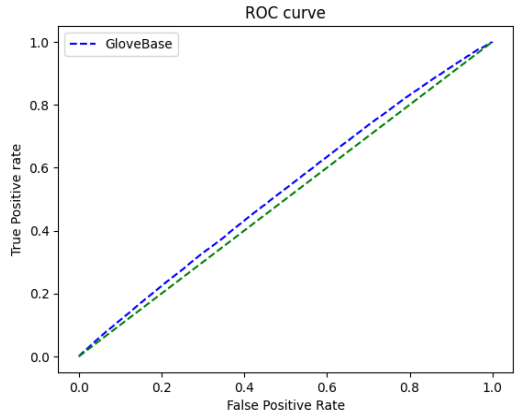


Figure 4.23: 2D Basemodel GloVe Model ROC, AUC score : 0.525

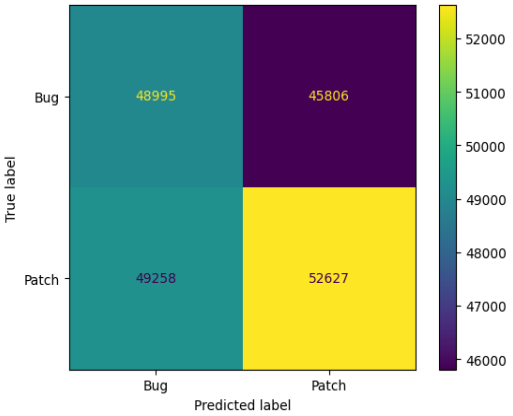


Figure 4.24: 2D Basemodel GloVe Confusion matrix

Results Interpretation

The 2D Basemodel as referred in Section 2.8 is also another simple convolutional neural network architecture with a 35x35 input layer. The models were able to achieve an average Accuracy of 0.5186, average Precision of 0.4758, average Recall of 0.5213 and average F1 Score of 0.4959.Looking at the results of the Accuracy vs Epoch Graph in Figures 4.13,4.17,4.21 we notice the models are struggling to perform on this dataset. The models also achieved an average AUC score of 0.527

with the Glove model performing worse than a random choice models as shown in the Figures 4.23.

The 2D Basemodel shares the same architecture as the 1D Basemodel, with the main difference being the input dimension. However, it faced similar challenges in learning the dataset. Despite increasing the number of epochs, the training accuracy remained stagnant, fluctuating between 0.51 and 0.52 across all the models.

Similar performance is also observed with the 2D Basemodel architecture trained on the CodeXGLUE dataset. The architecture struggles to learn the dataset with a 300-epoch training interval. In particular, training with early stopping on validation accuracy results in Word2Vec embedding models converging at 1000 epochs. In contrast, the FastText and Glove models were cut off at 2000 epochs.

4.1.3 AlexNet

This section contains model performance results from training our AlexNet models on our primary dataset, as well as the CodeXGLUE dataset. The models are trained until the maximum epoch of 300, with the results of the models trained till convergence based on the validation accuracy, moved to Appendix A for the CodeXGLUE and Appendix B for the primary dataset.

AlexNet	Accuracy	Precision	Recall	F1 Score
Word2Vec	0.5681	0.7866	0.5477	0.6457
Fasttext	0.6078	0.5472	0.6231	0.5827
GloVe	0.5936	0.7177	0.5753	0.6386

Table 4.5: AlexNet CNN Bug detection Models Performance trained with different embeddings

AlexNet	Accuracy	Precision	Recall	F1 Score
Word2Vec	0.5228	0.3777	0.5322	0.4418
Fasttext	0.5373	0.5356	0.5374	0.5365
GloVe	0.6509	0.7813	0.6196	0.6911

Table 4.6: CodeXGLUE AlexNet CNN Bug detection Models Performance trained with different embeddings

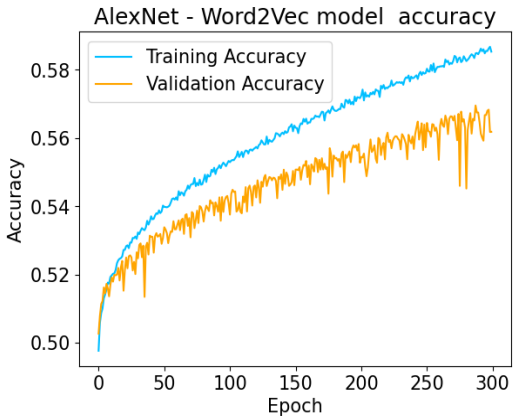


Figure 4.25: AlexNet Word2Vec Model Accuracy vs Epoch

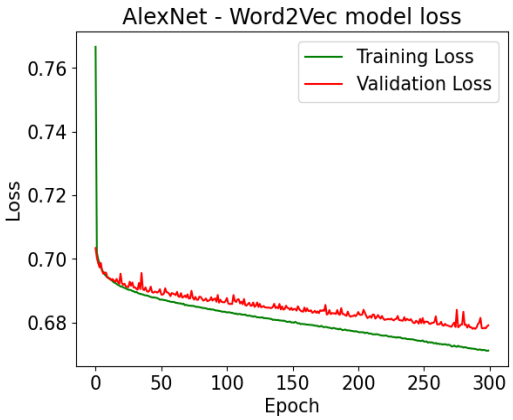


Figure 4.26: AlexNet Word2Vec Model Loss vs Epoch

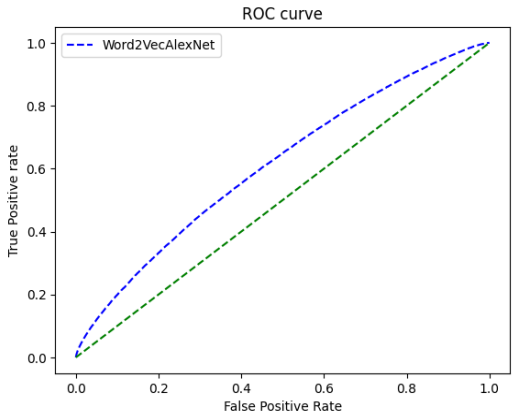


Figure 4.27: AlexNet Word2Vec Model ROC, AUC score :0.611

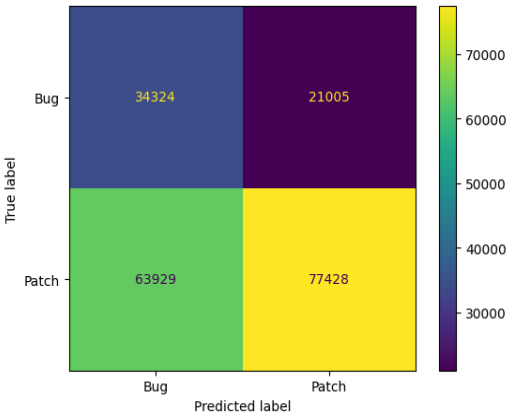


Figure 4.28: AlexNet Word2Vec Confusion matrix

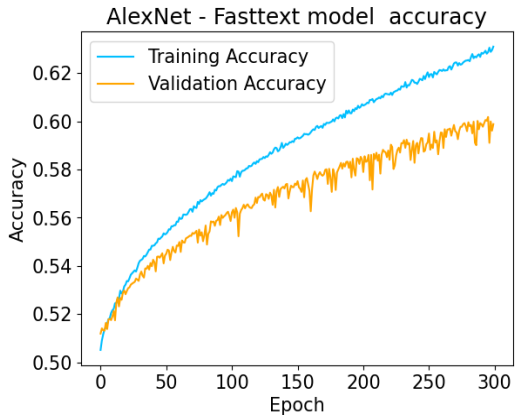


Figure 4.29: AlexNet Fasttext Model Accuracy vs Epoch

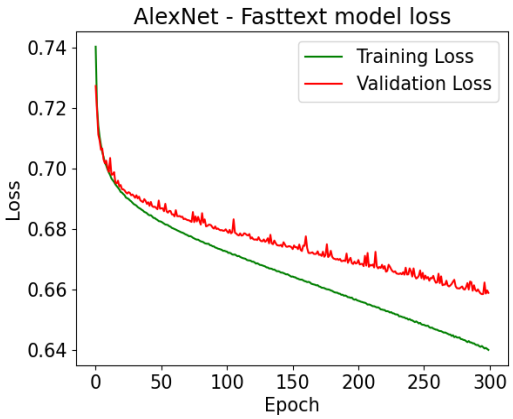


Figure 4.30: AlexNet Fasttext Model Loss vs Epoch

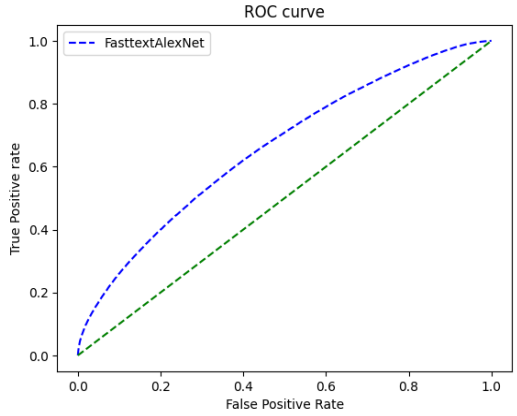


Figure 4.31: AlexNet Fasttext Model ROC, AUC score : 0.659

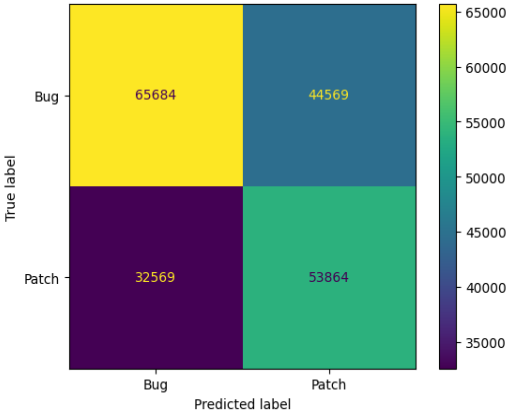


Figure 4.32: AlexNet Fasttext Confusion matrix

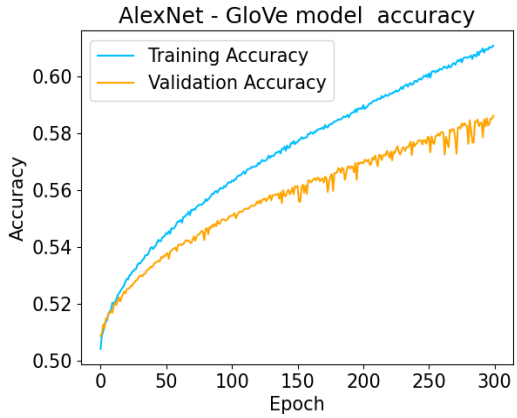


Figure 4.33: AlexNet GloVe Model Accuracy vs Epoch

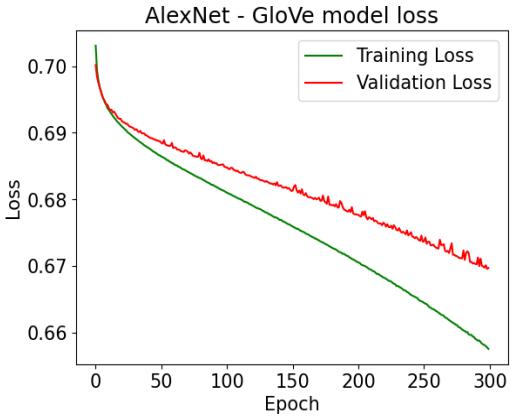


Figure 4.34: AlexNet GloVe Model Loss vs Epoch

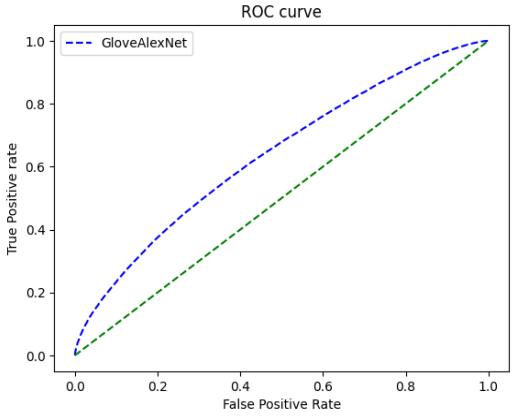


Figure 4.35: AlexNet GloVe Model ROC, AUC score : 0.637

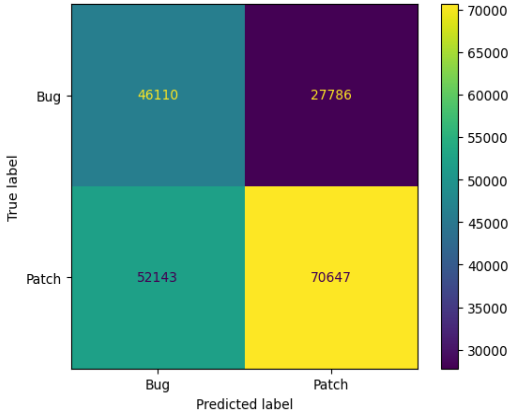


Figure 4.36: AlexNet GloVe Confusion matrix

Results Interpretation

AlexNet architecture models were able to achieve an average Accuracy of 0.5898, average Precision of 0.6838, average Recall of 0.5820 and average F1 Score of 0.6223. Looking at the results of the Accuracy vs Epoch Graph in Figures 4.25,4.29,4.33 the models started improving around the neighborhood of 20^{th} epoch. The models also achieved an average AUC score of 0.635 with the Fasttext model having the highest score of 0.659 as shown in Figure 4.31.

Unlike the base models, the AlexNet model demonstrates significant potential in learning from the dataset. As the number of epochs increases, the model becomes more proficient at understanding the data. All the models appear to be learning the dataset at a similar rate, given the different embeddings, which makes this architecture suitable for the task.

The AlexNet model demonstrates the potential to learn even on the CodeXGLUE dataset. With the dataset, the GloVe embedding produced a better model compared with the other embeddings, converging slightly above 350 epochs with early stop validation. Demonstrating that the architecture is suitable for the task when paired with a suitable embedding is provided.

4.1.4 VGG16

This section contains model performance results from training our VGG16 models on our primary dataset, as well as the CodeXGLUE dataset. The models are trained until the maximum epoch of 300, with the results of the models trained until convergence based on the validation accuracy, moved to Appendix A for the CodeXGLUE and Appendix B for the primary dataset.

VGG16	Accuracy	Precision	Recall	F1 Score
Word2Vec	0.5155	0.5111	0.5161	0.5136
Fasttext	0.5097	0.7270	0.5071	0.5975
GloVe	0.5137	0.6114	0.5119	0.5572

Table 4.7: VGG16 CNN Bug detection Models Performance trained with different embeddings

VGG16	Accuracy	Precision	Recall	F1 Score
Word2Vec	0.5207	0.4857	0.5223	0.5033
Fasttext	0.5212	0.4691	0.5237	0.4949
GloVe	0.5200	0.5184	0.5200	0.5192

Table 4.8: CodeXGLUE CNN Bug detection Models Performance trained with different embeddings

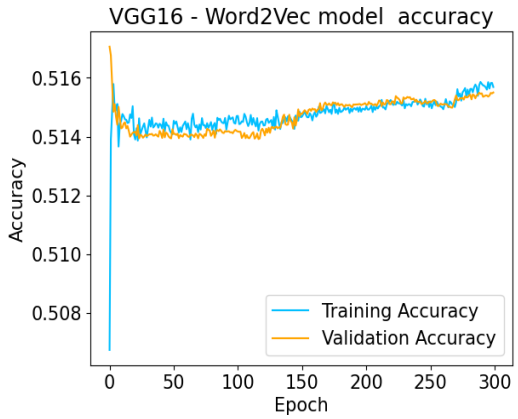


Figure 4.37: VGG16 Word2Vec Model Accuracy vs Epoch

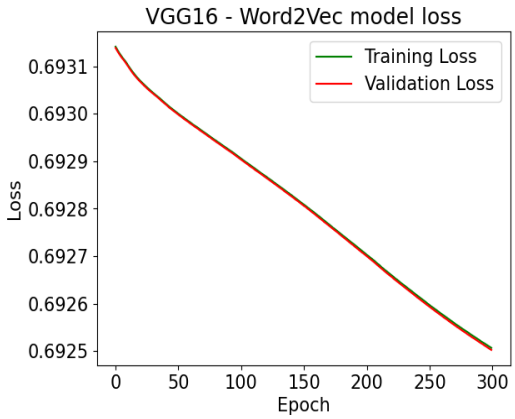


Figure 4.38: VGG16 Word2Vec Model Loss vs Epoch

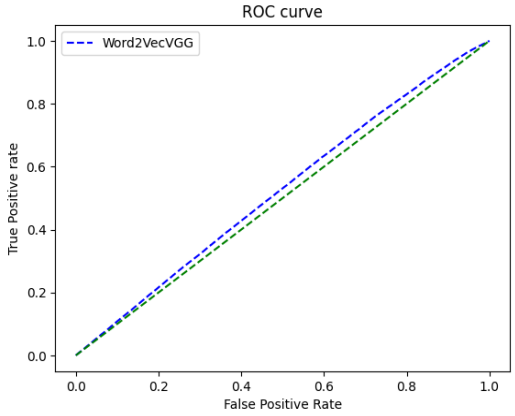


Figure 4.39: VGG16 Word2Vec Model ROC, AUC score :0.523

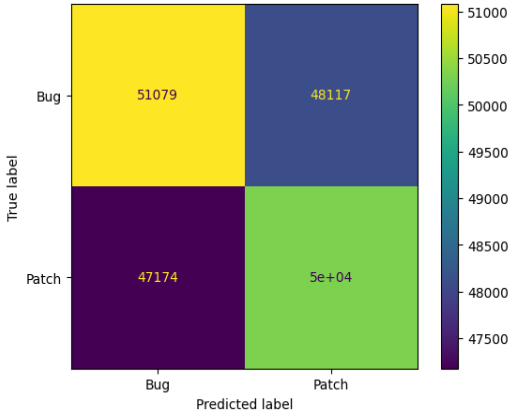


Figure 4.40: VGG16 Word2Vec Confusion matrix

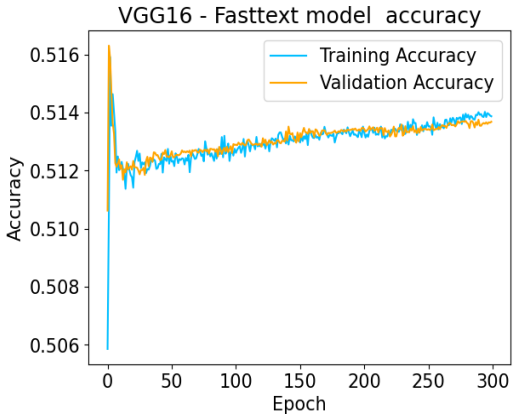


Figure 4.41: VGG16 Fasttext Model Accuracy vs Epoch

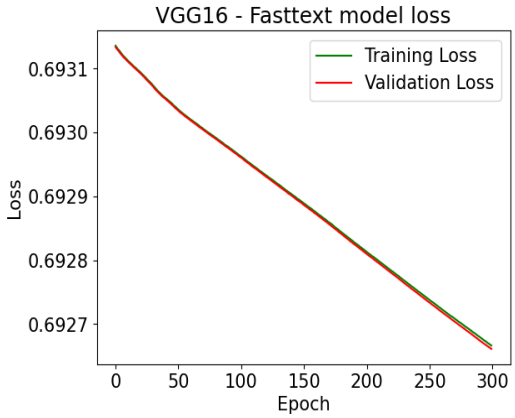


Figure 4.42: VGG16 Fasttext Model Loss vs Epoch

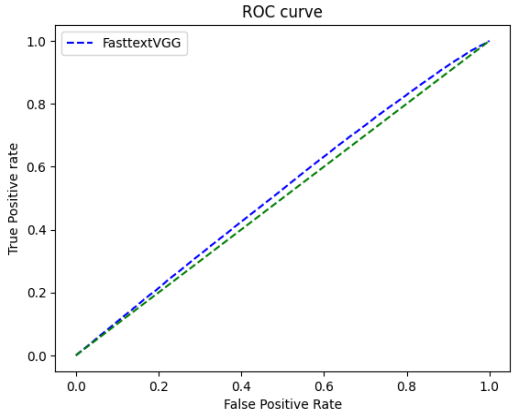


Figure 4.43: VGG16 Fasttext Model ROC, AUC score : 0.521

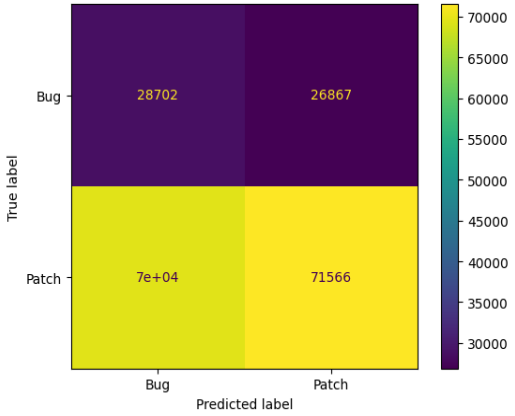


Figure 4.44: VGG16 Fasttext Confusion matrix

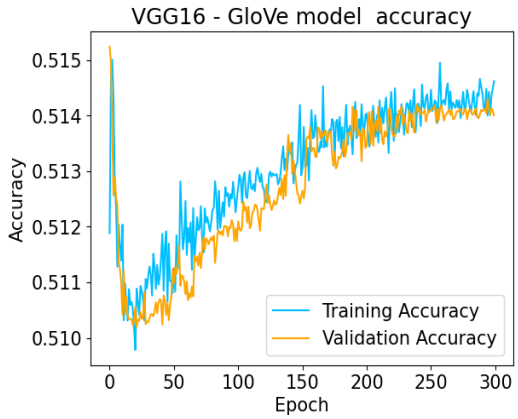


Figure 4.45: VGG16 Glove Model Accuracy vs Epoch

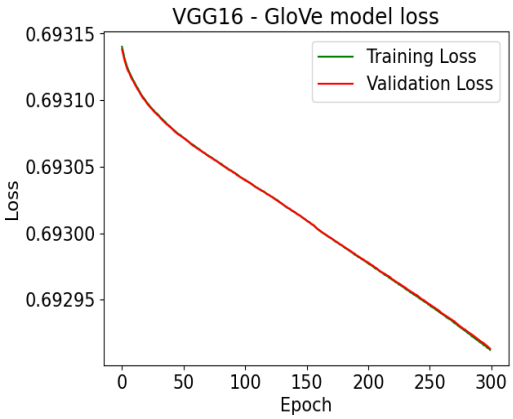


Figure 4.46: VGG16 GloVe Model Loss vs Epoch

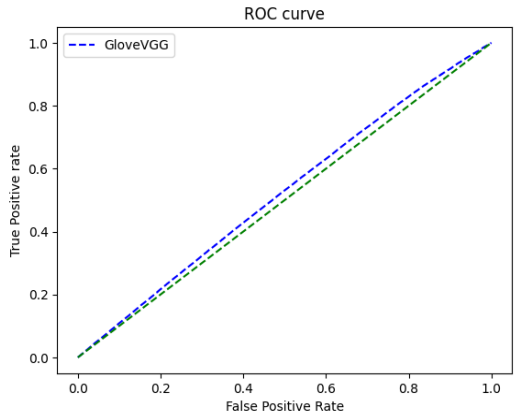


Figure 4.47: VGG16 GloVe Model ROC, AUC score : 0.521

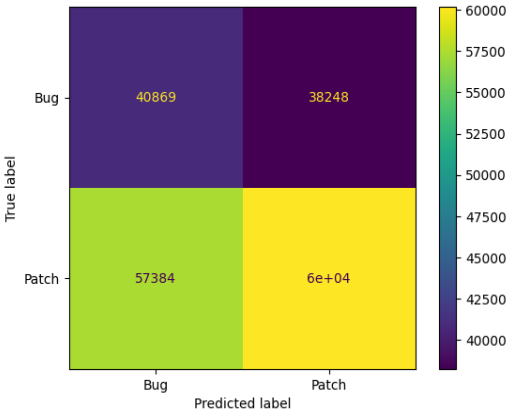


Figure 4.48: VGG16 Glove Confusion matrix

Results Interpretation

The VGG16 performed quite poor similar to the models in subsection 4.1.1 and 4.1.2 with an average Accuracy of 0.5129, average Precision of 0.6165, average Recall of 0.5117 and average F1 Score of 0.5561. Looking at the results of the Accuracy vs Epoch Graph in Figures 4.37,4.41,4.45, the models are struggling to learn features from the dataset. The models also achieved an average AUC score of 0.521 which is close to a random classifier with the score of 0.5.

The VGG16 models demonstrated similar performance on both the primary dataset

and the CodeXGLUE dataset, except for the GloVe embedding model, which showed potential to learn after the base epoch of 300. We observed the model's performance growth from the 400th epoch, which converged at around 1200 epochs using early stop validation.

4.1.5 MobileNet

This section contains model performance results from training our MobileNet models on our primary dataset, as well as the CodeXGLUE dataset. The models are trained until the maximum epoch of 300, with the results of the models trained until convergence based on the validation accuracy, moved to Appendix A for the CodeXGLUE and Appendix B for the primary dataset.

