

**A Framework for Software Code and Model Smell Detection
through Self-Supervised Learning**

BY

Amal Abdulrahman Ali Alazba

A Dissertation Presented to the
FACULTY OF THE COLLEGE OF GRADUATE STUDIES
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

In

Computer Science

December 2023

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN- 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **Amal Abdulrahman Ali Alazba** under the direction of his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE**.



Dr. Motaz Alfarraj
Department Chairman



Prof. Suliman Al-Homidan
Dean of Graduate Studies

Date



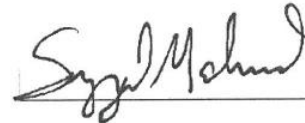
Prof. Mohammad Alshayeb
(Advisor)



Dr. Hamoud Aljamaan
(Co-Advisor)



Prof. Mahmood Niazi
(Member)



Dr. Sajjad Mahmood
(Member)



Dr. Jameleddine Hassine
(Member)



© Amal Abdulrahman Ali Alazba

2023

Dedication

To my Father, Prof. Abdulrahman Alazba: Your consultancy, ultimate support, wisdom, and encouragement have been a constant source of inspiration.

To my Mother, Monera Almoosa: Your endless love, care, and support have sustained me on this path. I am deeply grateful for our weekly journeys to attend my classes, which meant the world to me.

To my Brothers, Dr. Moath and Ibrahim: Your presence and belief in my abilities have been a driving force.

To my Sisters, Arwa, my mentor: Your presence has added joy and purpose to my journey. Your support and encouragement have been instrumental. Asma, my twin: Your thoughtful presents, has not only motivated me but also helped me in countless ways. Maryam, my friend: Your silent support has spoken volumes. Your presence has brought positivity and strength to my journey.

To my Nieces, Lujain, Elaf, Lama, and Rafeef: Your youthful energy and playful spirit have reminded me to take breaks and find joy in the simple moments. Your presence has fueled my determination to complete this dissertation.

To my Best Friend, Asma Alawadh: You are more than a friend; you are a sister and a kindred spirit. Your friendship has been a pillar of strength.

ACKNOWLEDGMENTS

First and foremost, Alhamdulillah, I want to express my deepest gratitude to Allah. This achievement would not have been possible without Allah's grace and guidance.

I am deeply indebted to my Advisors, Prof. Mohammad Alshayeb and Dr. Hamoud Aljamaan, whose unwavering guidance, expert insights, and continuous encouragement have been invaluable throughout the research process. Your dedication to my academic growth and the countless hours you invested in reviewing, critiquing, and shaping this work is deeply appreciated.

I want to express my sincere thanks to the members of my Dissertation Committee: Prof. Mahmood Niazi, Dr. Sajjad Mahmood, and Dr. Jameleddine Hassine. Your expertise, valuable feedback, and critical evaluations have contributed significantly to the refinement of this work. I am truly grateful for your commitment to my academic development.

I sincerely thank my family, especially my parents, Prof. Abdulrahman Alazba and Mrs. Monera Almoosa, for their unconditional love, unwavering support, and sacrifices. Your belief in my abilities and willingness to travel this academic journey with me has been a constant source of motivation.

I am also indebted to King Saud University (KSU) for allowing me to pursue my doctoral studies at King Fahd University of Petroleum & Minerals (KFUPM). I am grateful to the Department of Information Systems for granting me the time needed to complete my PhD journey. I sincerely thank KFUPM for accepting me as a PhD student. The resources, facilities, and academic environment at KFUPM have been conducive to my research endeavors.

My thanks also go to my friends and colleagues, both near and far, who have been a source of inspiration, camaraderie, and intellectual exchange. Your insights and discussions have enriched my research and personal growth, especially my batch, the first batch of female graduates at KFUPM. To all those who have contributed in ways seen and unseen, your support has been indispensable, and this dissertation would not have been possible without you. |

|

TABLE OF CONTENTS

ACKNOWLEDGMENTS	V
TABLE OF CONTENTS	VII
LIST OF TABLES	XIII
LIST OF FIGURES	XV
LIST OF ABBREVIATIONS	XVII
ABSTRACT	XXVII
ملخص الرسالة	XXIX
CHAPTER 1 INTRODUCTION	1
1.1 Problem Statement and Motivation	2
1.2 Research Objectives	6
1.3 Research Methodology	9
1.4 Dissertation Organization	12
CHAPTER 2 BACKGROUND	14
2.1 Bad Smells	14
2.1.1 Code level smells	14
2.1.2 Design level smells	16
2.1.3 Categories and classification	18
2.2 Machine learning techniques	22
2.2.1 Supervised learning	22
2.2.2 Unsupervised learning	25
2.3 Deep learning	26
2.3.1 Concepts and terminologies	27
2.3.2 Architectures and Algorithms	29