MobileNet	Accuracy	Precision	Recall	F1 Score
Word2Vec	0.8830	0.8865	0.0.8804	0.8835
Fasttext	0.8386	0.8566	0.8271	0.8416
GloVe	0.8163	0.8382	0.8032	0.8203

Table 4.9: MobileNet CNN Bug detection Models Performance trained with different embeddings

MobileNet	Accuracy	Precision	Recall	F1 Score
Word2Vec	0.8087	0.8084	0.8089	0.8087
Fasttext	0.8201	0.8223	0.8187	0.8205
GloVe	0.7229	0.7333	0.7184	0.7258

Table 4.10: CodeXGLUE MobileNet CNN Bug detection Models Performance trained with different embeddings

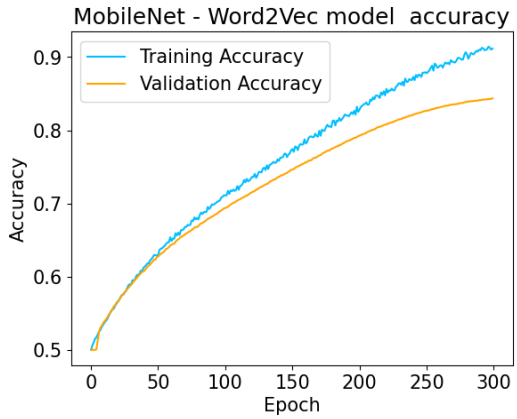


Figure 4.49: MobileNet Word2Vec Model Accuracy vs Epoch

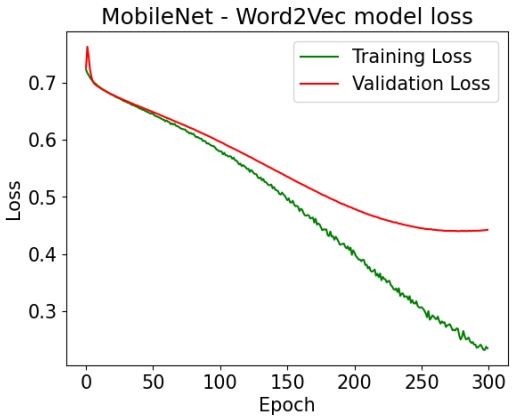


Figure 4.50: MobileNet Word2Vec Model Loss vs Epoch

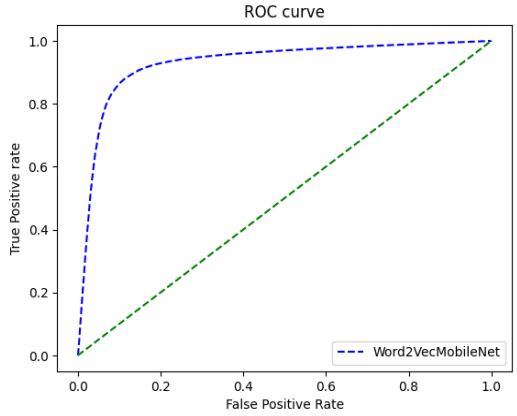


Figure 4.51: MobileNet Word2Vec Model ROC, AUC score : 0.929

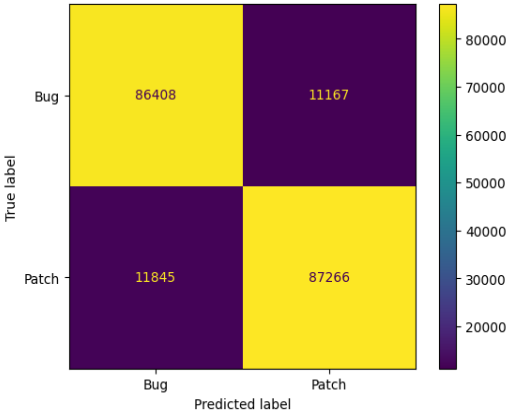


Figure 4.52: MobileNet Word2Vec Confusion matrix

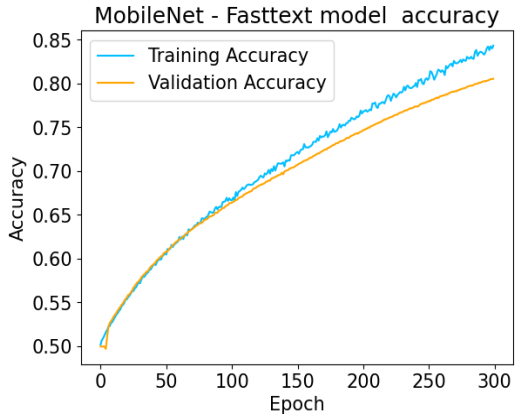


Figure 4.53: MobileNet Fasttext Model Accuracy vs Epoch

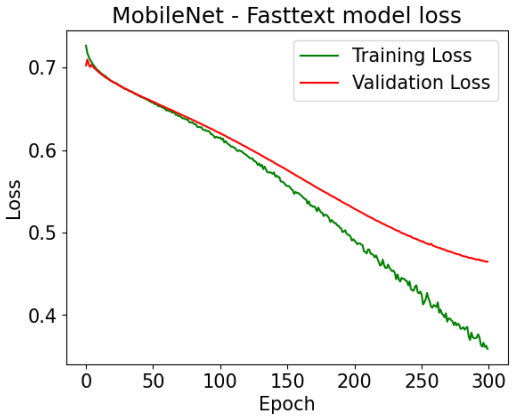


Figure 4.54: MobileNet Fasttext Model Loss vs Epoch

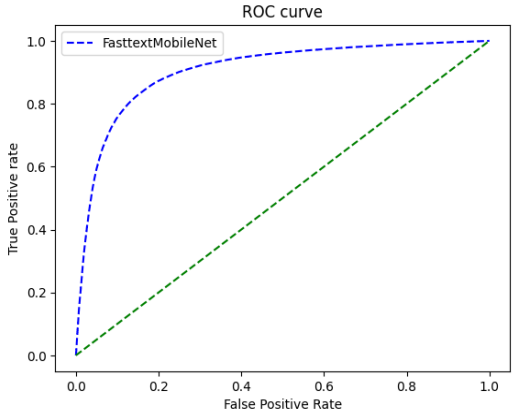


Figure 4.55: MobileNet Fasttext Model ROC, AUC score :0.905

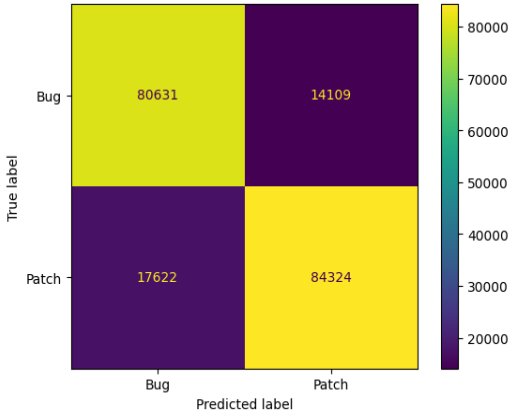


Figure 4.56: MobileNet Fasttext Confusion matrix

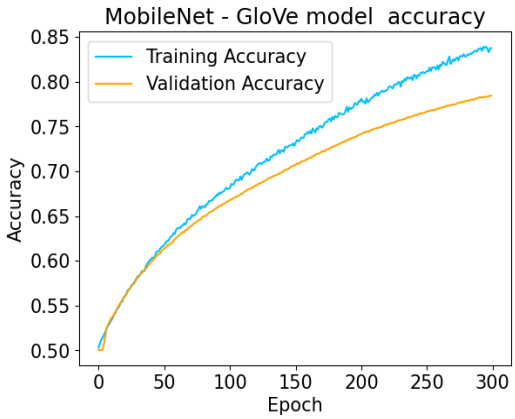


Figure 4.57: MobileNet GloVe Model Accuracy vs Epoch

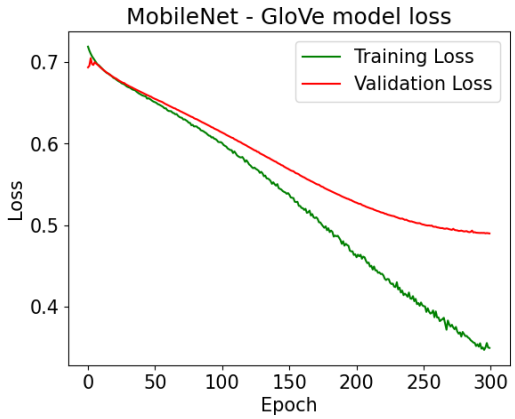


Figure 4.58: MobileNet GloVe Model Loss vs Epoch

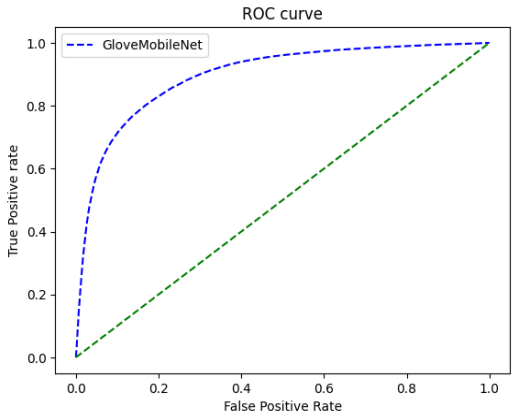


Figure 4.59: MobileNet GloVe Model ROC, AUC score : 0.895

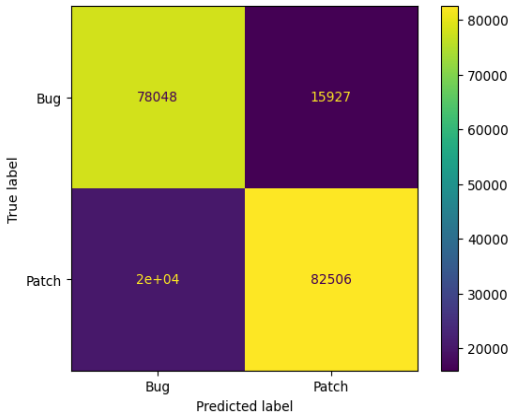


Figure 4.60: MobileNet GloVe Confusion matrix

Results Interpretation

MobileNet architecture models achieved an average Accuracy of 0.8459, average Precision of 0.8604, average Recall of 0.8369 and average F1 Score of 0.8484.Looking at the results of the Accuracy vs Epoch Graph in Figures 4.49,4.53,4.57 the models started improving around the neighborhood of 10^{th} epoch. The models also achieved an average AUC score of 0.909 with the Word2vec model having the highest score of 0.929 as shown in figure 4.51.

So far, the MobileNet architecture has demonstrated strong performance and learning capability on the dataset. It converges more quickly within the given epochs, making it the best-suited model for the dataset compared to the other architectures we tested in our experiments.

Similar performance is observed for the MobileNet architecture on both the primary dataset and the CodeXGLUE dataset, with the model showing strong performance across all embedding methods. The model continues to grow further after 300 epochs, until early stop validation is reached on both datasets.

4.1.6 ResNet18

This section contains model performance results from training our ResNet18 models on our primary dataset, as well as the CodeXGLUE dataset. The models are trained until the maximum epoch of 300, with the results of the models trained until convergence based on the validation accuracy, moved to Appendix A for the CodeXGLUE and Appendix B for the primary dataset.

ResNet18	Accuracy	Precision	Recall	F1 Score
Word2Vec	0.7415	0.6566	0.7913	0.7177
Fasttext	0.7831	0.8656	0.7432	0.7997
GloVe	0.7922	0.8531	0.7607	0.8043

Table 4.11: ResNet18 Word2Vec CNN Bug detection Models Performance

ResNet18	Accuracy	Precision	Recall	F1 Score
Word2Vec	0.5826	0.5504	0.5883	0.5687
Fasttext	0.6231	0.5835	0.6337	0.6075
GloVe	0.8516	0.9014	0.8198	0.8586

Table 4.12: CodeXGLUE ResNet18 Word2Vec CNN Bug detection Models Performance

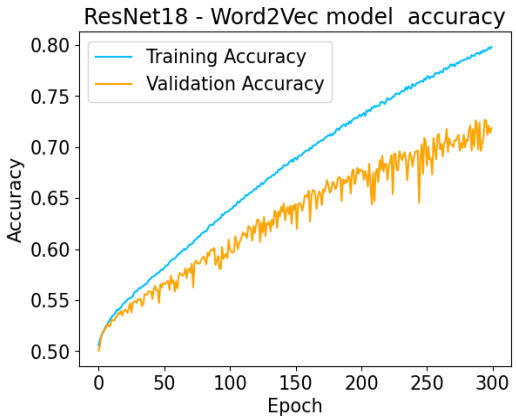


Figure 4.61: ResNet18 Word2Vec Model Accuracy vs Epoch

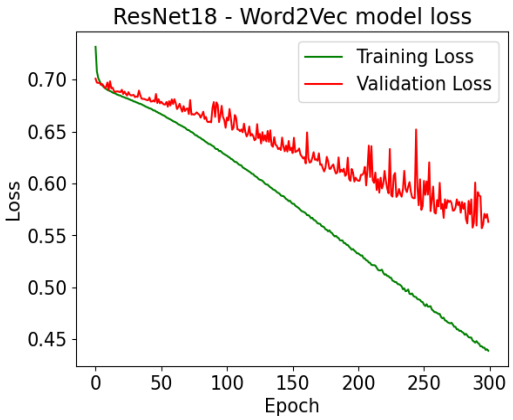


Figure 4.62: ResNet18 Word2Vec Model Loss vs Epoch

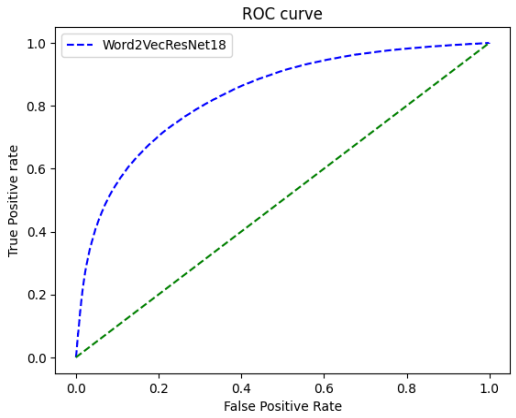


Figure 4.63: ResNet18 Word2Vec Model ROC, AUC score : 0.832

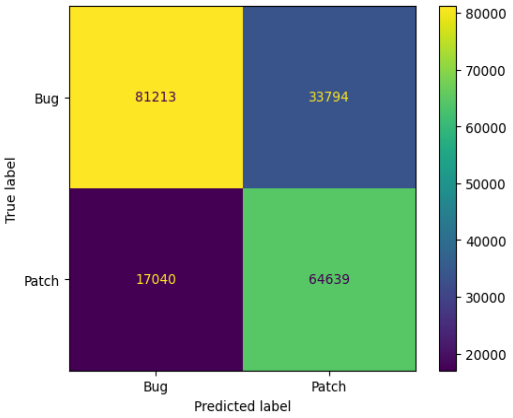


Figure 4.64: ResNet18 Word2Vec Confusion matrix

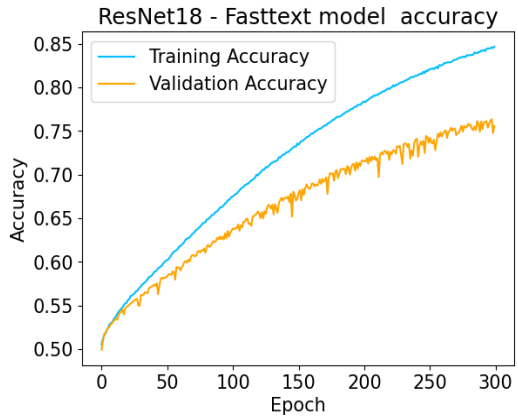


Figure 4.65: ResNet18 Fasttext Model Accuracy vs Epoch

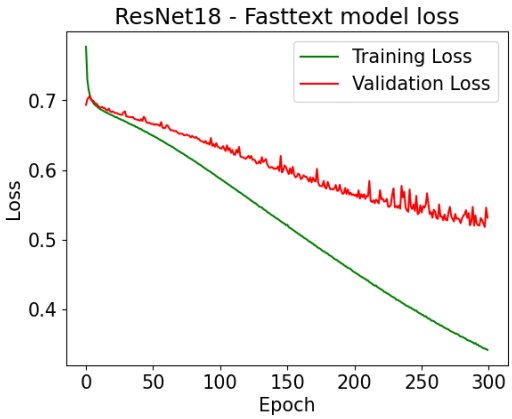


Figure 4.66: ResNet18 Fasttext Model Loss vs Epoch

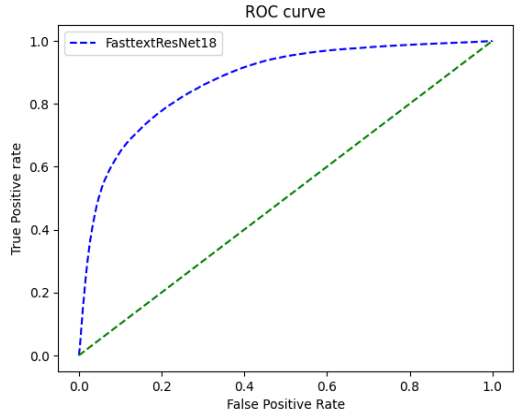


Figure 4.67: ResNet18 Fasttext Model ROC, AUC score : 0.871

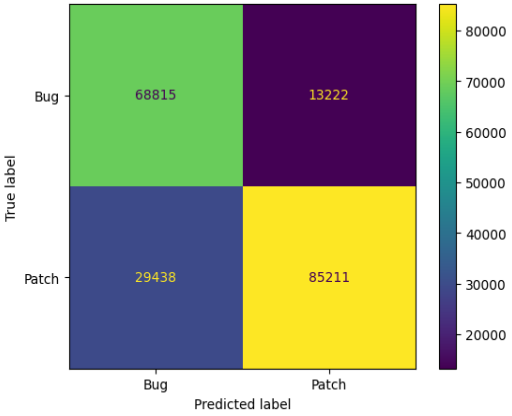


Figure 4.68: ResNet18 Fasttext Confusion matrix

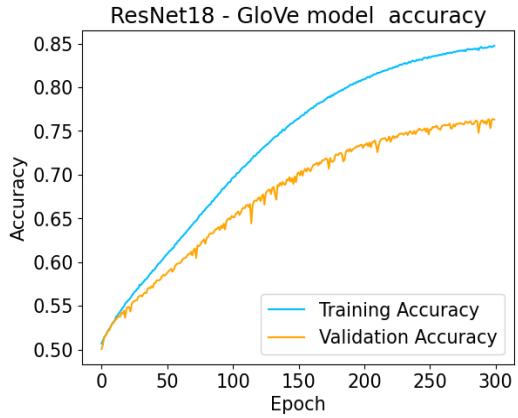


Figure 4.69: ResNet18 GloVe Model Accuracy vs Epoch

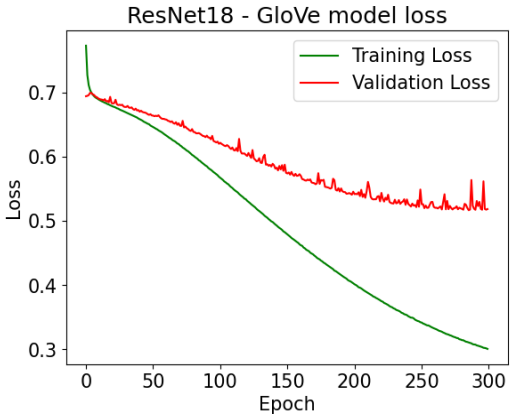


Figure 4.70: ResNet18 GloVe Model Loss vs Epoch

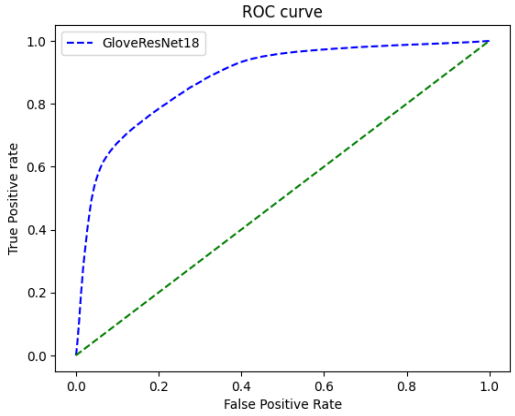


Figure 4.71: ResNet18 GloVe Model ROC, AUC score : 0.880

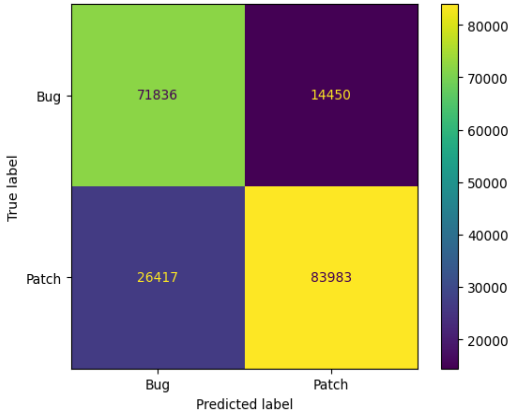


Figure 4.72: ResNet18 GloVe Confusion matrix

Results Interpretation

ResNet18 architecture models achieved an average Accuracy of 0.7722, average Precision of 0.7917, average Recall of 0.7650 and average F1 Score of 0.7739.Looking at the results of the Accuracy vs Epoch Graph in Figures 4.61,4.65,4.69 the models started improving around the neighborhood of 10^{th} epoch. The models also achieved an average AUC score of 0.861 with the Glove model having the highest score of 0.880 as shown in figure 4.59.

The ResNet18 architecture demonstrates strong performance and learning capability across the three embeddings, coming in second to MobileNet. It converges more quickly within the given number of epochs compared to the other architectures, with the potential for further improvement with additional epochs, making ResNet18 a suitable model for the dataset.

The ResNet18 architecture also shows potential for CodeXGLUE, with the best performing model being the GloVe embedding model, and showing faster convergence with early stop validation, and with the other two embeddings falling slightly behind.

4.2 Model Embedding Analysis

To analyze whether our 2D architecture representations captured meaningful relationships within our embedding representations for our three embedding choices, Word2Vec, Fasttext, and GloVe applied on our primary dataset, we extracted features from our selected top three performing models, AlexNet, Mobilenet, and ResNet18, using keras, sklearn TSNE, and matplotlib Python libraries to compute and plot t-SNE (t-distributed Stochastic Neighbor Embedding) visualizations and Silhouette scores.

Word2Vec t-SNE

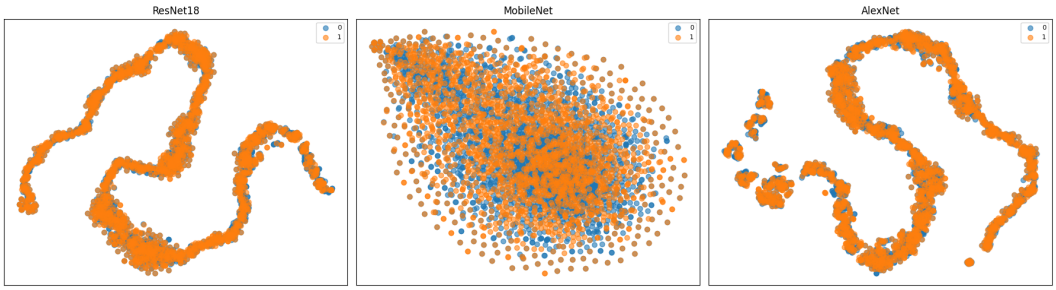


Figure 4.73: t-SNE visualizations of Word2vec embeddings applied on the models ResNet18 , MobileNet and AlexNet in the order from left to right

Figure 4.73 visualizes the t-SNE of our Word2vec embeddings applied to the ResNet18, MobileNet, and AlexNet models. Looking at the visualization of the ResNet18 model, we notice a clear, continuous manifold; however, we also notice a significant overlap between the two classes, indicating that while the ResNet18 model structurally organizes the data, the feature representation does not strongly separate the classes. The results align with the silhouette score of 0.00046, indicating weak clustering quality.

Results from the MobileNet display a dense, compact, cloudy, and heavily overlapping manifold with small class-discriminative boundaries. The model produced a silhouette score of 0.00112, slightly higher than that of ResNet18, but still essentially no clustering structure with respect to the labels.

Results from AlexNet create a similar structured manifold to ResNet18, also with overlapping classes, but slightly more structured than MobileNet. We can also notice a few local groupings, but no distinct separations between the classes. The silhouette score 0.00085 also shows weak class separation, but marginally better organization than ResNet18.

Fasttext t-SNE

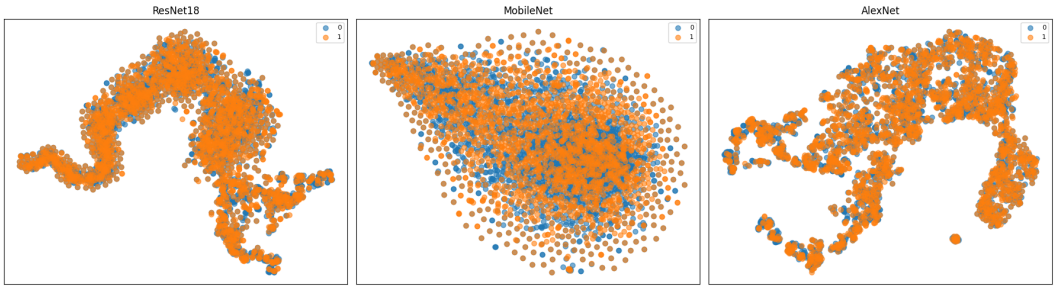


Figure 4.74: t-SNE visualizations of Fasttext embeddings applied on the models ResNet18 , MobileNet and AlexNet in the order from left to right

Figure 4.74 visualizes the t-SNE of our Fasttext embeddings applied to the ResNet18, MobileNet, and AlexNet models. The results of the ResNet model show a winding manifold similar to those from the Word2Vec embedding, with overlapping between the two classes and nearly nonexistent class separation, as reflected in a low silhouette score of 0.00042.

Results from MobileNet display a dense, compact, cloudy, and heavily overlapping manifold with small class-discriminative boundaries, similar to those in the Word2Vec embedding. The model produced a silhouette score of 0.00078. While the results from AlexNet produce a manifold structure that is more entangled than that of Word2Vec, it still maintains the overlapping classes with a silhouette of 0.00026.

GloVe t-SNE

Figure 4.75 visualizes the t-SNE of our GloVe embeddings applied to the ResNet18, MobileNet, and AlexNet models. The results of the ResNet model show a cloud-like,

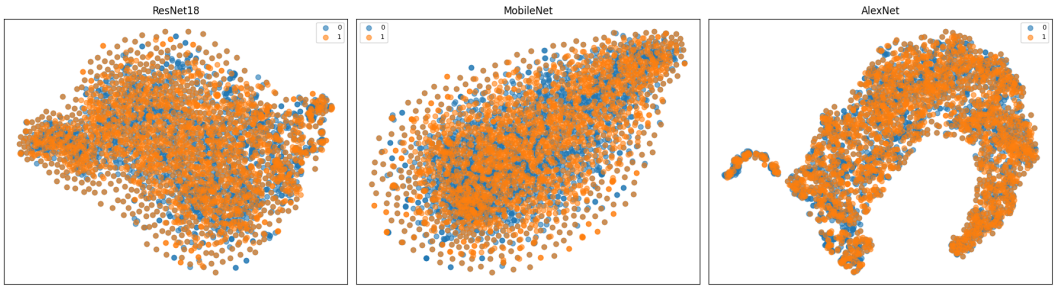


Figure 4.75: t-SNE visualizations of GloVe embeddings applied on the models ResNet18 , MobileNet and AlexNet in the order from left to right

densely overlapping manifold with no clear grouping structure, with a silhouette value of 0.00046. Results from MobileNet display a dense, compact, and cloudy representation, similar to both Word2vec and FastText, with a near-zero silhouette score of 0.00088. While the results from AlexNet produce an arc-like manifold structure similar to FastText embedding, with a silhouette of 0.00048.

Results Interpretation

Across the selected embeddings, Word2Vec, GloVe, and FastText, MobileNet consistently produced the relatively best model, however, the results still fail to show a meaningful separation of classes. Word2Vec embeddings, together with MobileNet, perform marginally better than the other combinations, but are still inadequate for discriminative clustering. Since t-SNE is an unsupervised method that reduces high-dimensional features to a lower dimension for visualization while trying to preserve local structure, it seems not to fully capture the whole high-dimensional feature space, which the models are able to capture based on the performances of the models as illustrated in Section 4.1.

4.3 Visualization Results

To further analyze our visualization results, we will look at two comparison metrics Intersection over union with relative thresholds and Fraction of Elements Above Threshold [77] applied on our primary dataset. Along with the two metrics, we

will look at mapping the weights from our visualization models to the line of codes predicted as containing defects and calculate the percentages. The scope of this experiment will be limited to our selected performing models, which are AlexNet, MobileNet, and ResNet18 on our primary dataset.

4.3.1 Intersection Over Union with Relative Thresholds

This metric compares two heatmaps generated using explainable ai methods by different models that have been trained with distinct embeddings but share the same architecture. The results are binarized using a relative threshold, allowing us to calculate the intersection over union to determine the overlap of words deemed important by both models. This metric enables us to assess if two convolutional neural networks with the same architecture, yet different embeddings, are extracting similar features.

	$f(0.3)$	$f(0.7)$
	$\mu \pm \sigma$	$\mu \pm \sigma$
Word2Vec and Fasttext	0.03195 ± 0.02865	0.00029 ± 0.00326
GloVe and Fasttext	0.03305 ± 0.02719	0.00038 ± 0.00411
Word2Vec and GloVe	0.03340 ± 0.02836	0.00032 ± 0.00340

Table 4.13: AlexNet CNN Saliency Map Intersection of Elements Above Threshold

	$f(0.3)$	$f(0.7)$
	$\mu \pm \sigma$	$\mu \pm \sigma$
Word2Vec and Fasttext	0.04177 ± 0.08916	0.02309 ± 0.04084
GloVe and Fasttext	0.02309 ± 0.04084	0.00112 ± 0.00642
Word2Vec and GloVe	0.02768 ± 0.05169	0.00196 ± 0.00907

Table 4.14: AlexNet CNN GradCam Map Intersection of Elements Above Threshold

	$f(0.3)$	$f(0.7)$
	$\mu \pm \sigma$	$\mu \pm \sigma$
Word2Vec and Fasttext	0.63024 ± 0.29914	0.12129 ± 0.10253
GloVe and Fasttext	0.30111 ± 0.20603	0.02063 ± 0.04140
Word2Vec and GloVe	0.45078 ± 0.18039	0.06887 ± 0.06332

Table 4.15: AlexNet CNN Scorecam Map Intersection of Elements Above Threshold

	$f(0.3)$	$f(0.7)$
	$\mu \pm \sigma$	$\mu \pm \sigma$
Word2Vec and Fasttext	0.59228 ± 0.24876	0.07110 ± 0.06841
GloVe and Fasttext	0.18287 ± 0.12341	0.00910 ± 0.02685
Word2Vec and GloVe	0.27303 ± 0.12171	0.02792 ± 0.03906

Table 4.16: AlexNet CNN FasterScorecam Map Intersection of Elements Above Threshold

Results Interpretation

The heatmap results generated by different explainability models for the three AlexNet variants indicate a strong relationship between the Word2Vec and FastText models, as well as between Word2Vec and GloVe models, based on the findings in Tables 4.14, 4.15, and 4.16. Additionally, the results from the Saliency map demonstrate a close correlation across all the embedding models.

	$f(0.3)$	$f(0.7)$
	$\mu \pm \sigma$	$\mu \pm \sigma$
Word2Vec and Fasttext	0.00006 ± 0.00142	0.00000 ± 0.00001
GloVe and Fasttext	0.00006 ± 0.00140	0.00000 ± 0.00001
Word2Vec and GloVe	0.05525 ± 0.04851	0.00142 ± 0.00816

Table 4.17: Mobilenet CNN Saliency Map Intersection of Elements Above Threshold

	$f(0.3)$	$f(0.7)$
	$\mu \pm \sigma$	$\mu \pm \sigma$
Word2Vec and Fasttext	0.04399 ± 0.07456	0.00291 ± 0.01188
GloVe and Fasttext	0.01967 ± 0.03433	0.00120 ± 0.00746
Word2Vec and GloVe	0.03000 ± 0.04717	0.00179 ± 0.00911

Table 4.18: Mobilenet CNN GradCam Map Intersection of Elements Above Threshold

	$f(0.3)$	$f(0.7)$
	$\mu \pm \sigma$	$\mu \pm \sigma$
Word2Vec and Fasttext	0.58851 ± 0.33180	0.10455 ± 0.09632
GloVe and Fasttext	0.28865 ± 0.22296	0.02195 ± 0.04555
Word2Vec and GloVe	0.46587 ± 0.19392	0.06553 ± 0.07454

Table 4.19: Mobilenet CNN Scorecam Map Intersection of Elements Above Threshold

	$f(0.3)$	$f(0.7)$
	$\mu \pm \sigma$	$\mu \pm \sigma$
Word2Vec and Fasttext	0.61135 ±0.24220	0.07060 ± 0.06644
GloVe and Fasttext	0.20729 ± 0.14017	0.00841 ± 0.02242
Word2Vec and GloVe	0.30452 ± 0.14314	0.02715 ± 0.03615

Table 4.20: Mobilenet CNN FasterScorecam Map Intersection of Elements Above Threshold

Results Interpretation

The heatmaps generated by various explainability models on the three MobileNet architectures reveal a strong correlation between the Word2Vec and FastText models, as well as between Word2Vec and GloVe. This observation is supported by the results presented in Tables 4.18, 4.19, and 4.20. In contrast, the results from the saliency map indicate a significant relationship between the GloVe and FastText models when compared to the other embedding models.

	$f(0.3)$	$f(0.7)$
	$\mu \pm \sigma$	$\mu \pm \sigma$
Word2Vec and Fasttext	0.00003 ± 0.00052	0.00000 ± 0.00002
GloVe and Fasttext	0.00007 ± 0.00084	0.00000 ± 0.00003
Word2Vec and GloVe	0.00004 ± 0.00059	0.00000 ± 0.00002

Table 4.21: Resnet18 CNN Saliency Map Intersection of Elements Above Threshold

	$f(0.3)$	$f(0.7)$
	$\mu \pm \sigma$	$\mu \pm \sigma$
Word2Vec and Fasttext	0.05961 ± 0.09032	0.00412 ± 0.01240
GloVe and Fasttext	0.02186 ± 0.03040	0.001052 ±0.00533
Word2Vec and GloVe	0.03559 ± 0.044642	0.00252 ± 0.00919

Table 4.22: Resnet18 CNN GradCam Map Intersection of Elements Above Threshold

	$f(0.3)$	$f(0.7)$
	$\mu \pm \sigma$	$\mu \pm \sigma$
Word2Vec and Fasttext	0.00673 ± 0.02879	0.00034 ± 0.00200
GloVe and Fasttext	0.05444 ± 0.05813	0.00196 ± 0.00692
Word2Vec and GloVe	0.00167 ± 0.00769	0.00007 ± 0.00077

Table 4.23: Resnet18 CNN Scorecam Map Intersection of Elements Above Threshold

	$f(0.3)$	$f(0.7)$
	$\mu \pm \sigma$	$\mu \pm \sigma$
Word2Vec and Fasttext	0.02083 ± 0.04496	0.00208 ± 0.00787
GloVe and Fasttext	0.03249 ± 0.03607	0.00121 ± 0.00442
Word2Vec and GloVe	0.00609 ± 0.01648	0.00026 ± 0.00237

Table 4.24: Resnet18 CNN FasterScorecam Map Intersection of Elements Above Threshold

Results Interpretation

The heatmaps generated by the various explainability models for the three ResNet18 models indicate a strong correlation between the Glove and Fasttext on average, with a stronger correlation between Word2Vec and Fasttext on the GradCam results.

This is evident from the results presented in Tables 4.22, 4.23, and 4.24. In contrast, the results from the Saliency map demonstrate a very weak correlation among all the embedding models, particularly when examining the thresholds.

4.3.2 Fraction of Elements Above Threshold

This metric measures the proportion of words in the inputs that were considered for each prediction and were above a certain threshold. It helps us understand which portion of each input was taken into account by the model for its prediction. Taking the mean fraction of our dataset token lengths over the maximum input vector of 1225 we get a base of 0.14782 and calculating the ratio of tokens with bug and fix we get a base truth average of 0.00738 which we can use to compare our results. The formula to calculate the relative fraction is given by $f(t)$.

$$f(t) = \frac{g(t).m}{l}$$

where : g(t) is the number of elements in the vector that are larger than the relative threshold t, m is the maximum value in the vector and l is the length of the vector

	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.00977 ± 0.01243	0.00269 ± 0.00363	0.00077 ± 0.00114
Word2Vec	0.00875 ± 0.01349	0.00215 ± 0.00366	0.00052 ± 0.00102
Fasttext	0.00842 ± 0.01277	0.00210 ± 0.00346	0.00053 ± 0.00099

Table 4.25: AlexNet CNN Saliency Map Fraction of Elements Above Threshold

	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.00926 ± 0.01055	0.00408 ± 0.00422	0.00175 ± 0.00168
Word2Vec	0.02653 ± 0.05377	0.01125 ± 0.02470	0.00287 ± 0.00545
Fasttext	0.01352 ± 0.02013	0.00495 ± 0.00697	0.00178 ± 0.00209

Table 4.26: AlexNet CNN GradCam Map Fraction of Elements Above Threshold

	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.04973 ± 0.05356	0.02040 ± 0.01890	0.00526 ± 0.00451
Word2Vec	0.14036 ± 0.15735	0.10827 ± 0.13274	0.03344 ± 0.04670
Fasttext	0.06000 ± 0.08063	0.02478 ± 0.03308	0.00729 ± 0.00761

Table 4.27: AlexNet CNN Scorecam Map Fraction of Elements Above Threshold

	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.02393 ± 0.02519	0.00872 ± 0.00822	0.00283 ± 0.00262
Word2Vec	0.12880 ± 0.15348	0.08621 ± 0.11734	0.02371 ± 0.03849
Fasttext	0.05402 ± 0.06727	0.01769 ± 0.02226	0.00432 ± 0.00502

Table 4.28: AlexNet CNN FasterScorecam Map Fraction of Elements Above Threshold

Results Interpretation

Examining the AlexNet thresholds for the Word2Vec models, we observe that at a threshold of 0.4, the values are nearly reaching the maximum base of 0.14782, suggesting that a significant portion of the input is considered relevant when using this threshold of 0.4. For GloVe and FastText at thresholds of 0.6 and 0.8, we notice ratios close to the average base truth ratio of 0.00738, indicating that more accurate and relevant tokens can be found at these thresholds.

	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.01684 ± 0.01683	0.00545 ± 0.00521	0.00177 ± 0.00161
Word2Vec	0.01018 ± 0.01356	0.00281 ± 0.00379	0.00083 ± 0.00113
Fasttext	0.00012 ± 0.00302	0.00002 ± 0.00068	0.00000 ± 0.00015

Table 4.29: Mobilenet CNN Saliency Map Fraction of Elements Above Threshold

	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.00964 ± 0.01115	0.00420 ± 0.00441	0.00179 ± 0.00174
Word2Vec	0.02266 ± 0.03776	0.00854 ± 0.01427	0.00244 ± 0.00325
Fasttext	0.01192 ± 0.01757	0.00443 ± 0.00589	0.00167 ± 0.00180

Table 4.30: Mobilenet CNN GradCam Map Fraction of Elements Above Threshold

	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.05207 ± 0.05574	0.02137 ± 0.01934	0.00560 ± 0.00449
Word2Vec	0.13895 ± 0.15536	0.09804 ± 0.12190	0.02320 ± 0.03349
Fasttext	0.05610 ± 0.07925	0.02328 ± 0.03250	0.00610 ± 0.00791

Table 4.31: Mobilenet CNN Scorecam Map Fraction of Elements Above Threshold

	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.02496 ± 0.02508	0.00896 ± 0.00830	0.00283 ± 0.00267
Word2Vec	0.13186 ± 0.15076	0.08727 ± 0.11440	0.02468 ± 0.03821
Fasttext	0.05501 ± 0.06426	0.01775 ± 0.02076	0.00451 ± 0.00467

Table 4.32: Mobilenet CNN FasterScorecam Map Fraction of Elements Above Threshold

Results Interpretation

Examining the results from our Mobilenet models, we observe that the thresholds for Word2vec are 0.4 for all explainability methods, while ScoreCam and FastScoreCam have thresholds of 0.6 and 0.8, which are closer to the maximum base and farther from the base truth ratios, suggesting that a substantial portion of the input is considered relevant when using this threshold. Additionally, we note that thresholds of 0.6 and 0.8 are nearer to the base truth ratio, excluding ScoreCam and FastScoreCam at the 0.6 threshold.

	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.00034 ± 0.00209	0.00005 ± 0.00053	0.00001 ± 0.00016
Word2Vec	0.00004 ± 0.00081	0.00001 ± 0.00026	0.00000 ± 0.00008
Fasttext	0.00009 ± 0.00122	0.00002 ± 0.00036	0.00000 ± 0.00011

Table 4.33: Resnet18 CNN Saliency Map Fraction of Elements Above Threshold

	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.00997 ± 0.01056	0.00439 ± 0.00438	0.00183 ± 0.00177
Word2Vec	0.03172 ± 0.05877	0.01374 ± 0.02750	0.00364 ± 0.00580
Fasttext	0.01426 ± 0.02081	0.00526 ± 0.00715	0.00188 ± 0.00221

Table 4.34: Resnet18 CNN GradCam Map Fraction of Elements Above Threshold

	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.00956 ± 0.01017	0.00426 ± 0.00442	0.00175 ± 0.00181
Word2Vec	0.00225 ± 0.01287	0.00082 ± 0.00444	0.00026 ± 0.00107
Fasttext	0.03097 ± 0.04317	0.00977 ± 0.01348	0.00275 ± 0.00335

Table 4.35: Resnet18 CNN Scorecam Map Fraction of Elements Above Threshold

	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.01000 ± 0.01059	0.00442 ± 0.00447	0.00181 ± 0.00183
Word2Vec	0.00182 ± 0.00650	0.00073 ±0.00232	0.00028 ± 0.00077
Fasttext	0.02393 ± 0.03587	0.00778 ± 0.01180	0.00231 ± 0.00322

Table 4.36: Resnet18 CNN FasterScorecam Map Fraction of Elements Above Threshold

Results Interpretation

Analyzing the results from Resnet18 reveals that most saliency thresholds were not identified as significant features across all thresholds, resulting in an unusable saliency map for the Resnet model. Additionally, we observe that the thresholds for Word2vec

on GradCam at 0.4 and 0.6, as well as Fasttext on ScoreCam and FastScoreCam at 0.4, are nearing the maximum base and farther from the baseline truth ratio.

We also note that, on average, the thresholds are significantly lower than our base fraction of 0.14782, indicating that the models identify only a select few features as significant contributors to the predictions while disregarding others, such as zero padding. This trend is further illustrated by the low values observed in the heat map shown in Figure 116.

4.3.3 Analysis of Data Threshold

In this section, we will further analyze the results obtained from our dataset, as well as those from the explainability methods applied to our top three models. Figure 4.76 shows a histogram generated from our dataset of Java method tokens. Our machine learning models were designed to accept a maximum input of 1225 tokens, so the histogram displays methods ranging from 5 tokens to 1,225 tokens. We observe that most methods fall within the first quartile, with lengths of around 53 tokens, and the third quartile, with lengths of approximately 230 tokens, as shown in Figure 4.77. There are also a few outliers, with lengths reaching up to 460 tokens.

Figure 4.78 presents a normalized distribution of our dataset, which has been adjusted using the maximum size of the largest method, comprising 1225 tokens. We observe a common relationship between this normalized data and our formula $f(t)$, used for calculating the fraction of elements above a specified threshold, as both datasets are normalized based on the maximum length of 1225 tokens.

Additionally, the mean and standard deviations of 0.14722 and 0.16121 represent the average number of tokens in the input vector. For instance, we can reference the results from MobileNet Word2Vec with a threshold of 0.4, as shown in Table 4.31. We note that the mean fraction of elements considered by our model at this threshold is 0.13895, which is close to the mean value of our dataset. This trend is further illustrated in Figure 4.80.

Furthermore, we can observe that the model ignores the padded values, as demonstrated in Figure 117. These padded values exhibit the lowest importance scores.

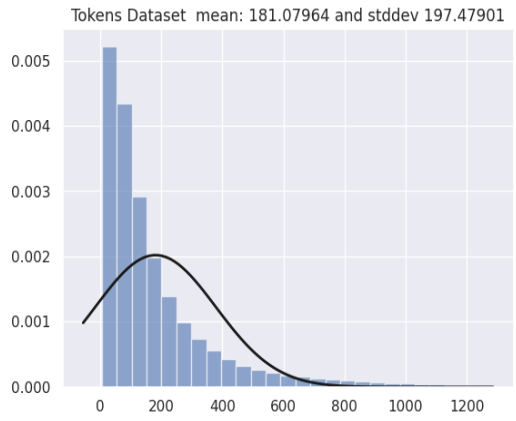


Figure 4.76: Represents a histogram of the lengths of each method tokenized before padding it to length 1225

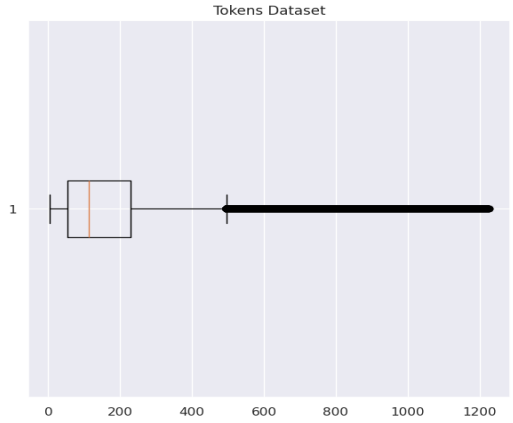


Figure 4.77: Represents a box and whisker plot of the lengths of each method tokenized before padding it to length 1225

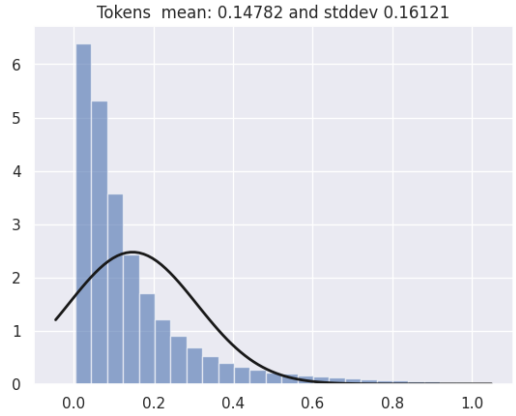


Figure 4.78: Represents a histogram of the normalized lengths of each method tokenized before padding it to length 1225

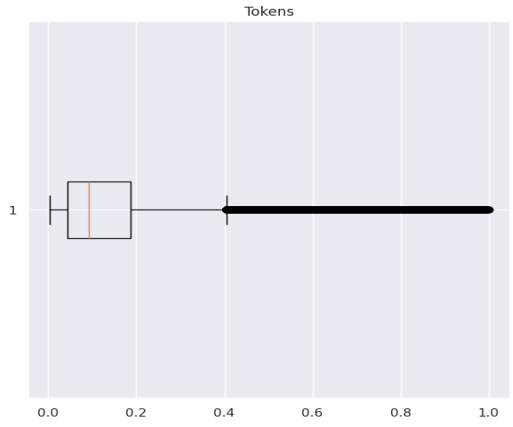


Figure 4.79: Represents a box and whisker plot of the normalized lengths of each method tokenized before padding it to length 1225

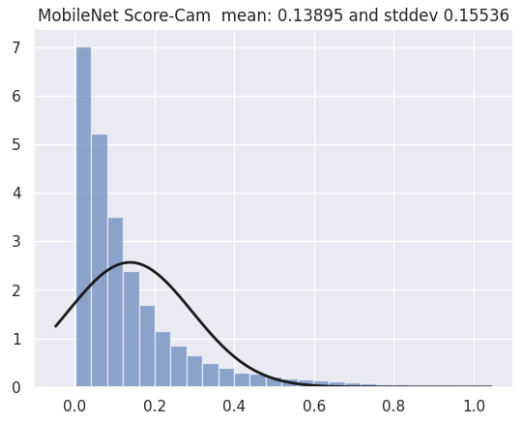


Figure 4.80: Represents a histogram of the MobileNet Fraction of Elements above threshold 0.4 using Score-CAM and Word2Vec embedding

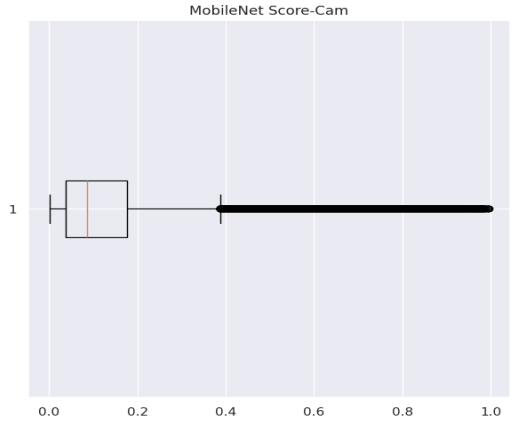


Figure 4.81: Represents a box and whisker plot of the MobileNet Fraction of Elements above threshold 0.4 using Score-CAM and Word2Vec embedding

4.3.4 Weights Mapping Percentage Thresholds

In this section we look at the mapping the weights of our explainability models to each token and mapping the tokens with high thresholds to lines containing bugs to calculate the percentage of correctly identified tokens to each respective line containing a bug. The percentage thresholds are calculated by $f(t)$.

$$f(t) = \frac{h(t)}{g(t)}$$

where : g(t) is the number of buggy lines picked up given the threshold t, and h(t) is the number of true lines with bugs given the threshold t.

	$f(0.2)$	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.2600 ± 0.2516	0.2766 ± 0.2819	0.2907 ± 0.3031	0.3338 ± 0.3315
Word2Vec	0.2526 ± 0.2459	0.2824 ± 0.2908	0.2816 ± 0.3051	0.3044 ± 0.3252
Fasttext	0.2707 ± 0.2514	0.2972 ±0.2997	0.2908 ± 0.3139	0.2935 ± 0.3235

Table 4.37: AlexNet CNN Saliency Map Weights mapping percentage Threshold

	$f(0.2)$	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.2973 ± 0.2886	0.3119 ± 0.30140	0.3396 ± 0.3203	0.3912 ± 0.3458
Word2Vec	0.2702 ± 0.2847	0.2823 ± 0.2948	0.3035 ± 0.3120	0.3559 ± 0.3406
Fasttext	0.2873 ± 0.2722	0.3012 ± 0.2919	0.3182 ± 0.3117	0.3561 ± 0.3374

Table 4.38: AlexNet CNN GradCam Map Weights mapping percentage Threshold

	$f(0.2)$	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.2733 ± 0.2671	0.2786 ± 0.2702	0.2935 ± 0.2805	0.3321 ± 0.3074
Word2Vec	0.2199 ± 0.2075	0.2227 ± 0.2116	0.2401 ± 0.2328	0.2754 ± 0.2741
Fasttext	0.2385 ± 0.2164	0.2569 ± 0.2405	0.2755 ± 0.2610	0.3176 ± 0.3093

Table 4.39: AlexNet CNN Scorecam Map Weights mapping percentage Threshold

	$f(0.2)$	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.2787 ± 0.2713	0.2930 ± 0.2836	0.2836 ± 0.2999	0.3640 ± 0.3263
Word2Vec	0.2199 ± 0.2075	0.2215 ± 0.2101	0.2405 ± 0.2375	0.2821 ± 0.2824
Fasttext	0.2469 ± 0.2241	0.2626 ± 0.2461	0.3009 ± 0.2923	0.3696 ± 0.3360

Table 4.40: AlexNet CNN FasterScorecam Map Weights mapping percentage Threshold

	$f(0.2)$	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.2319 ± 0.2316	0.2529 ± 0.2554	0.2799 ± 0.2826	0.3294 ± 0.3191
Word2Vec	0.2371 ± 0.2368	0.2814 ± 0.2783	0.3517 ± 0.3291	0.4428 ± 0.3686
Fasttext	0.1048 ± 0.2041	0.1007 ± 0.2178	0.1019 ± 0.2171	0.1089 ± 0.2179

Table 4.41: Mobilenet CNN Saliency Map Weights mapping percentage Threshold

	$f(0.2)$	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.2670 ± 0.2760	0.2786 ± 0.2850	0.2999 ± 0.2987	0.3499 ± 0.3259
Word2Vec	0.2390 ± 0.2464	0.2525 ± 0.2606	0.2751 ±0.2812	0.3288 ± 0.3172
Fasttext	0.2379 ± 0.2453	0.2553 ± 0.2635	0.2846 ± 0.2889	0.3366 ± 0.3225

Table 4.42: Mobilenet CNN GradCam Map Weights mapping percentage Threshold

	$f(0.2)$	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.2503 ± 0.2558	0.2540 ± 0.2581	0.2682 ± 0.2673	0.3015 ± 0.2886
Word2Vec	0.2083 ± 0.2032	0.2093 ± 0.2048	0.2213 ± 0.2219	0.2570 ± 0.2655
Fasttext	0.2090 ± 0.2046	0.2271 ± 0.2286	0.2448 ± 0.2421	0.2908 ± 0.2926

Table 4.43: Mobilenet CNN Scorecam Map Weights mapping percentage Threshold

	$f(0.2)$	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.2541 ± 0.2592	0.2691 ± 0.2696	0.2943 ± 0.2875	0.3563 ± 0.3258
Word2Vec	0.2083 ± 0.2032	0.2151 ± 0.2114	0.2294 ± 0.2335	0.2603 ± 0.2685
Fasttext	0.2091 ± 0.2051	0.2051 ± 0.2248	0.2411 ± 0.2451	0.3053 ± 0.2989

Table 4.44: Mobilenet CNN FasterScorecam Map Weights mapping percentage Threshold

	$f(0.2)$	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.1784 ± 0.2475	0.1306 ± 0.2254	0.1393 ± 0.2405	0.1351 ± 0.2440
Word2Vec	0.1188 ± 0.2181	0.1144 ± 0.2234	0.1204 ± 0.2355	0.1260 ± 0.2242
Fasttext	0.1823 ± 0.2765	0.1016 ± 0.2010	0.1290 ± 0.2408	0.1537 ± 0.2668

Table 4.45: Resnet18 CNN Saliency Map Weights mapping percentage Threshold

	$f(0.2)$	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.2670 ± 0.2750	0.2740 ± 0.2825	0.2955 ± 0.29655	0.3473 ± 0.3233
Word2Vec	0.2196 ± 0.2406	0.2271 ± 0.2486	0.2427 ± 0.2622	0.2863 ± 0.2947
Fasttext	0.2287 ± 0.2434	0.2401 ± 0.2545	0.2636 ± 0.2734	0.3101 ± 0.3044

Table 4.46: Resnet18 CNN GradCam Map Weights mapping percentage Threshold

	$f(0.2)$	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.2685 ± 0.2717	0.2828 ± 0.2886	0.2771 ± 0.284	0.3148 ± 0.3074
Word2Vec	0.2795 ± 0.3198	0.2897 ± 0.3291	0.3097 ± 0.3394	0.3731 ± 0.3645
Fasttext	0.2195 ± 0.2210	0.2441 ± 0.2542	0.2721 ± 0.2806	0.2967 ± 0.2926

Table 4.47: Resnet18 CNN Scorecam Map Weights mapping percentage Threshold

	$f(0.2)$	$f(0.4)$	$f(0.6)$	$f(0.8)$
	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$	$\mu \pm \sigma$
GloVe	0.2737 ± 0.2806	0.2798 ± 0.2849	0.3004 ± 0.2987	0.3391 ± 0.3209
Word2Vec	0.2805 ± 0.2806	0.2978 ± 0.2875	0.3362 ± 0.3097	0.3818 ± 0.3336
Fasttext	0.1958 ± 0.2102	0.2045 ± 0.2209	0.2367 ± 0.2504	0.2967 ± 0.2933

Table 4.48: Resnet18 CNN FasterScorecam Map Weights mapping percentage Threshold

Results Interpretation

Analyzing the results from the weights mapping Tables reveals that as we increase the thresholds, the percentages also rise, indicating a decrease in the number of tokens mapped to the lines of code containing bugs.

Examining the embedding models looking at the highest threshold for AlexNet we obtain the following averages: 0.3552 with GloVe, 0.3044 with Word2Vec, and 0.3342 with FastText. For MobileNet, the averages are 0.3342 with GloVe, 0.3222 with Word2Vec, and 0.2604 with FastText. Finally, for ResNet18, the averages are 0.2840 with GloVe, 0.2918 with Word2Vec, and 0.2643 with FastText.

4.4 Results Analysis

4.4.1 RQ1 What are the current state-of-the-art methods in the embedded representation of the source code?

In this research question we want to investigate the embedding methods we can use for representing and embedding source code. Our approach to this was to go with tokenizing the source code snippets and using Word2Vec, FastText, and GloVe to train our embeddings on the tokens, we used a lower embedding dimension to make it easier to map the final results.

4.4.2 RQ2 What are the current state-of-the-art machine learning algorithms operating on embedded source code representation for defect detection?

In this research question, we look at different neural network architectures, with a focus on convolutional neural networks. Our approach was to look at different literature, select a few performing architectures, implement them, and train our models. From the different architectures, we selected the AlexNet, MobileNet, and ResNet18 models, which managed to learn the data set in the number of epochs provided, compared to the other models from the results shown in Section 4.1.

4.4.3 RQ3 Which metrics can be used to assess and measure the performance of the machine learning algorithms in source code defect detection?

In this research question we look at the different metrics we can use to measure our models performance.Our approach was to look at the traditional metrics Accuracy, Precision, Recall, F1 score and AUC score to measure the performance of the model

on the dataset, along with custom metrics to measure the importance of each value in the input token to the models prediction. These custom metrics which are focused on the heatmap results from our selected explainability models are Intersection of important features of the same architecture given different embeddings, and the percentage of features taken into consideration by the model to make a prediction.

4.4.4 RQ4 From the current state-of-the-art defect detection methods, which one performs best based on these metrics?

In this research question we will be using the results from RQ3 to answer it. For our experiments the embedding models where trained on the same dataset, and the different architecture where kept the same with the difference being the embeddings of the input meaning with each architecture we trained three models one with Word2Vec, another Fasttext and third with GloVe. All the models were trained to a maximum of 300 epochs, and the input vector max length was 1225.

4.5 Summary

Section 4.1 shows the results from our traditional metrics, where three architectures, AlexNet, MobileNet, and ResNet18, outperformed the others on the dataset. Among them, MobileNet achieved the highest performance when averaging across embeddings. Visualization metrics (Section 4.3) were therefore examined only for these top-performing architectures.

In terms of the relationship between the three embedding models given each architecture, we see an overall strong correlation between the models trained with the Word2Vec and Fasttext embeddings. Analysis of the proportion of how many tokens were taken into consideration in the input vector to make a prediction we see consistency among the models with fractions far below our base fraction of inputs implying that the padding and most of the input features were given less weight towards the prediction of whether the snippet contains a defect.

Together, these results highlight that both architectural choice and embedding

selection play a decisive role in model performance. Our findings parallel those of Ngoc et al. [61], who demonstrated the impact of embeddings on software vulnerability detection, with FastText showing slightly stronger results. Similarly, we corroborate Yang et al.[72], who argued that deeper architectures do not necessarily guarantee better performance. In our case, lightweight models such as MobileNet matched and exceeded the performance of deeper networks such as VGG, while requiring far fewer parameters. These insights underscore the importance of balancing model complexity with efficiency in developing scalable approaches to software defect detection.

Chapter 5

Conclusion

In this study, we explore the use of convolutional neural networks and embedding models to train defect prediction models on Java methods code snippets, along with exploring how different explainability models visualize the importance of each input towards the prediction of the model. For our embedding models, we focused on Word2Vec, FastText and GloVe. We also examined different CNN architectures, simple CNN architectures for our baseline, and more complex architectures like AlexNet, VGG16, Mobilenet, and Resnet18. And for our explainability models we went with Saliency maps, GradCam, ScoreCam and FIMF score-CAM.

Based on our models performance looking at the Accuracy, Precision, Recall, F1 Score and AUC score we decided to go on with the explainability visual analysis on the AlexNet, Mobilenet and Resnet18 models to investigate the intersection of importance of the models given different embeddings along with the proportion of tokens considered by each model for its prediction. And finally to see if we can use the tokens with high importance to pinpoint the line with defect from a given method snippet.

From our experiments, we notice that we were able to map some of those highly weighted features that contributed to the prediction to the line of code with defects. While convolutional neural network-based approaches were able to capture local token-level features, it would be beneficial to expand the study also to capture the structural relationships of source codes. Future research would extend to explainability to Recurrent Neural Networks (RNNs), Graph Neural Networks (GNNs), and Transformer-based models. To provide insights into these architectures, examining methods such as Integrated Gradients, Shapley Additive Explanations (SHAP), GNNExplainer, and attention visualization would help enrich insights about the models. We can also consider expanding the dimensions of the embedding models to

better capture the relationship and context of the input, while combining the token representations.

Bibliography

[1] M. A. Nielsen, *Neural networks and deep learning*. Determination press San Francisco, CA, USA, 2015, vol. 25.

[2] M. Sit, B. Demiray, Z. Xiang, G. Ewing, Y. Sermet, and I. Demir, "A comprehensive review of deep learning applications in hydrology and water resources," *Water Science and Technology*, vol. 82, 08 2020.

[3] P. Purwono, A. Ma'arif, W. Rahmaniar, H. Imam, H. I. K. Fathurrahman, A. Frisky, and Q. M. U. Haq, "Understanding of convolutional neural network (cnn): A review," *International Journal of Robotics and Control Systems*, vol. 2, pp. 739–748, 01 2023.

[4] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[5] S. Wang, W. Zhou, and C. Jiang, "A survey of word embeddings based on deep learning," *Computing*, vol. 102, 03 2020.

[6] O. Melamud, J. Goldberger, and I. Dagan, "context2vec: Learning generic context embedding with bidirectional lstm," 01 2016, pp. 51–61.

[7] M. Hemmer, H. V. Khang, K. Robbersmyr, T. Waag, and T. Meyer, "Fault classification of axial and radial roller bearings using transfer learning through a pretrained convolutional neural network," *Designs*, vol. 2, p. 56, 12 2018.

[8] D. Choi, W. Bell, D. Kim, and J. Kim, "Uav-driven structural crack detection and location determination using convolutional neural networks," *Sensors*, vol. 21, p. 2650, 04 2021.

[9] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural

networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[10] F. Ramzan, M. U. Khan, A. Rehmat, S. Iqbal, T. Saba, A. Rehman, and Z. Mehmood, "A deep learning approach for automated diagnosis and multi-class classification of alzheimer's disease stages using resting-state fmri and residual neural networks," *Journal of Medical Systems*, vol. 44, 12 2019.

[11] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *CoRR*, vol. abs/1812.08693, 2018. [Online]. Available: http://arxiv.org/abs/1812.08693

[12] V. Suma and T. Nair, "Effective defect prevention approach in software process for achieving better quality levels," *arXiv preprint arXiv:1001.3552*, 2010.

[13] A. Bhanushali, "Ensuring software quality through effective quality assurance testing: Best practices and case studies," *International Journal of Advances in Scientific Research and Engineering*, vol. 26, no. 1, 2023.

[14] M. A. Umar, "Comprehensive study of software testing: Categories, levels, techniques, and types," *International Journal of Advance Research, Ideas and Innovations in Technology*, vol. 5, no. 6, pp. 32–40, 2019.

[15] Q. S.M.K and S. U. Farooq, "Software testing – goals, principles, and limitations," *International Journal of Computer Applications*, vol. 6, 09 2010.

[16] N. Akhtar, A. Rana, P. P. Deshpande, M. Kumar, P. K. Parida, and K. K. Bajaj, "Software bug prediction and detection using machine learning and deep learning," *International Journal of Intelligent Systems and Applications in Engineering*, vol. 12, no. 9s, p. 301–308, Dec. 2023. [Online]. Available: https://ijisae.org/index.php/IJISAE/article/view/4277

[17] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 297–308.

[18] Z. Chen and M. Monperrus, "A literature study of embeddings on source code," 2019.

[19] ——, "The remarkable role of similarity in redundancy-based program repair," *CoRR*, vol. abs/1811.05703, 2018. [Online]. Available: http://arxiv.org/abs/1811.05703

[20] D. Azcona, P. Arora, I.-H. Hsiao, and A. Smeaton, "user2code2vec: Embeddings for profiling students based on distributional representations of source code," in *Proceedings of the 9th International Conference on Learning Analytics & Knowledge*, ser. LAK19. New York, NY, USA: Association for Computing Machinery, 2019, p. 86–95. [Online]. Available: https://doi.org/10.1145/3303772.3303813

[21] G. Fischer, J. Lusiardi, and J. Wolff von Gudenberg, "Abstract syntax trees - and their role in model driven software development," in *International Conference on Software Engineering Advances (ICSEA 2007)*, 2007, pp. 38–38.

[22] G. Tao, D. Guowei, Q. Hu, and C. Baojiang, "Improved plagiarism detection algorithm based on abstract syntax tree," in *2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies*, 2013, pp. 714–719.

[23] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, "Software defect prediction via attention-based recurrent neural network," *Scientific Programming*, vol. 2019, pp. 1–14, 04 2019.

[24] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2017, pp. 318–328.

[25] M. Pradel and K. Sen, "Deep learning to find bugs," *TU Darmstadt, Department of Computer Science*, vol. 4, p. 1, 2017.

[26] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring api embedding for api usages and applications," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 438–449.

[27] M. Xue and C. Zhu, "A study and application on machine learning of artificial intellligence," in *2009 International Joint Conference on Artificial Intelligence*, 2009, pp. 272–274.

[28] A. Chug and S. Dhall, "Software defect prediction using supervised learning algorithm and unsupervised learning algorithm," in *Confluence 2013: The Next Generation Information Technology Summit (4th International Conference)*, 2013, pp. 173–179.

[29] M. A. El Mrabet, K. El Makkaoui, and A. Faize, "Supervised machine learning: A survey," in *2021 4th International Conference on Advanced Communication Technologies and Networking (CommNet)*, 2021, pp. 1–10.

[30] J. Kauffmann, M. Esders, L. Ruff, G. Montavon, W. Samek, and K.-R. Müller, "From clustering to cluster explanations via neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, no. 2, pp. 1926–1940, 2024.

[31] Y. H. Lee and J. S. Chahl, "Performance evaluation dimensional reduction techniques for image classification," in *2021 International Conference on Machine Learning and Cybernetics (ICMLC)*, 2021, pp. 1–6.

[32] S. Chippalakatti, C. H. Renumadhavi, and A. Pallavi, "Comparison of unsupervised machine learning algorithm f or dimensionality reduction," in *2022 International Conference on Knowledge Engineering and Communication Systems (ICKES)*, 2022, pp. 1–7.

[33] S. K. Alhabeeb and A. A. Al-Shargabi, "Text-to-image synthesis with generative models: Methods, datasets, performance metrics, challenges, and future direction," *IEEE Access*, vol. 12, pp. 24 412–24 427, 2024.

[34] R. Uhrig, "Introduction to artificial neural networks," in *Proceedings of IECON '95 - 21st Annual Conference on IEEE Industrial Electronics*, vol. 1, 1995, pp. 33–37 vol.1.

[35] M. Solazzi and A. Uncini, "Artificial neural networks with adaptive multidimensional spline activation functions," in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, vol. 3, 2000, pp. 471–476 vol.3.

[36] Huang Yi, Sun Shiyu, Duan Xiusheng, and Chen Zhigang, "A study on deep neural networks framework," in *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, 2016, pp. 1519–1522.

[37] N. Aloysius and M. Geetha, "A review on deep convolutional neural networks," in *2017 International Conference on Communication and Signal Processing (ICCSP)*, 2017, pp. 0588–0592.

[38] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 783–794.

[39] P. R. and T. Mg, "Ast-based and token-based neural networks for source code classification: A comparative performance analysis," 06 2024, pp. 1–7.

[40] A. Hidayat. Esprima lexical and syntactical analysis platform https://esprima.org/index.html. [Online]. Available: https://esprima.org/index.html

[41] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *CoRR*, vol. abs/1310.4546, 2013. [Online]. Available: http://arxiv.org/abs/1310.4546

[42] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[43] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[44] O. Melamud, J. Goldberger, and I. Dagan, "context2vec: Learning generic context embedding with bidirectional lstm," in *Proceedings of the 20th SIGNLL conference on computational natural language learning*, 2016, pp. 51–61.

[45] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," *CoRR*, vol. abs/1802.05365, 2018. [Online]. Available: http://arxiv.org/abs/1802.05365

[46] P. Gohel, P. Singh, and M. Mohanty, "Explainable AI: current status and future directions," *CoRR*, vol. abs/2107.07045, 2021. [Online]. Available: https://arxiv.org/abs/2107.07045

[47] Y. Lu and T. Ebrahimi, "Explanation of face recognition via saliency maps," 2023. [Online]. Available: https://arxiv.org/abs/2304.06118

[48] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," 2015. [Online]. Available: https://arxiv.org/abs/1512.04150

[49] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," *International Journal of Computer Vision*, vol. 128, no. 2, p. 336–359, Oct. 2019. [Online]. Available: http://dx.doi.org/10.1007/s11263-019-01228-7

[50] H. Wang, Z. Wang, M. Du, F. Yang, Z. Zhang, S. Ding, P. Mardziel, and X. Hu, "Score-cam: Score-weighted visual explanations for convolutional neural networks," 2020. [Online]. Available: https://arxiv.org/abs/1910.01279

[51] R. C. Fong and A. Vedaldi, "Interpretable explanations of black boxes by meaningful perturbation," in *2017 IEEE International Conference on Computer Vision (ICCV)*. IEEE, Oct. 2017. [Online]. Available: http://dx.doi.org/10.1109/ICCV.2017.371

[52] J. Li and D. Zhang, "Fimf score-cam: Score-cam based visual explanations via fast integrating multiple features of local space for deep networks," in *Pattern Recognition and Computer Vision: 5th Chinese Conference, PRCV 2022, Shenzhen, China, November 4–7, 2022, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 106–117. [Online]. Available: https://doi.org/10.1007/978-3-031-18907-4_9

[53] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[54] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:14124313

[55] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[56] M. R. Kounte, P. K. Tripathy, P. P., and H. Bajpai, "Analysis of intelligent machines using deep learning and natural language processing," in *2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184)*, 2020, pp. 956–960.

[57] M. G. Prakash, A. Ponmalar, S. Deeba, A. Akilandeswari, S. B. M. Rasool, and P. Lavanya, "Vscode- code with voice using natural language processing(nlp)," in *2022 International Conference on Computer, Power and Communications (ICCPC)*, 2022, pp. 571–574.

[58] G. Vadlamudi, N. Vemuru, S. Vangapalli, R. K. Surapaneni, and S. Nimmagadda, "Meeting summarizer using natural language processing," in *2022 6th International Conference on Trends in Electronics and Informatics (ICOEI)*, 2022, pp. 1610–1614.

[59] B. Kynabay, A. Aldabergen, and A. Zhamanov, "Automatic summarizing the news from inform.kz by using natural language processing tools," in *2021 IEEE International Conference on Smart Information Systems and Technologies (SIST)*, 2021, pp. 1–4.

[60] Y. Tang, H. Zhou, and H. Su, "Automatic classification of software bug reports based on lda and word2vec," in *2022 2nd International Conference on Computer Science, Electronic Information Engineering and Intelligent Control Technology (CEI)*, 2022, pp. 491–495.

[61] H. N. Ngoc, H. N. Viet, and T. Uehara, "An extended benchmark system of word embedding methods for vulnerability detection," in *Proceedings of the 4th International Conference on Future Networks and Distributed Systems*, ser. ICFNDS '20. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3440749.3442661

[62] Y. JianJie and W. Le, "Code defect detection method based on bert and ensemble," in *2023 9th International Conference on Computer and Communications (ICCC)*, 2023, pp. 2130–2138.

[63] H. Li, X. Li, X. Chen, X. Xie, Y. Mu, and Z. Feng, "Cross-project defect prediction via asttoken2vec and blstm-based neural network," in *2019 International Joint Conference on Neural Networks (IJCNN)*, 2019, pp. 1–8.

[64] G. Liu, Y. Lu, K. Shi, J. Chang, and X. Wei, "Convolutional neural networks-based locating relevant buggy code files for bug reports affected by data imbalance," *IEEE Access*, vol. 7, pp. 131 304–131 316, 2019.

[65] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, "Deep semantic feature learning with embedded static metrics for software defect prediction," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, 2019, pp. 244–251.

[66] S. Qiu, H. Huang, W. Jiang, F. Zhang, and W. Zhou, "Defect prediction via tree-based encoding with hybrid granularity for software sustainability," *IEEE Transactions on Sustainable Computing*, vol. 9, no. 3, pp. 249–260, 2024.

[67] V. Petsiuk, R. Jain, V. Manjunatha, V. I. Morariu, A. Mehra, V. Ordonez, and K. Saenko, "Black-box explanation of object detectors via saliency maps," 2021. [Online]. Available: https://arxiv.org/abs/2006.03204

[68] L. Brocki and N. C. Chung, "Concept saliency maps to visualize relevant features in deep generative models," 2019. [Online]. Available: https://arxiv.org/abs/1910.13140

[69] S. Yin, L. Wang, M. Shafiq, L. Teng, A. A. Laghari, and M. F. Khan, "G2grad-camrl: An object detection and interpretation model based on gradient-weighted class activation mapping and reinforcement learning in remote sensing images," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 16, pp. 3583–3598, 2023.

[70] S. Tasneem and K. A. Islam, "Development of trustable deep learning model in remote sensing through explainable-ai method selection," in *2023 IEEE 14th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, 2023, pp. 0261–0267.

[71] S. Soomro, A. Niaz, and K. Nam Choi, "Grad++scorecam: Enhancing visual explanations of deep convolutional networks using incremented gradient and score- weighted methods," *IEEE Access*, vol. 12, pp. 61 104–61 112, 2024.

[72] Y. Yang, L. Zhang, M. Du, J. Bo, H. Liu, L. Ren, X. Li, and M. J. Deen, "A comparative analysis of eleven neural networks architectures for small datasets of lung images of covid-19 patients toward improved clinical decisions," *Computers in Biology and Medicine*, vol. 139, p. 104887, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010482521006818

[73] M. Giammarco, F. Martinelli, A. Santone, M. Cesarelli, and F. Mercaldo, "Colon cancer diagnosis by means of explainable deep learning," *Scientific Reports*, vol. 14, 07 2024.

[74] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.

[75] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50, http://is.muni.cz/publication/884893/en.

[76] N. Dingwall and C. Potts, "Mittens: An extension of glove for learning domain-specialized representations," 03 2018.

[77] L. Górski, S. Ramakrishna, and J. M. Nowosielski, "Towards grad-cam based explainability in a legal text processing pipeline," *CoRR*, vol. abs/2012.09603, 2020. [Online]. Available: https://arxiv.org/abs/2012.09603

Appendix A

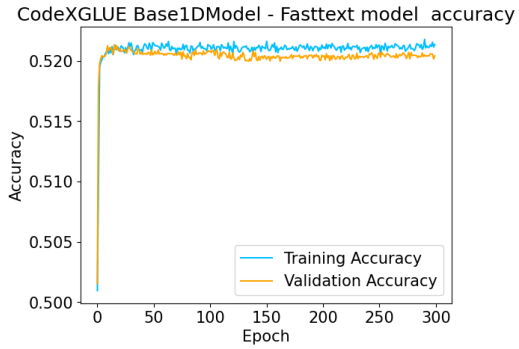


Figure 1: 1DBasemodel Fasttext Model Accuracy vs Epoch

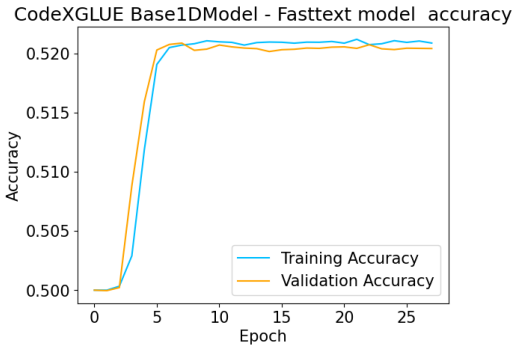


Figure 2: 1DBasemodel Fasttext earlystop Model Accuracy vs Epoch

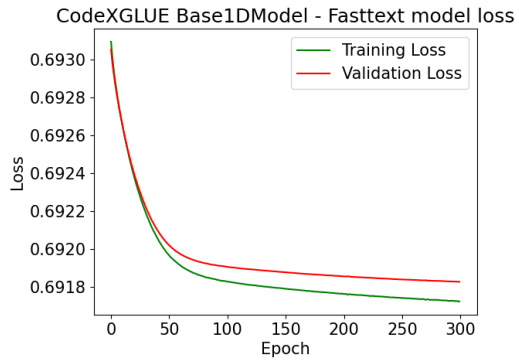


Figure 3: 1DBasemodel Fasttext Model Loss vs Epoch

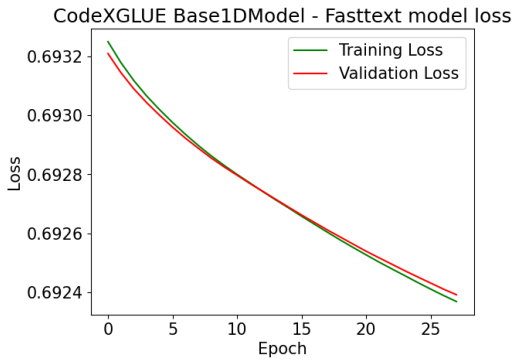


Figure 4: 1DBasemodel Fasttext earlystop Model Loss vs Epoch

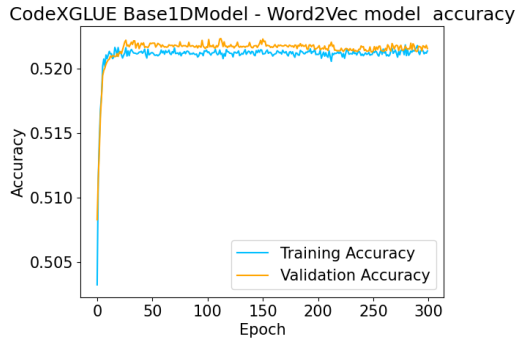


Figure 5: 1DBasemodel Word2Vec Model Accuracy vs Epoch

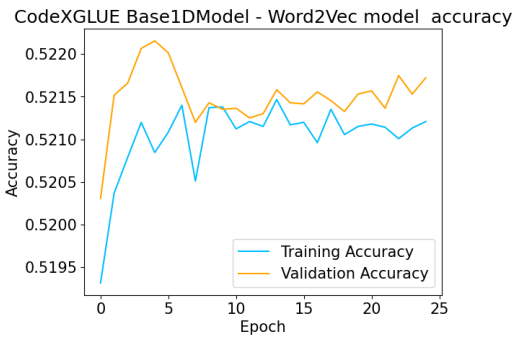


Figure 6: 1DBasemodel Word2Vec earlystop Model Accuracy vs Epoch

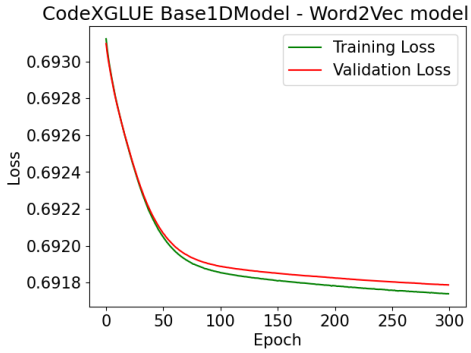


Figure 7: 1DBasemodel Word2Vec Model Loss vs Epoch

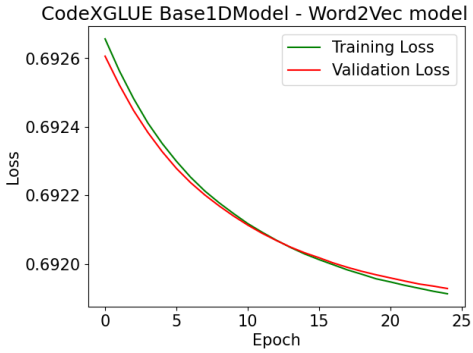


Figure 8: 1DBasemodel Word2Vec earlystop Model Loss vs Epoch

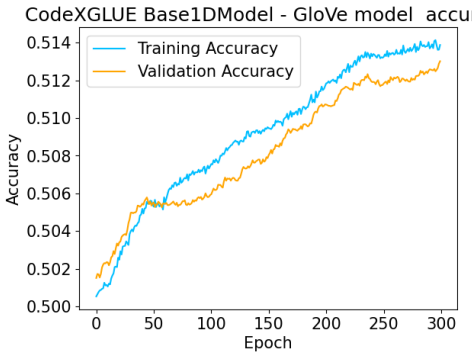


Figure 9: 1DBasemodel GloVe Model Accuracy vs Epoch

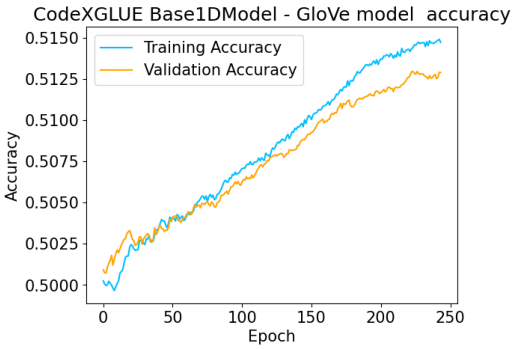


Figure 10: 1DBasemodel GloVe earlystop Model Accuracy vs Epoch

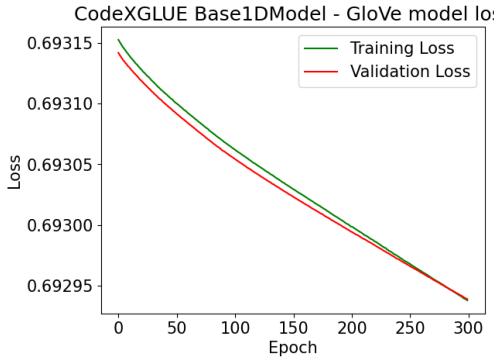


Figure 11: 1DBasemodel GloVe Model Accuracy vs Epoch

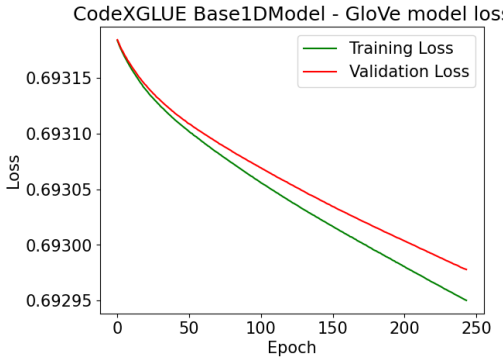


Figure 12: 1DBasemodel GloVe earlystop Model Loss vs Epoch

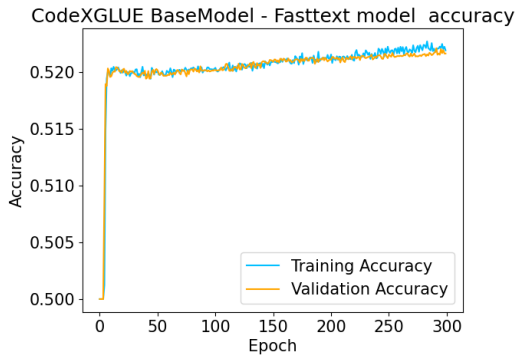


Figure 13: 2D Basemodel Fasttext Model Accuracy vs Epoch

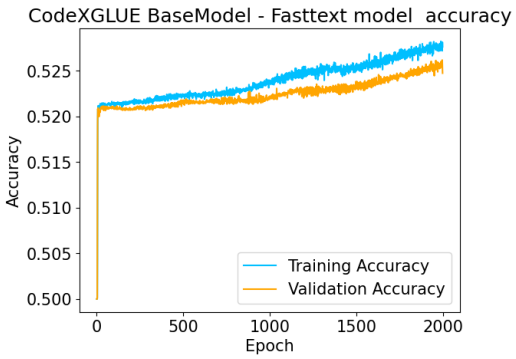


Figure 14: 2D Basemodel Fasttext earlystop Model Accuracy vs Epoch

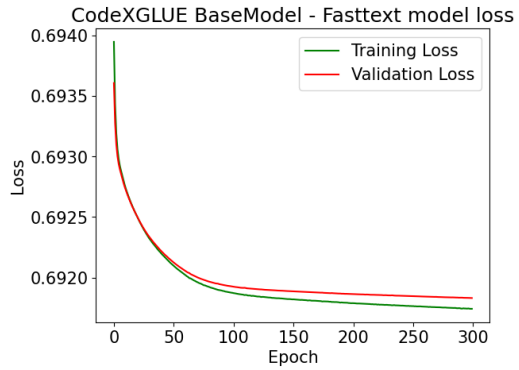


Figure 15: 2D Basemodel Fasttext Model Loss vs Epoch

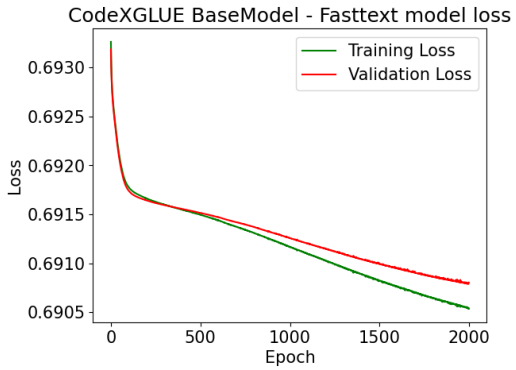


Figure 16: 2D Basemodel Fasttext earlystop Model Loss vs Epoch

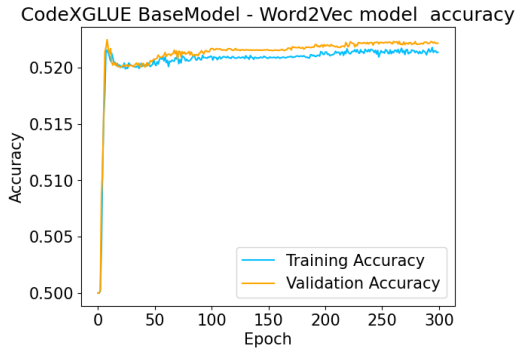


Figure 17: 2D Basemodel Word2Vec Model Accuracy vs Epoch

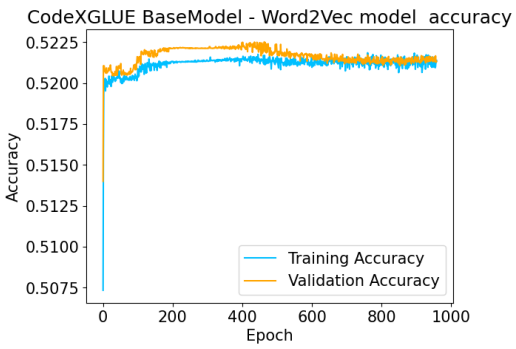


Figure 18: 2D Basemodel Word2Vec earlystop Model Accuracy vs Epoch

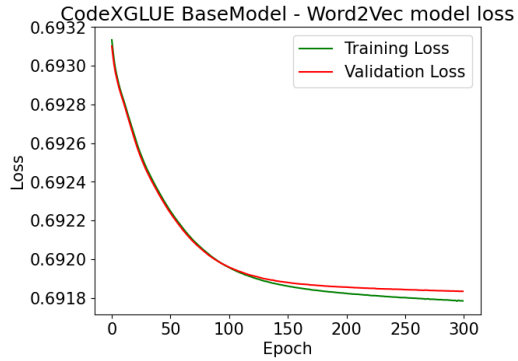


Figure 19: 2D Basemodel Word2Vec Model Loss vs Epoch

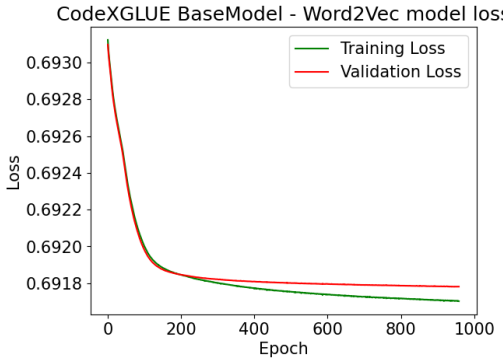


Figure 20: 2D Basemodel Word2Vec earlystop Model Loss vs Epoch

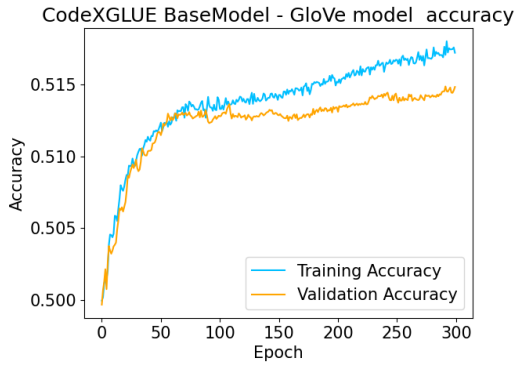


Figure 21: 2D Basemodel GloVe Model Accuracy vs Epoch

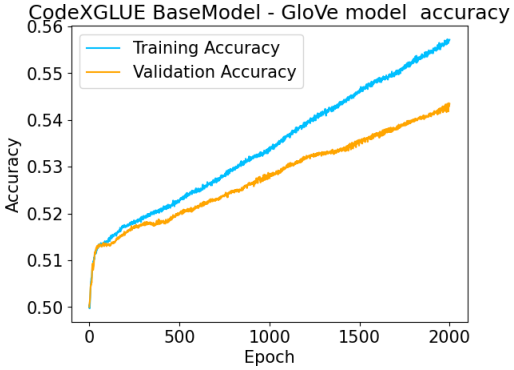


Figure 22: 2D Basemodel GloVe earlystop Model Accuracy vs Epoch

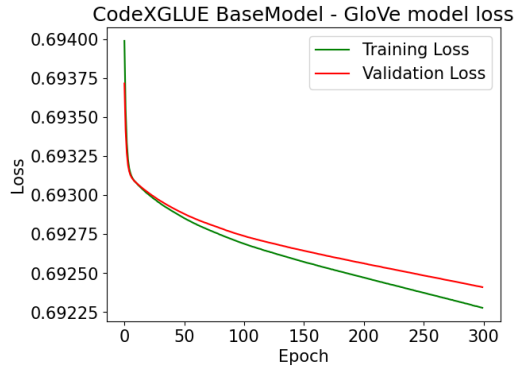


Figure 23: 2D Basemodel GloVe Model Loss vs Epoch

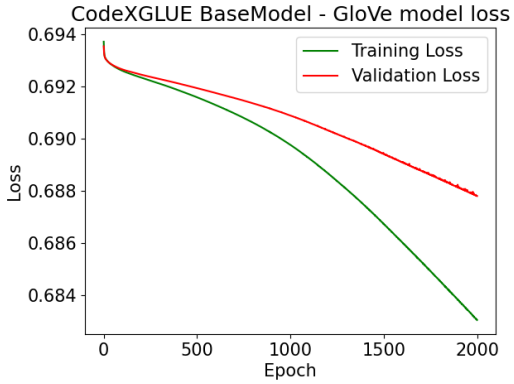


Figure 24: 2D Basemodel GloVe earlystop Model Loss vs Epoch

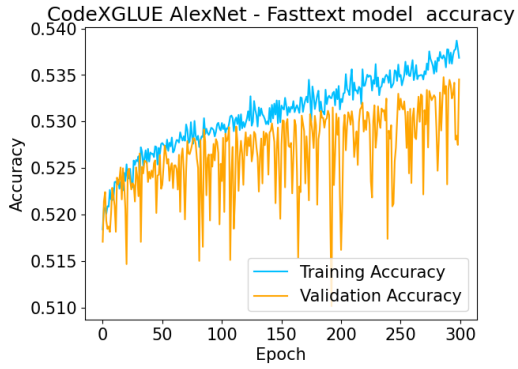


Figure 25: AlexNet Fasttext Model Accuracy vs Epoch

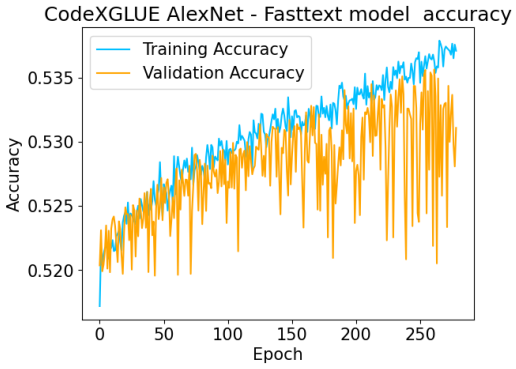


Figure 26: AlexNet Fasttext earlystop Model Accuracy vs Epoch

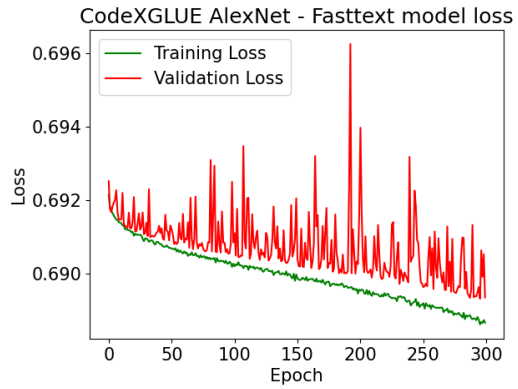


Figure 27: AlexNet Fasttext Model Loss vs Epoch

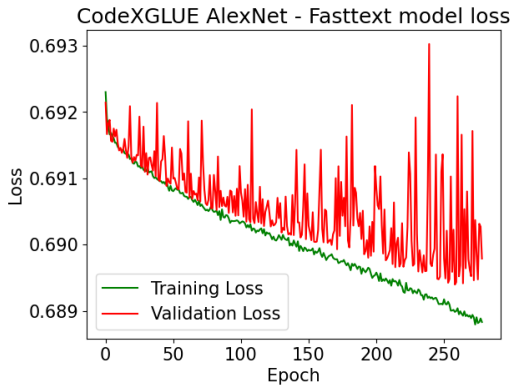


Figure 28: AlexNet Fasttext earlystop Model Loss vs Epoch

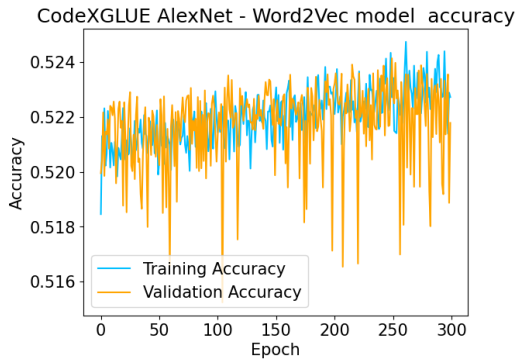


Figure 29: AlexNet Word2Vec Model Accuracy vs Epoch

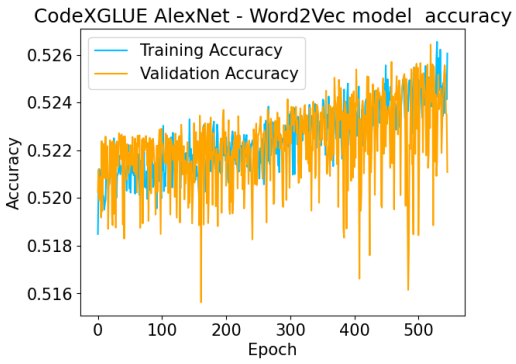


Figure 30: AlexNet Word2Vec earlystop Model Accuracy vs Epoch

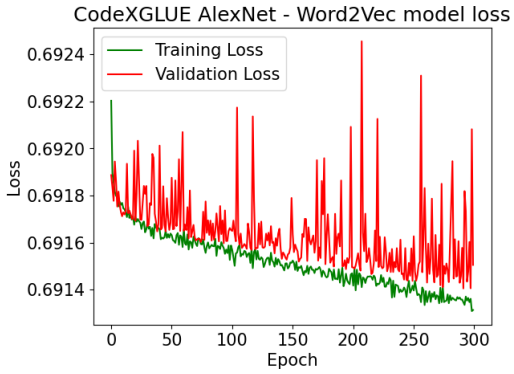


Figure 31: AlexNet Word2Vec Model Loss vs Epoch

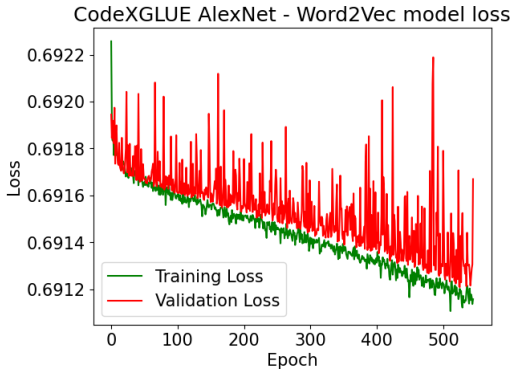


Figure 32: AlexNet Word2Vec earlystop Model Loss vs Epoch

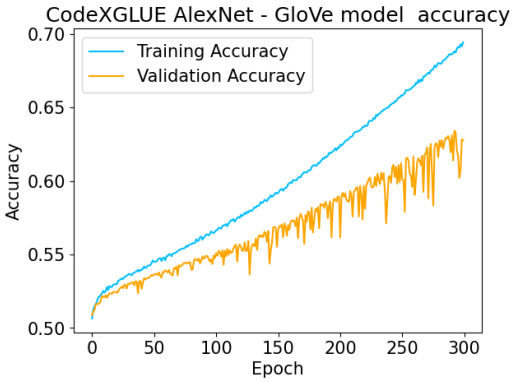


Figure 33: AlexNet GloVe Model Accuracy vs Epoch

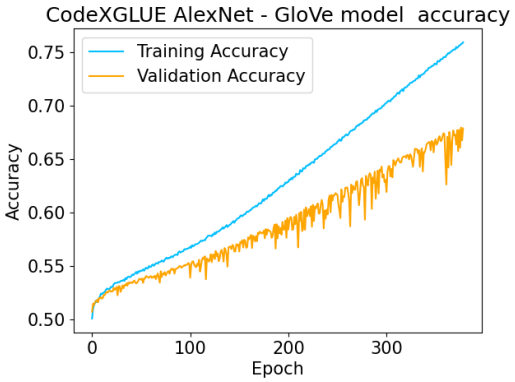


Figure 34: AlexNet GloVe earlystop Model Accuracy vs Epoch

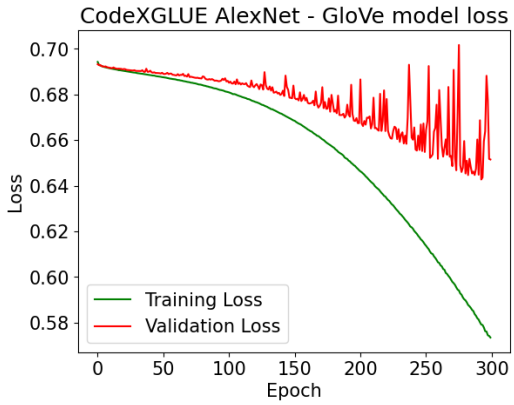


Figure 35: AlexNet GloVe Model Loss vs Epoch

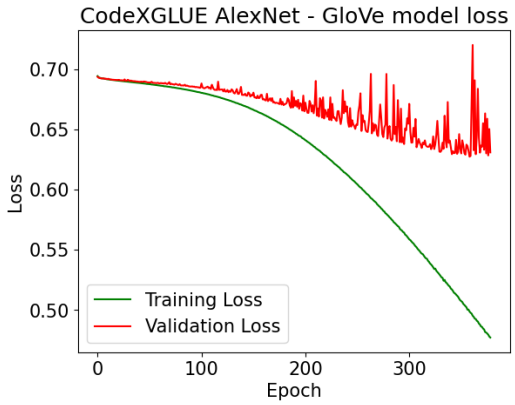


Figure 36: AlexNet GloVe earlystop Model Loss vs Epoch

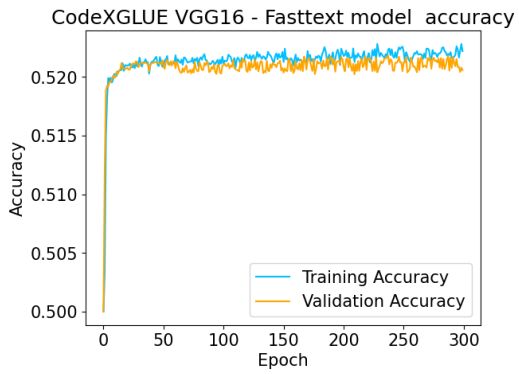


Figure 37: VGG16 Fasttext Model Accuracy vs Epoch

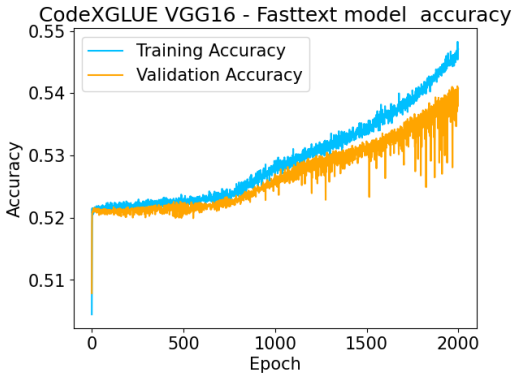


Figure 38: VGG16 Fasttext earlystop Model Accuracy vs Epoch

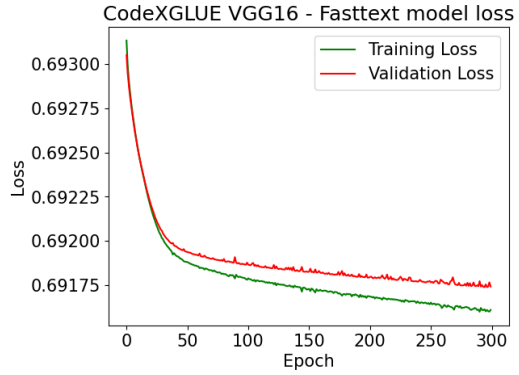


Figure 39: VGG16 Fasttext Model Loss vs Epoch

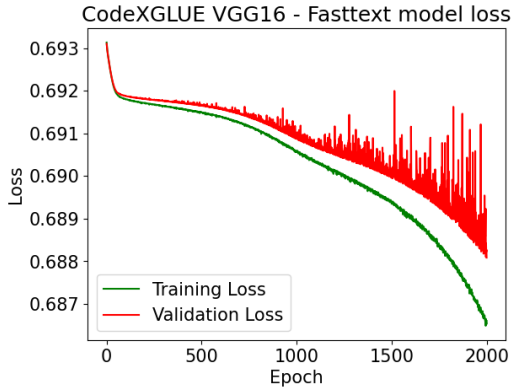


Figure 40: VGG16 Fasttext earlystop Model Loss vs Epoch

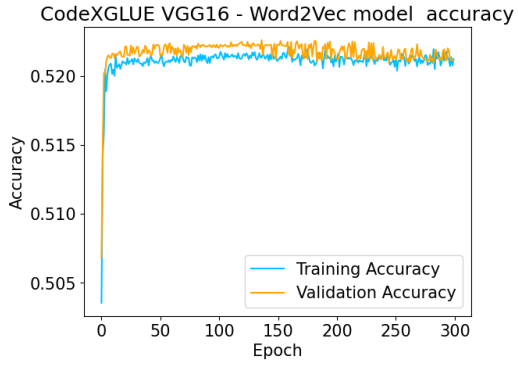


Figure 41: VGG16 Word2Vec Model Accuracy vs Epoch

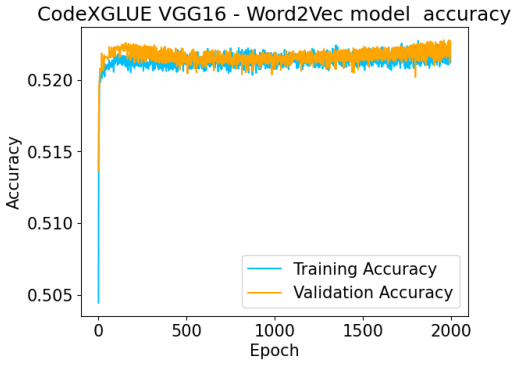


Figure 42: VGG16 Word2Vec earlystop Model Accuracy vs Epoch

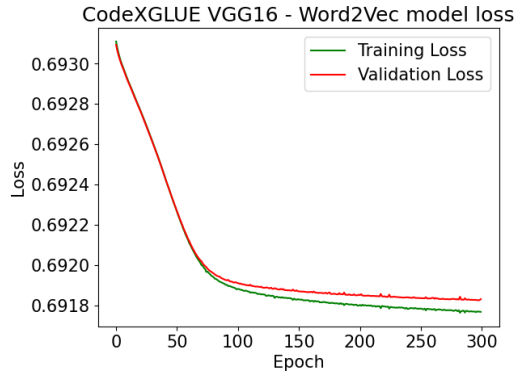


Figure 43: VGG16 Word2Vec Model Loss vs Epoch

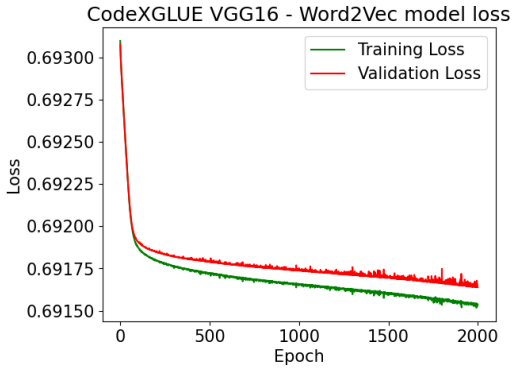


Figure 44: VGG16 Word2Vec earlystop Model Loss vs Epoch

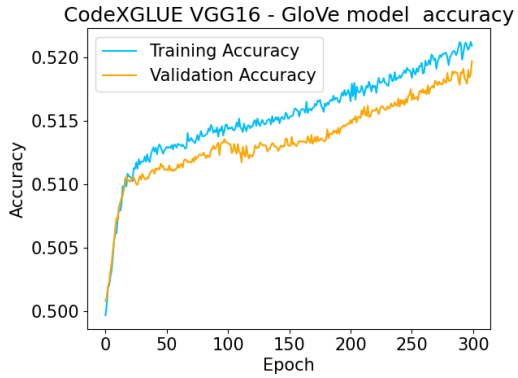


Figure 45: VGG16 GloVe Model Accuracy vs Epoch

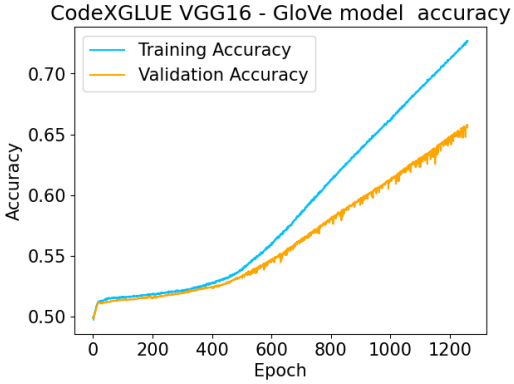


Figure 46: VGG16 GloVe earlystop Model Accuracy vs Epoch

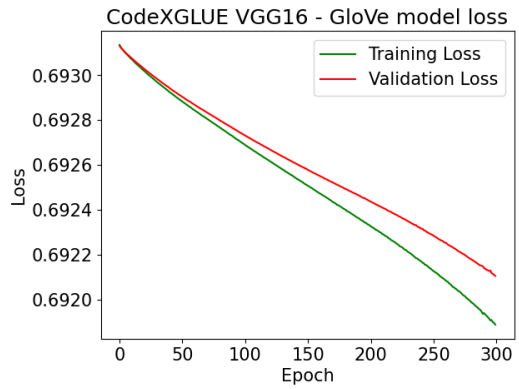


Figure 47: VGG16 GloVe Model Loss vs Epoch

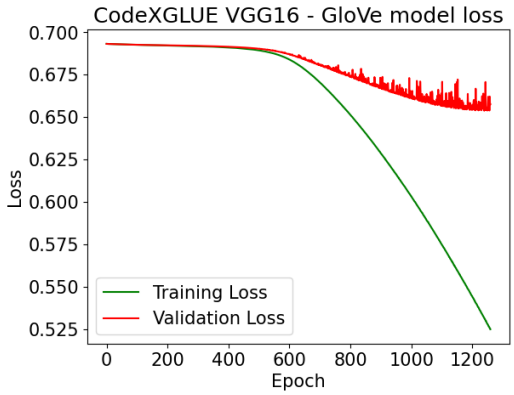


Figure 48: VGG16 GloVe earlystop Model Loss vs Epoch

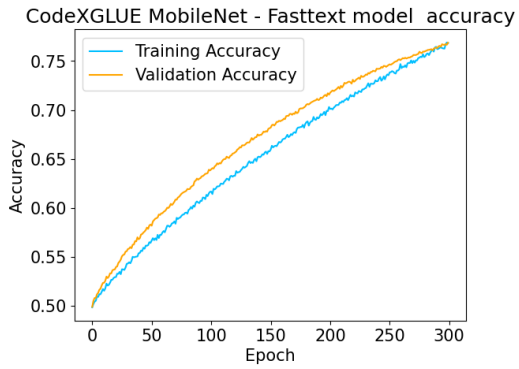


Figure 49: MobileNet Fasttext Model Accuracy vs Epoch

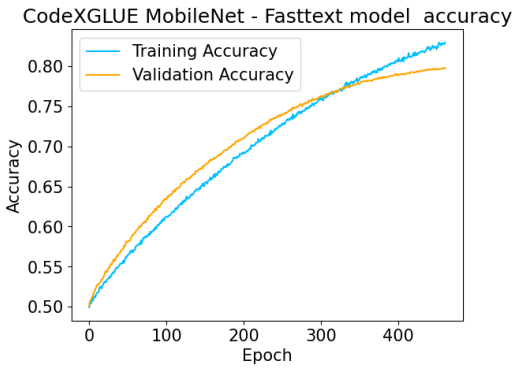


Figure 50: MobileNet Fasttext earlystop Model Accuracy vs Epoch

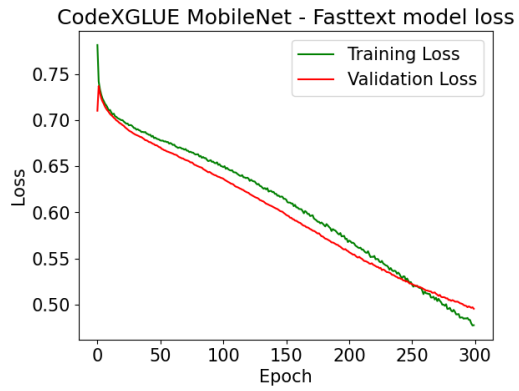


Figure 51: MobileNet Fasttext Model Loss vs Epoch

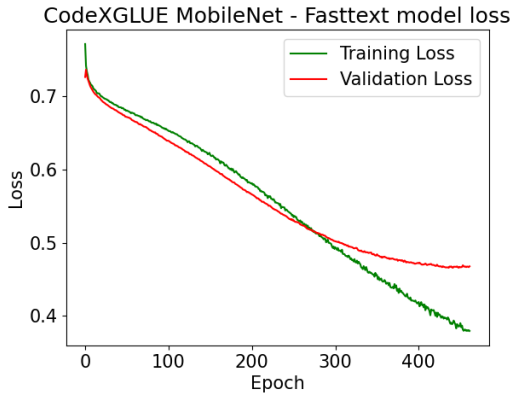


Figure 52: MobileNet Fasttext earlystop Model Loss vs Epoch

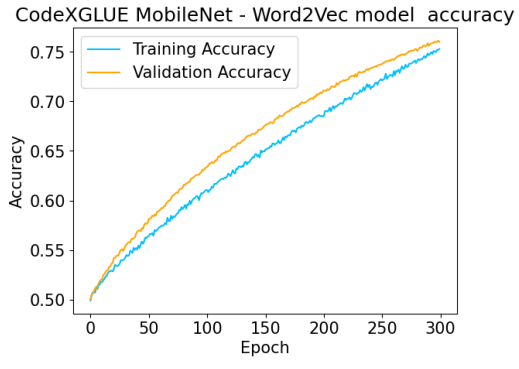


Figure 53: MobileNet Word2Vec Model Accuracy vs Epoch

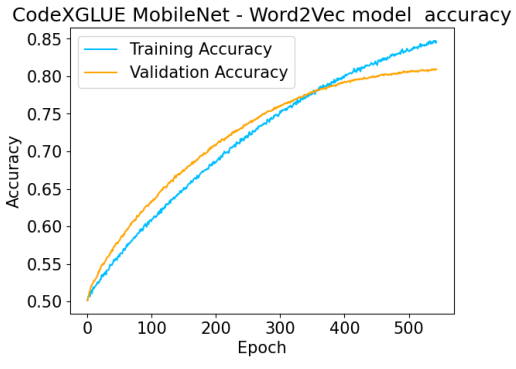


Figure 54: MobileNet Word2Vec earlystop Model Accuracy vs Epoch

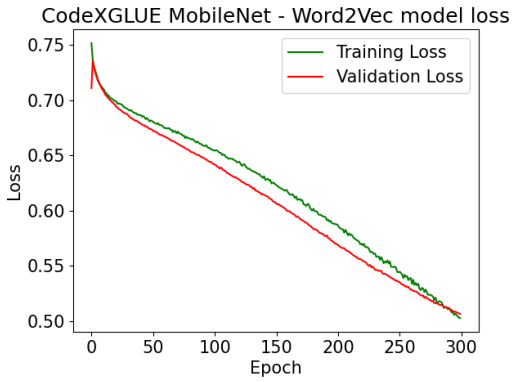


Figure 55: MobileNet Word2Vec Model Loss vs Epoch

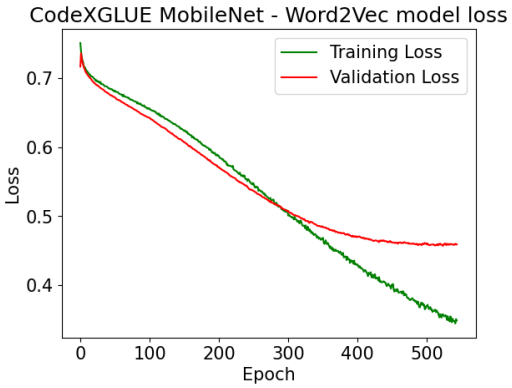


Figure 56: MobileNet Word2Vec earlystop Model Loss vs Epoch

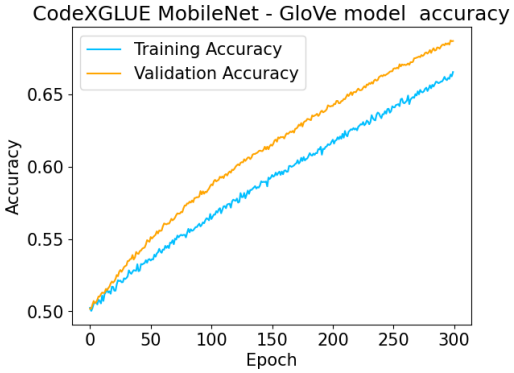


Figure 57: MobileNet GloVe Model Accuracy vs Epoch

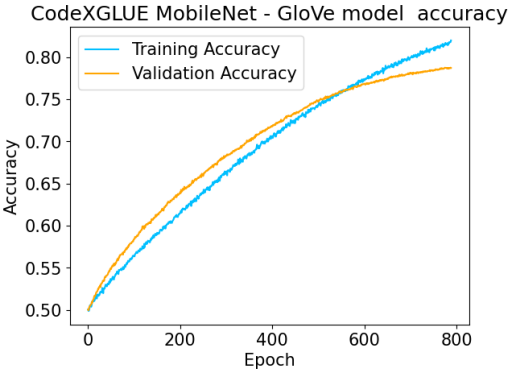


Figure 58: MobileNet GloVe earlystop Model Accuracy vs Epoch

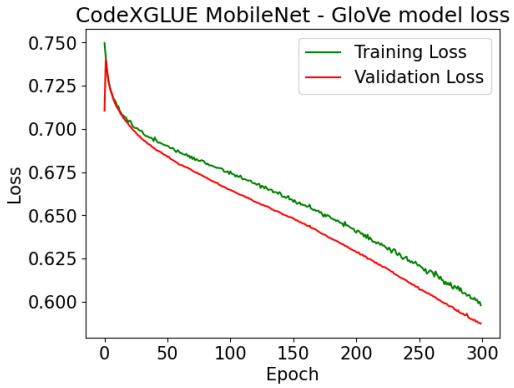


Figure 59: MobileNet GloVe Model Accuracy vs Epoch

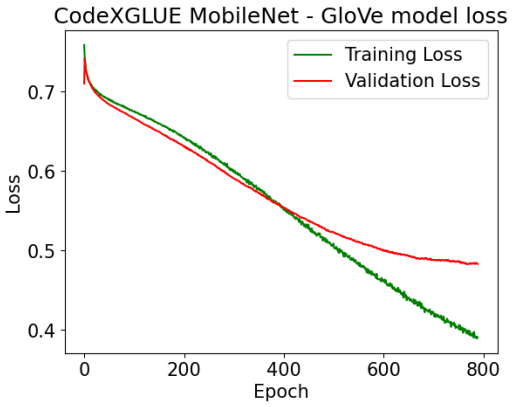


Figure 60: MobileNet GloVe earlystop Model Loss vs Epoch

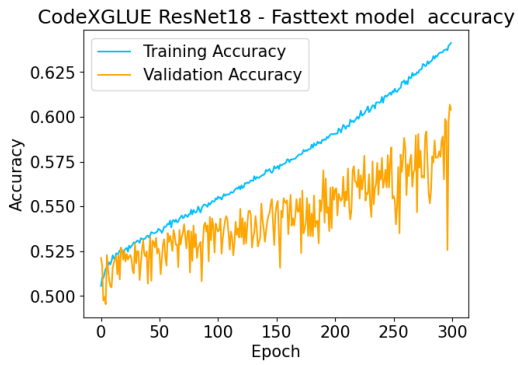


Figure 61: ResNet18 Fasttext Model Accuracy vs Epoch

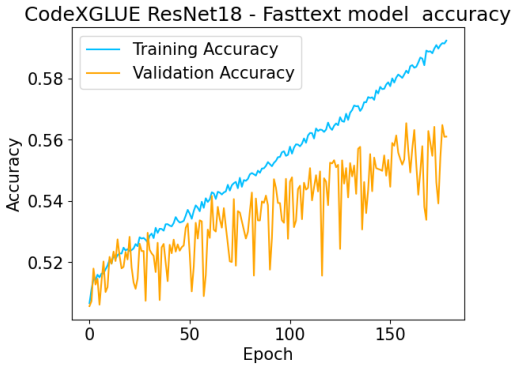


Figure 62: ResNet18 Fasttext earlystop Model Accuracy vs Epoch

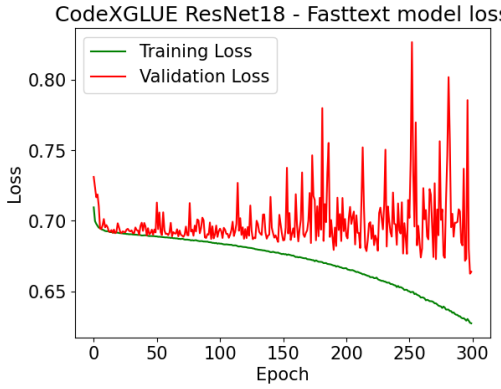


Figure 63: ResNet18 Fasttext Model Loss vs Epoch

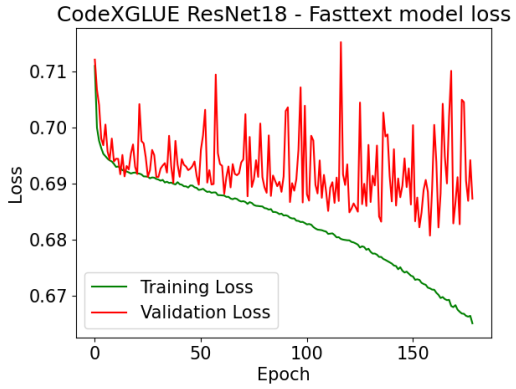


Figure 64: ResNet18 Fasttext earlystop Model Loss vs Epoch

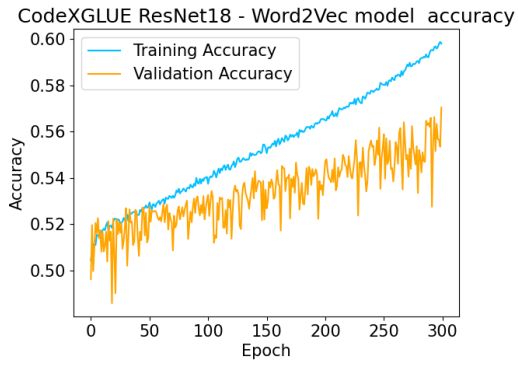


Figure 65: ResNet18 Word2Vec Model Accuracy vs Epoch

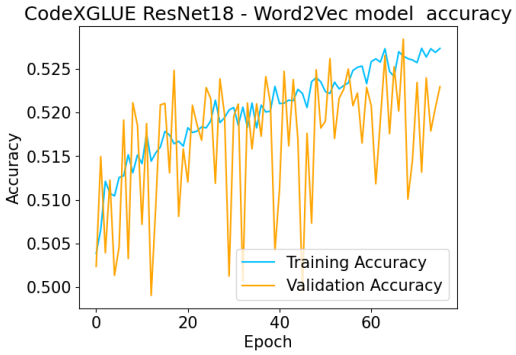


Figure 66: ResNet18 Word2Vec earlystop Model Accuracy vs Epoch

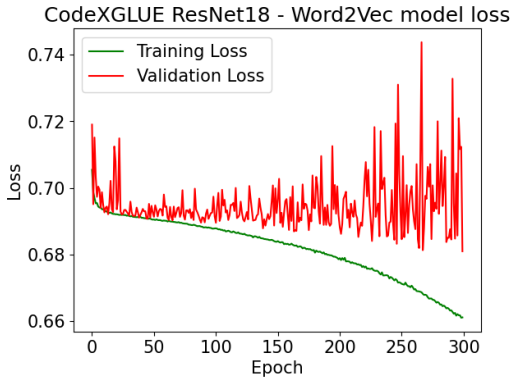


Figure 67: ResNet18 Word2Vec Model Loss vs Epoch

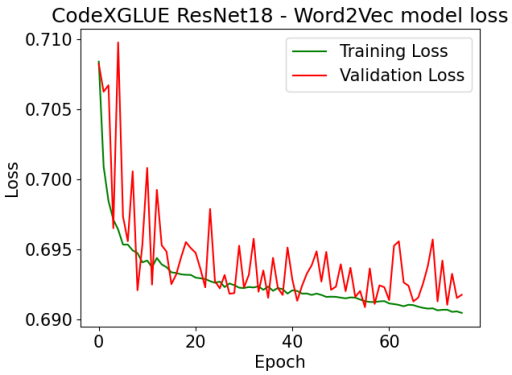


Figure 68: ResNet18 Word2Vec earlystop Model Loss vs Epoch

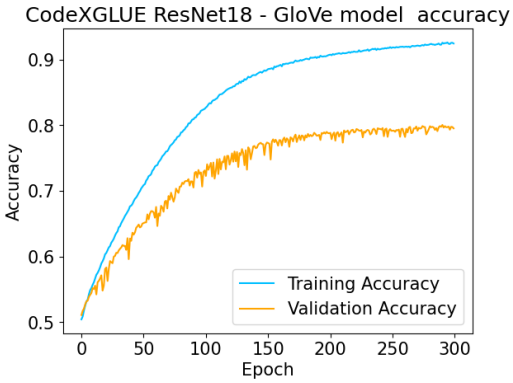


Figure 69: ResNet18 GloVe Model Accuracy vs Epoch

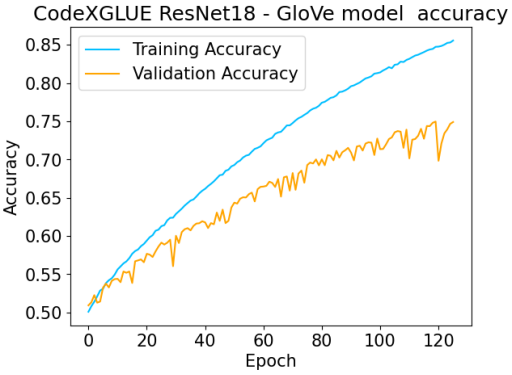


Figure 70: ResNet18 GloVe earlystop Model Accuracy vs Epoch

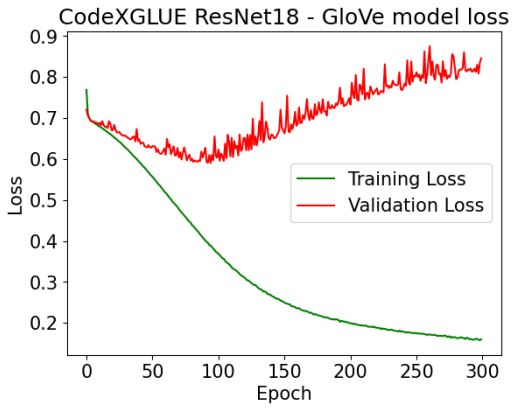


Figure 71: ResNet18 GloVe Model Loss vs Epoch

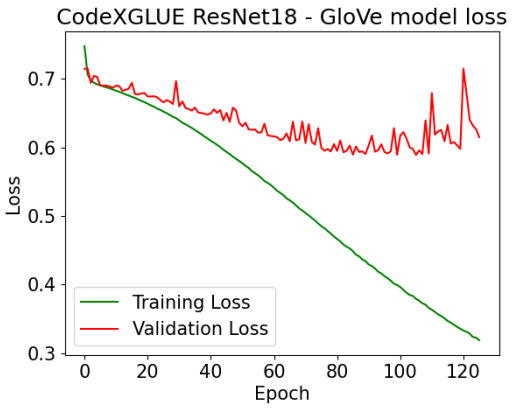


Figure 72: ResNet18 GloVe earlystop Model Loss vs Epoch

Appendix B

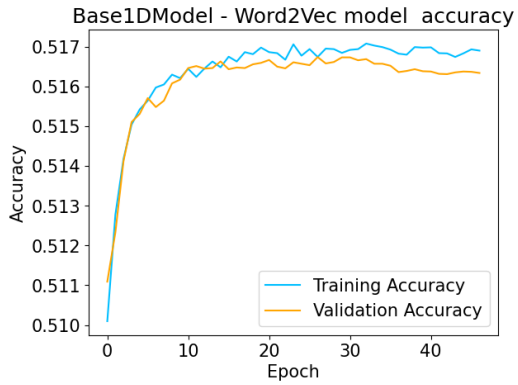


Figure 73: 1DBasemodel Word2Vec Model Accuracy vs Epoch

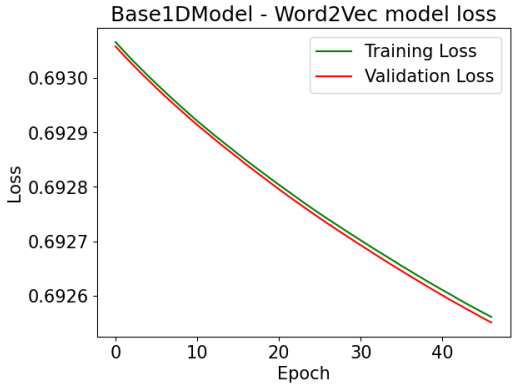


Figure 74: 1DBasemodel Word2Vec Model Loss vs Epoch

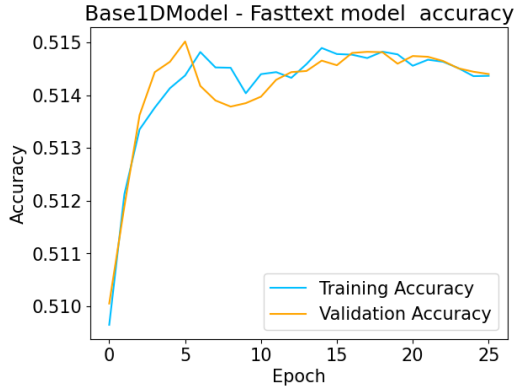


Figure 75: 1DBasemodel Fasttext Model Accuracy vs Epoch

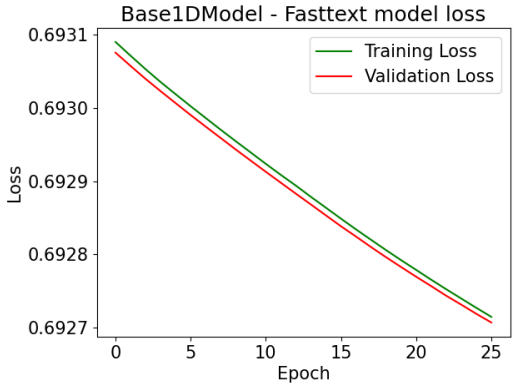


Figure 76: 1DBasemodel Fasttext Model Loss vs Epoch

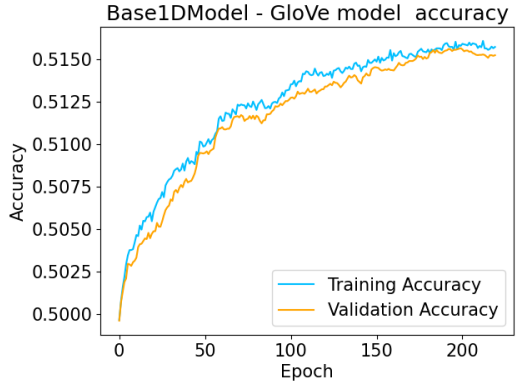


Figure 77: 1DBasemodel GloVe Model Accuracy vs Epoch

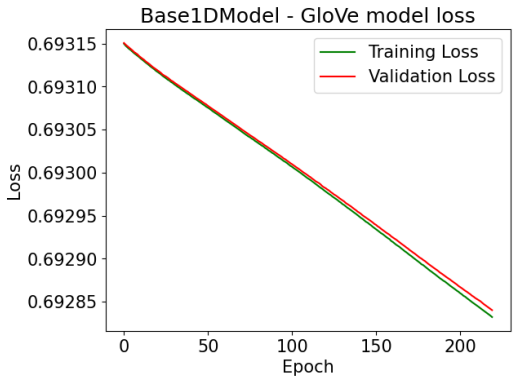


Figure 78: 1DBasemodel GloVe Model Loss vs Epoch

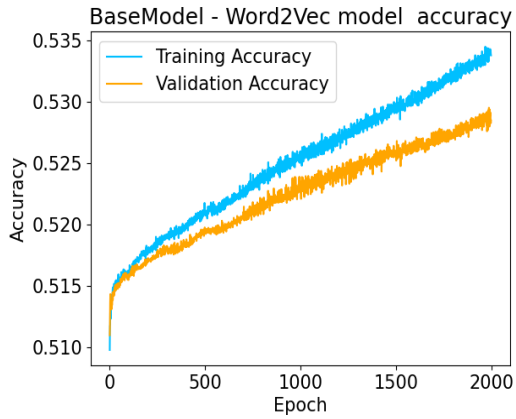


Figure 79: 2D Basemodel Word2Vec Model Accuracy vs Epoch

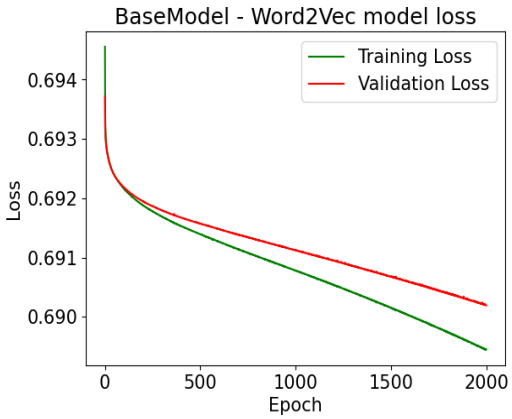


Figure 80: 2D Basemodel Word2Vec Model Loss vs Epoch

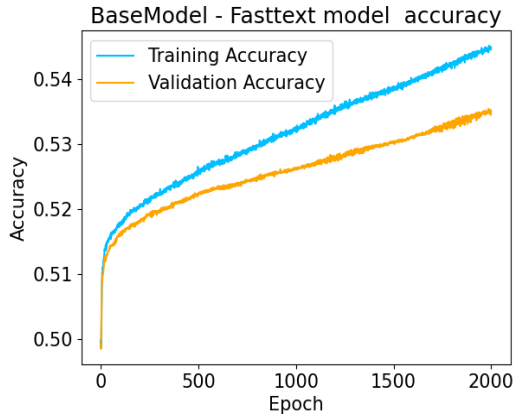


Figure 81: 2D Basemodel Fasttext Model Accuracy vs Epoch

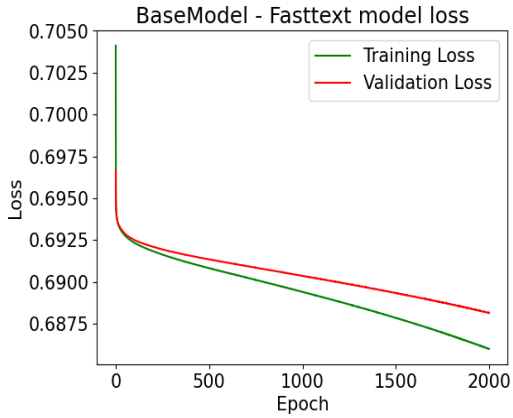


Figure 82: 2D Basemodel Fasttext Model Loss vs Epoch

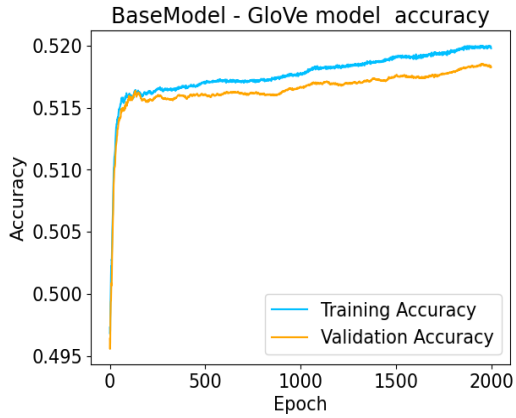


Figure 83: 2D Basemodel GloVe Model Accuracy vs Epoch

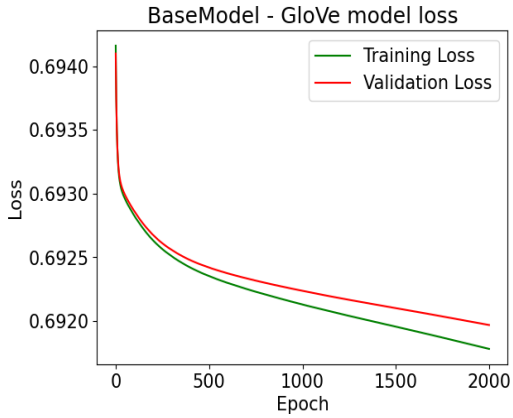


Figure 84: 2D Basemodel GloVe Model Loss vs Epoch

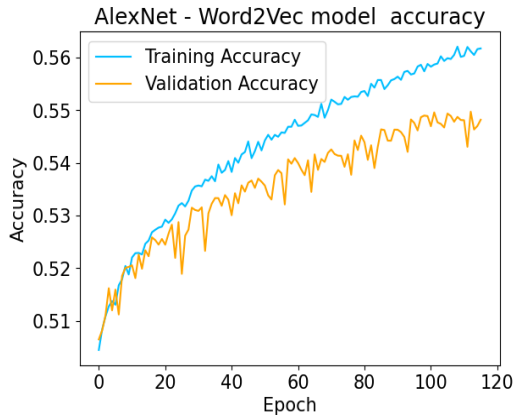


Figure 85: AlexNet Word2Vec Model Accuracy vs Epoch

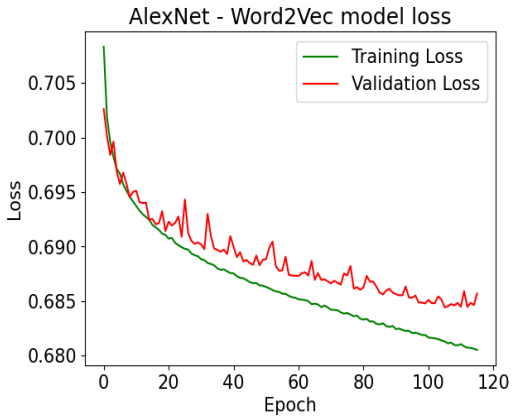


Figure 86: AlexNet Word2Vec Model Loss vs Epoch

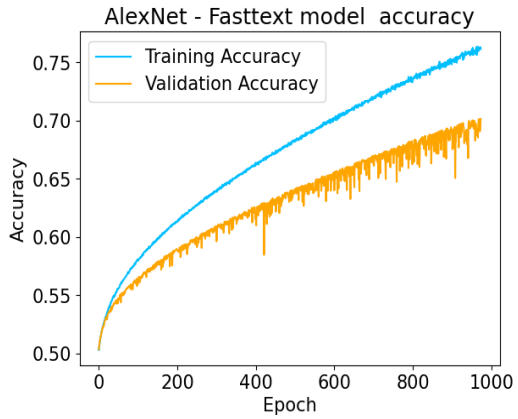


Figure 87: AlexNet Fasttext Model Accuracy vs Epoch

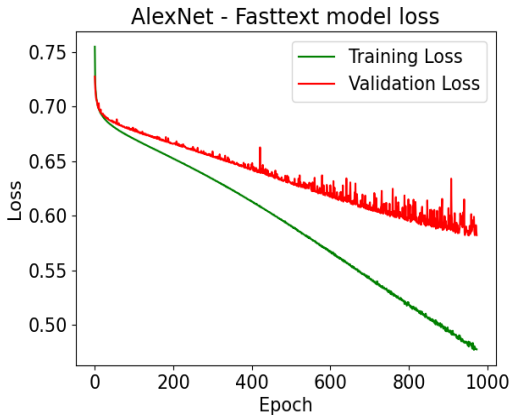


Figure 88: AlexNet Fasttext Model Loss vs Epoch

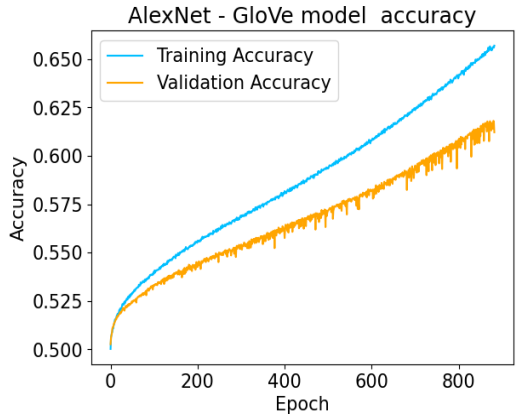


Figure 89: AlexNet GloVe Model Accuracy vs Epoch

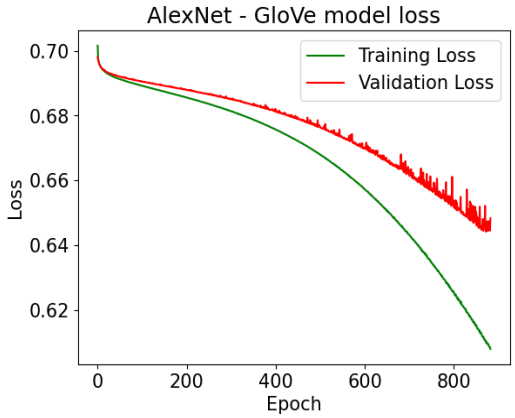


Figure 90: AlexNet GloVe Model Loss vs Epoch

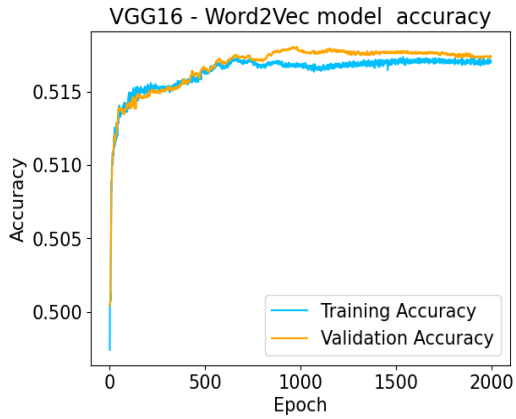


Figure 91: VGG16 Word2Vec Model Accuracy vs Epoch

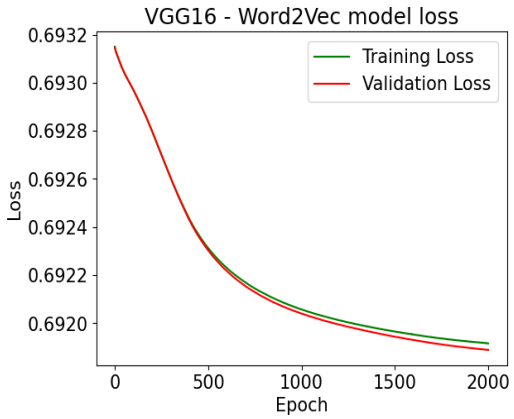


Figure 92: VGG16 Word2Vec Model Loss vs Epoch

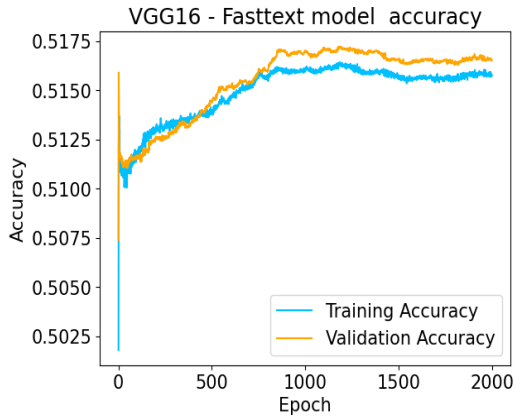


Figure 93: VGG16 Fasttext Model Accuracy vs Epoch

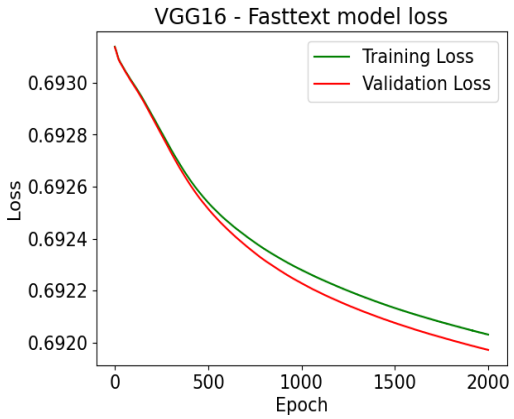


Figure 94: VGG16 Fasttext Model Loss vs Epoch

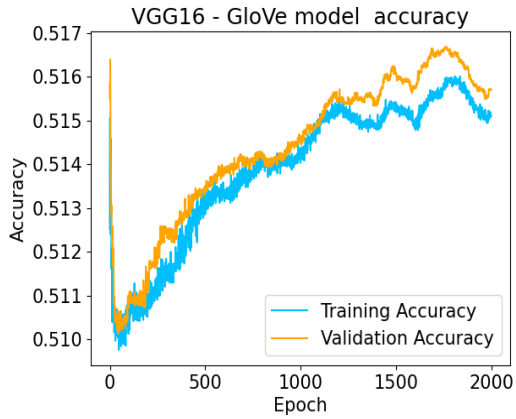


Figure 95: VGG16 Glove Model Accuracy vs Epoch

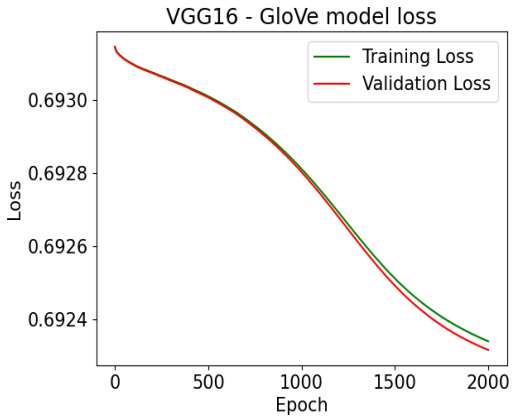


Figure 96: VGG16 GloVe Model Loss vs Epoch

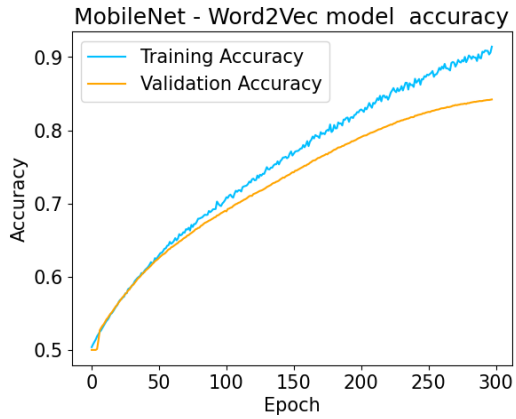


Figure 97: MobileNet Word2Vec Model Accuracy vs Epoch

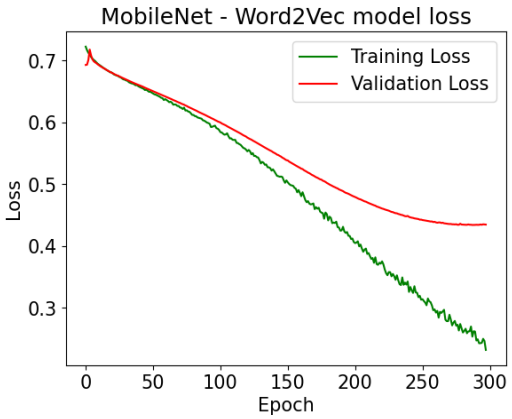


Figure 98: MobileNet Word2Vec Model Loss vs Epoch

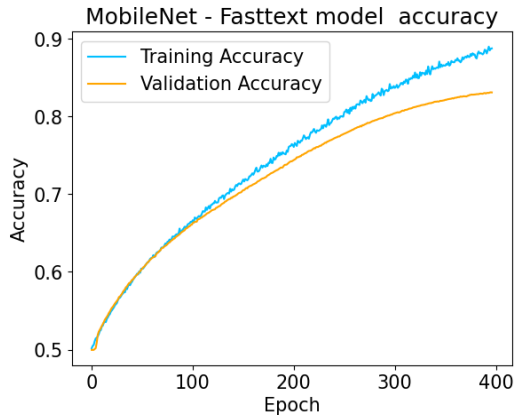


Figure 99: MobileNet Fasttext Model Accuracy vs Epoch

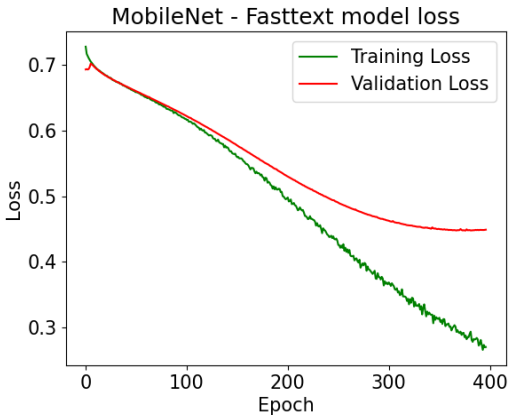


Figure 100: MobileNet Fasttext Model Loss vs Epoch

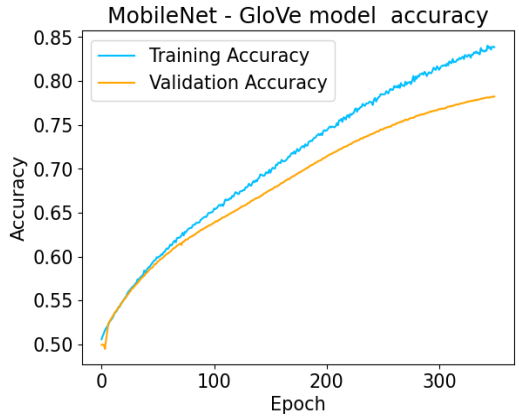


Figure 101: MobileNet GloVe Model Accuracy vs Epoch

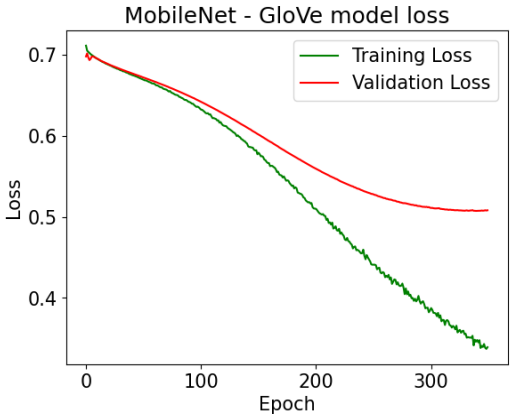


Figure 102: MobileNet GloVe Model Loss vs Epoch

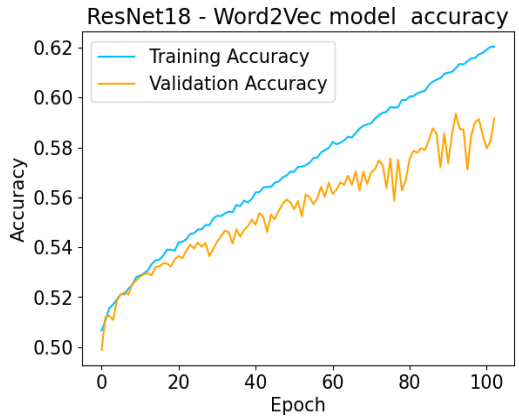


Figure 103: ResNet18 Word2Vec Model Accuracy vs Epoch

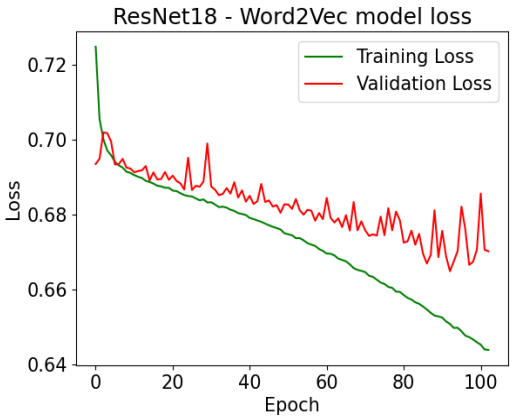


Figure 104: ResNet18 Word2Vec Model Loss vs Epoch

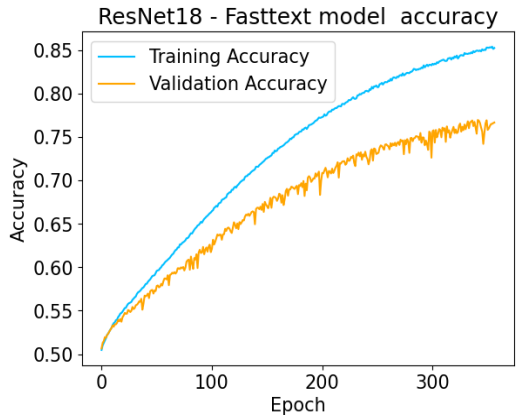


Figure 105: ResNet18 Fasttext Model Accuracy vs Epoch

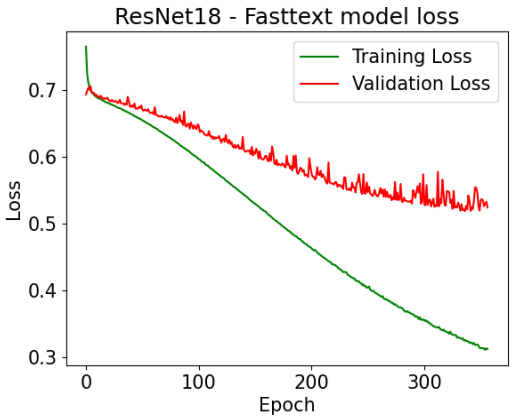


Figure 106: ResNet18 Fasttext Model Loss vs Epoch

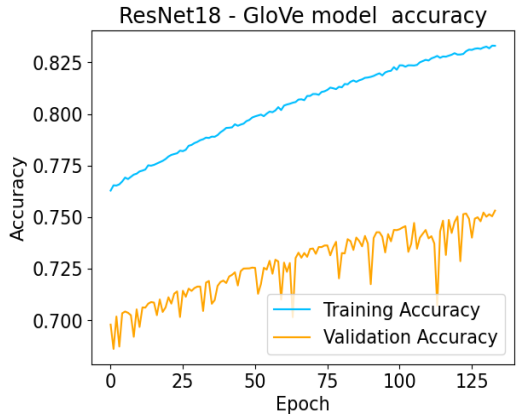


Figure 107: ResNet18 GloVe Model Accuracy vs Epoch



Figure 108: ResNet18 GloVe Model Loss vs Epoch

Appendix C

```
1  @org.junit.Test
2  public void cacheManagerByLocationClasspathTest() throws java.net.
       URISyntaxException {
3      java.net.URI uri1 = new java.net.URI("MY-SCOPE");
4      java.util.Properties properties = new java.util.Properties();
5      properties.setProperty(HazelcastCachingProvider.
           HAZELCAST_CONFIG_LOCATION, "classpath:test-hazelcast-jcache.
           xml");
6      javax.cache.CacheManager cacheManager = javax.cache.Caching.
           getCachingProvider().getCacheManager(uri1, null, properties);
7      junit.framework.Assert.assertNotNull(cacheManager);
8      javax.cache.Cache<java.lang.Integer, java.lang.String> testCache =
            cacheManager.getCache("testCache", java.lang.Integer.class,
           java.lang.String.class);
9      junit.framework.Assert.assertNotNull(testCache);
10 }
```

Listing 1: Example java code snippet from training dataset

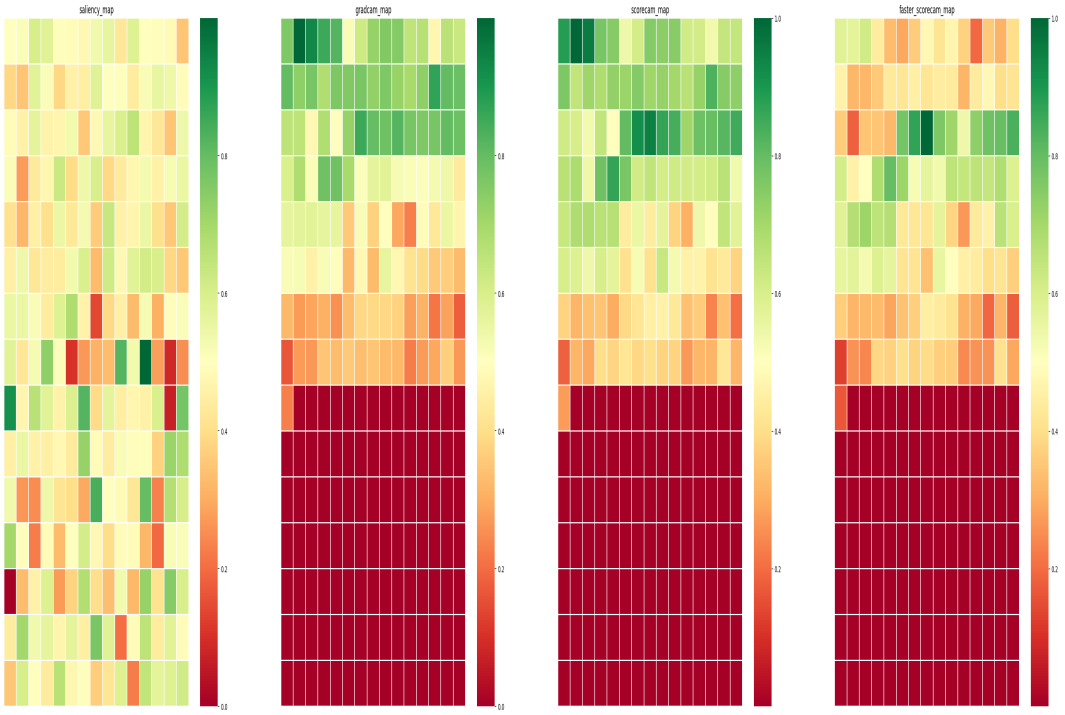


Figure 109: Explainable AI Visualization methods for code snipped 1 generated by AlexNet with Word2Vec embedding,From left to right is Saliency map ,Grad-CAM,Score-CAM, FIMF score-CAM with a heatmap scale of 0 to 1.0 of intensities

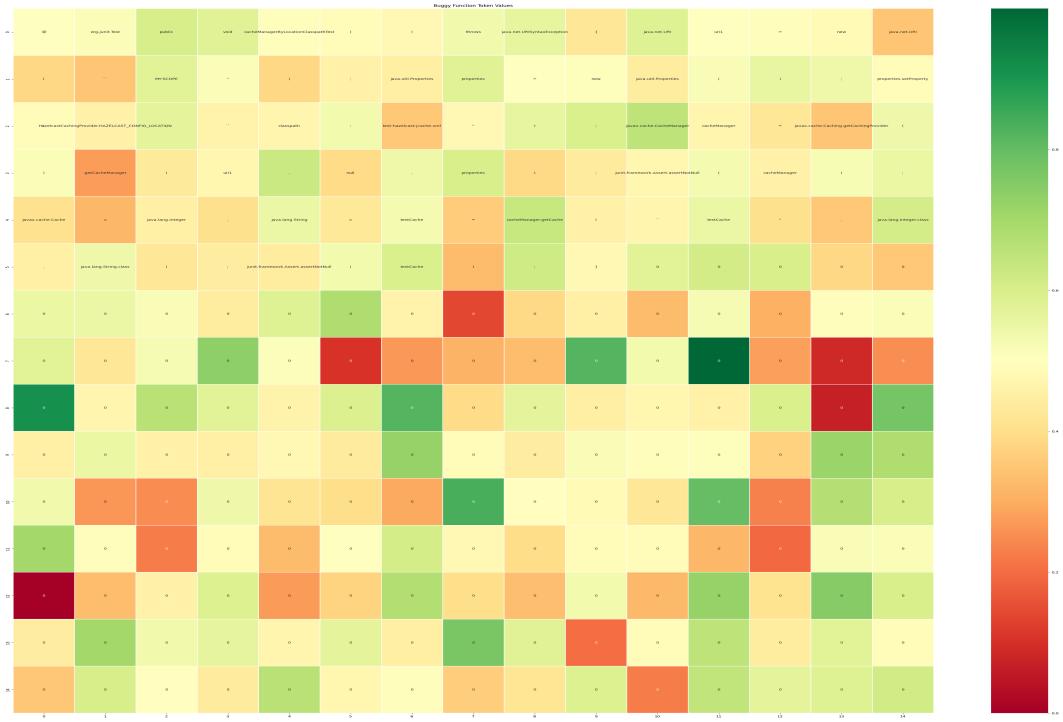


Figure 110: Saliency map representation of a prediction made from a AlexNet with Word2Vec embedding of code snippet in Listing 1

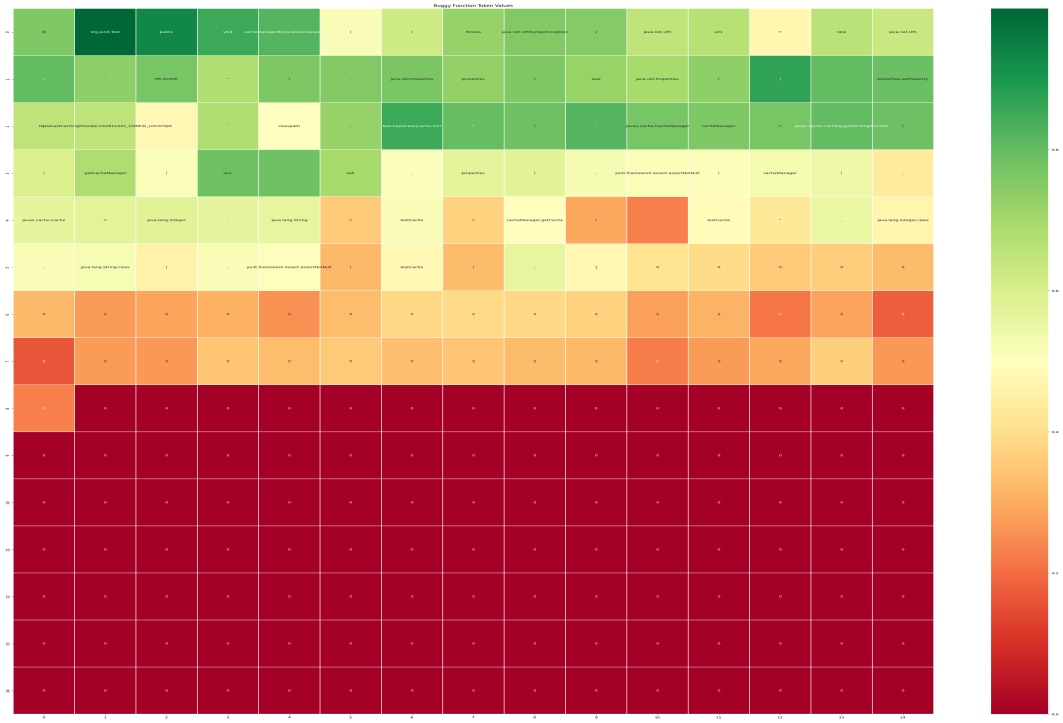


Figure 111: Grad-CAM map representation of a prediction made from a AlexNet with Word2Vec embedding of code snippet in Listing 1

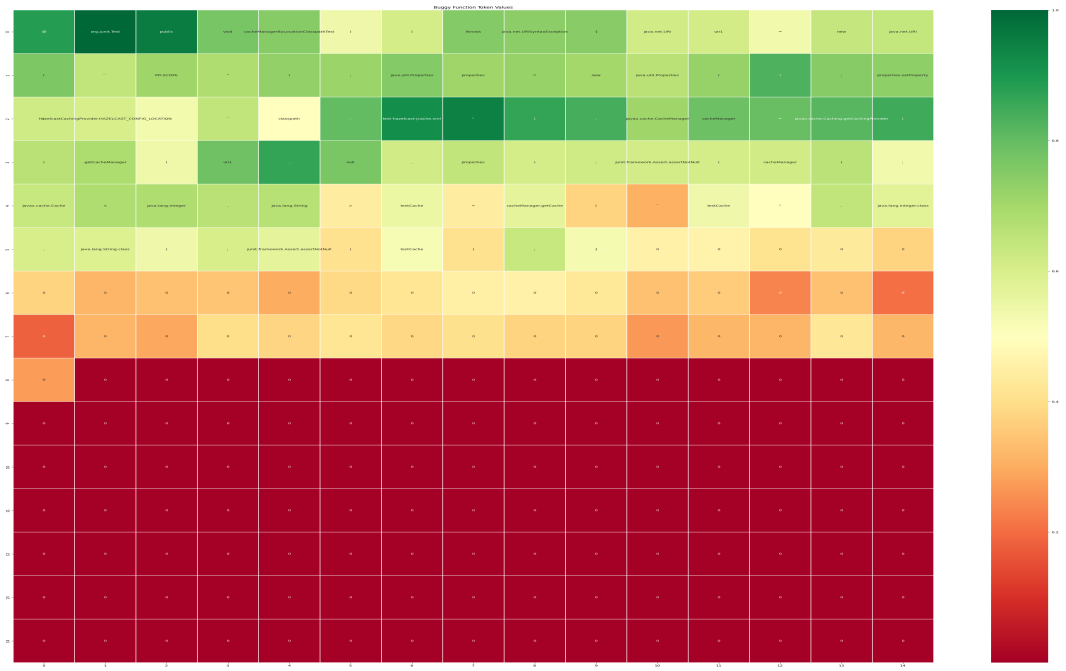


Figure 112: Score-CAM map representation of a prediction made from a AlexNet with Word2Vec embedding of code snippet in Listing 1

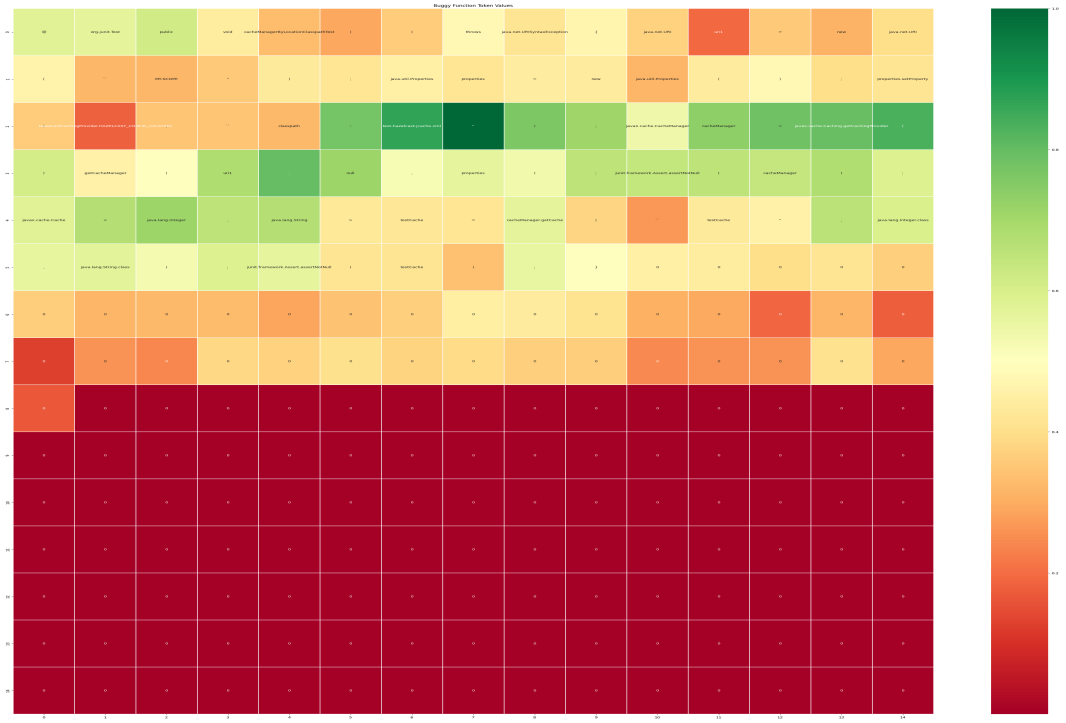


Figure 113: FIMF score-CAM map representation of a prediction made from a AlexNet with Word2Vec embedding of code snippet in Listing 1

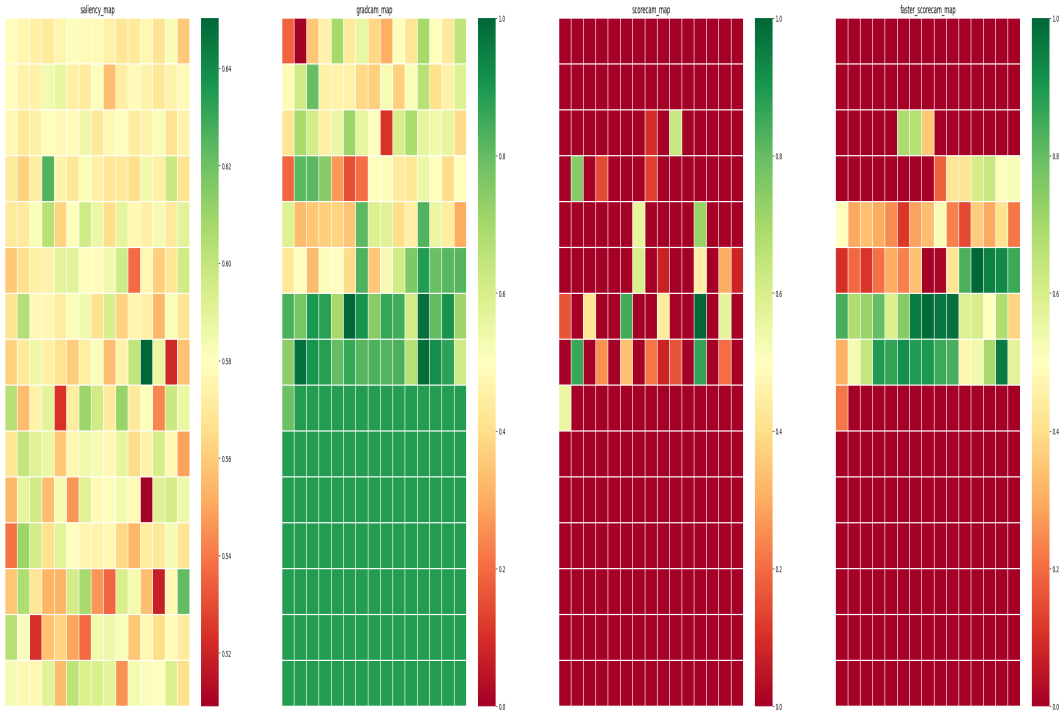


Figure 114: Explainable AI Visualization methods for code snipped 1 generated by ResNet18 with Word2Vec embedding,From left to right is Saliency map ,Grad-CAM,Score-CAM, FIMF score-CAM with a heatmap scale of 0 to 1.0 of intensities

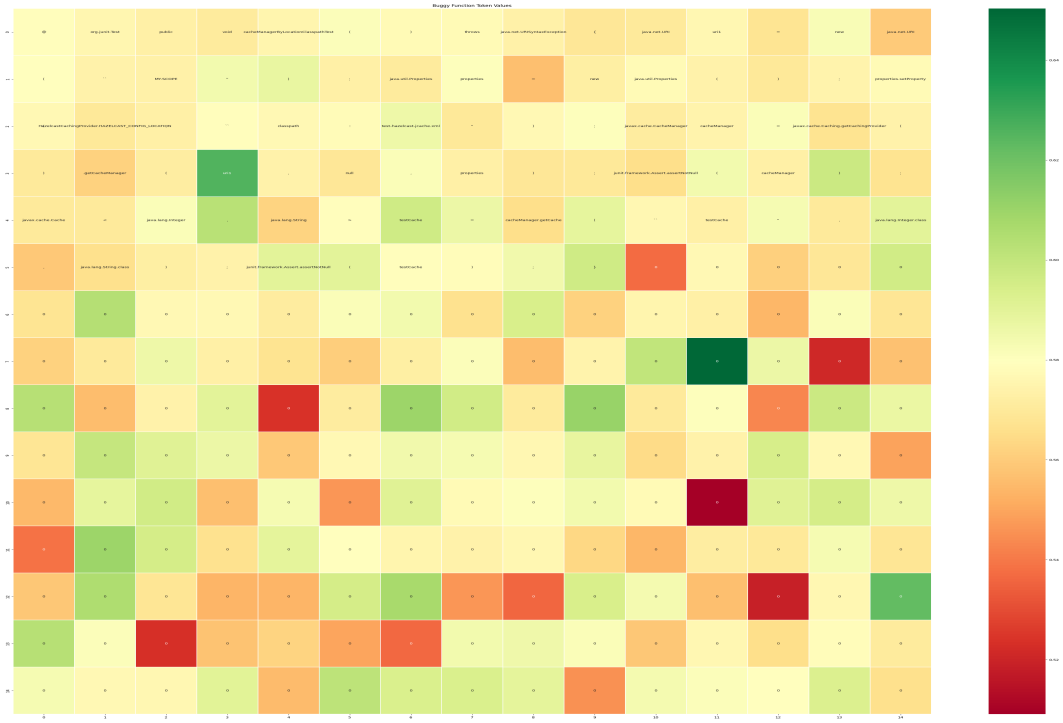


Figure 115: Saliency map representation of a prediction made from a ResNet18 with Word2Vec embedding of code snippet in Listing 1

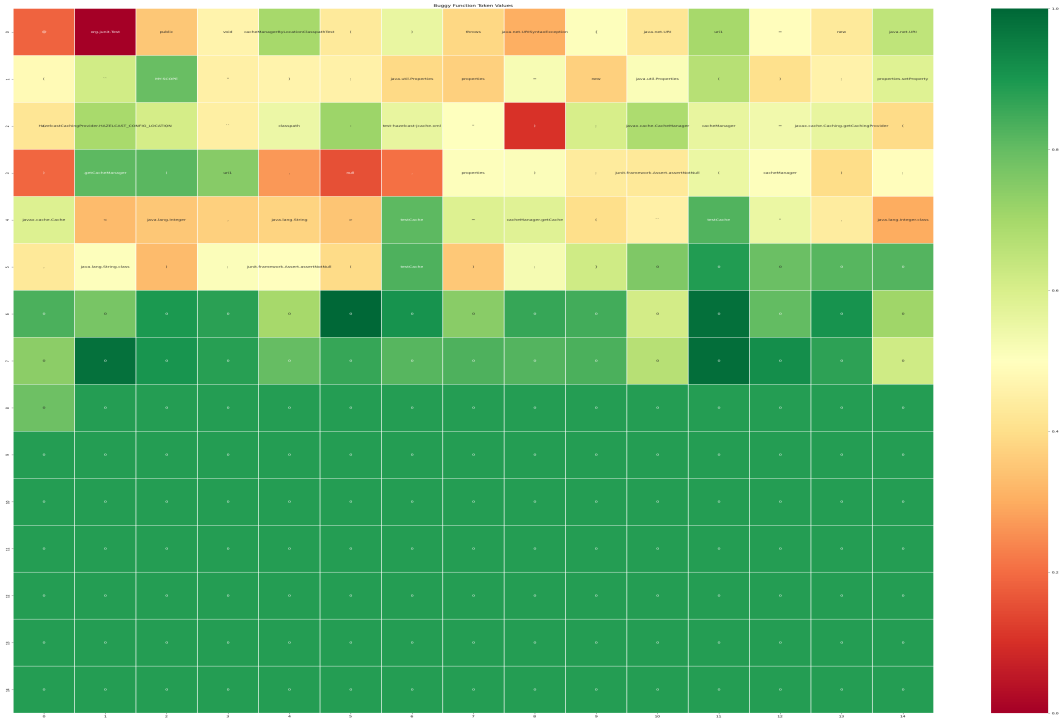


Figure 116: Grad-CAM map representation of a prediction made from a ResNet18 with Word2Vec embedding of code snippet in Listing 1

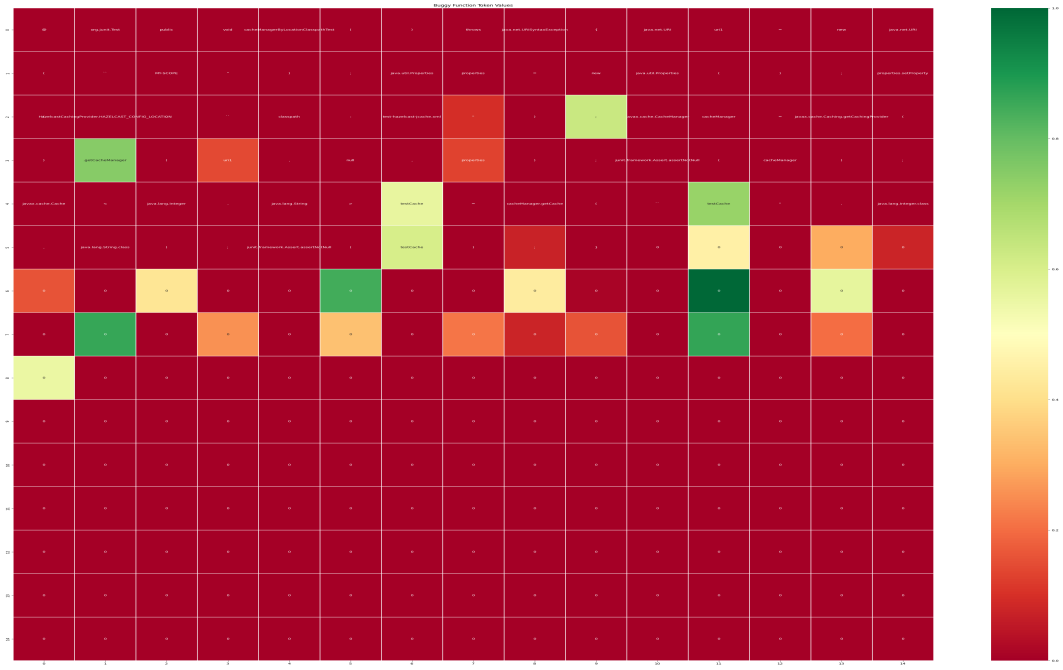


Figure 117: Score-CAM map representation of a prediction made from a ResNet18 with Word2Vec embedding of code snippet in Listing 1

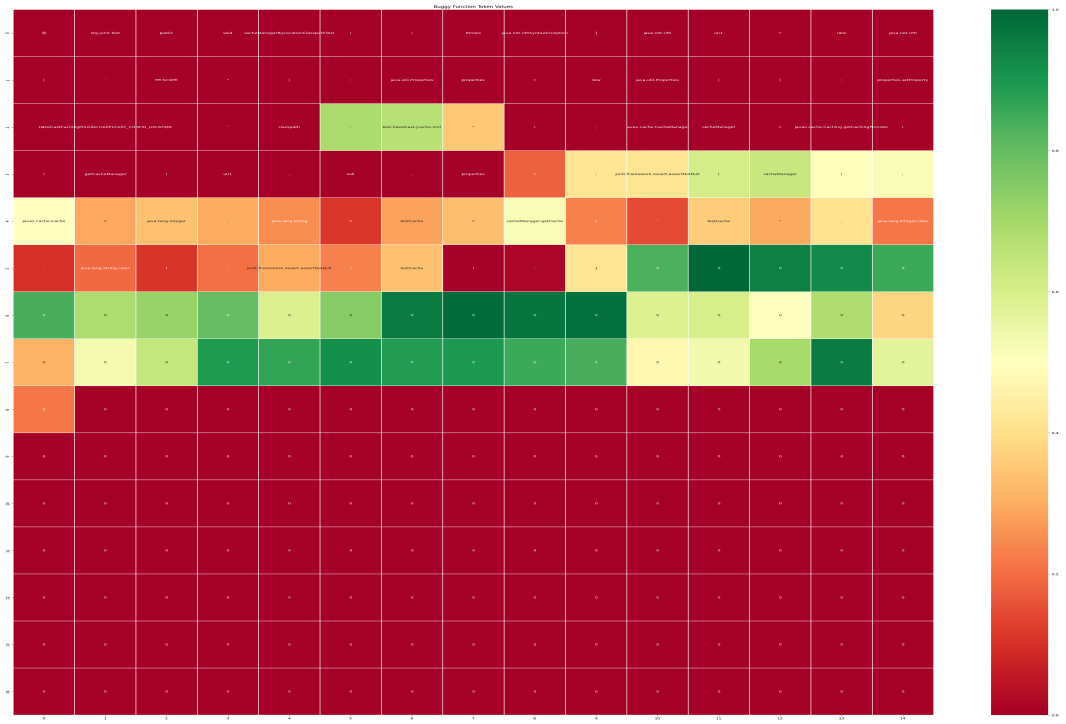


Figure 118: FIMF score-CAM map representation of a prediction made from a ResNet18 with Word2Vec embedding of code snippet in Listing 1