2.4	Bots.....	32
------------	------------------	-----------

CHAPTER 3 DEEP LEARNING APPROACHES FOR BAD SMELL DETECTION: A SYSTEMATIC LITERATURE REVIEW 33

3.1	Related Work	35
------------	---------------------------	-----------

3.2	Methodology	39
------------	--------------------------	-----------

3.2.1	Research Questions	39
3.2.2	Study Selection	41
3.2.3	Search Strategy Design	41
3.2.4	Quality Assessment	45
3.2.5	Data Extraction	46
3.2.6	Data Synthesis	46

3.3	Results	47
------------	----------------------	-----------

3.3.1	What is the current status of deep learning-based bad smell detection?	47
3.3.2	Which bad smells have been detected using deep learning?	50
3.3.3	What is the purpose of using deep learning for bad smell detection?	58
3.3.4	Which input data have been used in deep learning-based bad smell detection?.....	66
3.3.5	Which deep learning models are used for bad smell detection?	76
3.3.6	How are deep learning models evaluated and implemented in bad smell detection?	86
3.3.7	How are the bad Smell datasets being constructed and used?	95

3.4	Performance Analysis.....	102
------------	----------------------------------	------------

3.4.1	Dataset	103
3.4.2	Bad smells.....	105
3.4.3	Deep learning models.....	107
3.4.4	Encoding techniques.....	109
3.4.5	Preprocessing techniques.....	111
3.4.6	Features	113

3.5	Discussion and Implications	115
------------	--	------------

3.5.1	Exploring other bad smells (RQ2)	115
3.5.2	Model-level detection (RQ2)	116
3.5.3	Exploring new learning approaches (RQ3)	116
3.5.4	From binary classification to multilabel (RQ3)	117
3.5.5	Exploitation of hyperparameter optimization (RQ4).....	118
3.5.6	Data preprocessing (RQ5).....	118
3.5.7	More attention on model evaluation (RQ6)	119
3.5.8	Automated tools (RQ6).....	120
3.5.9	Large-scale and publicly available datasets (RQ7).....	120
3.5.10	Deep learning model performance (RQ8)	121

3.6	Threats to Validity.....	121
------------	---------------------------------	------------

3.6.1	Construct validity.....	122
-------	-------------------------	-----

3.6.2	Internal validity	122
3.6.3	External validity	123
3.6.4	Conclusion validity	124
3.7	Chapter Summary.....	124

CHAPTER 4 CoRT: TRANSFORMER-BASED CODE REPRESENTATIONS WITH SELF-SUPERVISION BY PREDICTING RESERVED WORDS FOR CODE SMELL DETECTION 127

4.1	Related Work	130
4.1.1	ML and DL studies	130
4.1.2	Pre-trained Models for Programming Languages	133
4.2	Datasets Construction	135
4.2.1	Collection of unlabeled datasets	136
4.2.2	Collection of labeled datasets	137
4.3	Approach.....	140
4.3.1	Overview	140
4.3.2	Data extraction.....	143
4.3.3	Data preprocessing.....	143
4.3.4	Pre-training.....	144
4.3.5	Fine-tuning	154
4.4	Experiments and evaluation	157
4.4.1	Research questions	158
4.4.2	Evaluation metrics.....	160
4.4.3	Validation method.....	162
4.5	Results and Discussion.....	162
4.5.1	RQ1 How does the models' architecture influence the performance of CoRT?	163
4.5.2	RQ2 How does hyperparameter optimization affect the performance of CoRT?	165
4.5.3	RQ3 What is the detection performance of CoRT for identifying class-level and method-level code smells?	170
4.5.4	RQ4 How efficient is the proposed approach?	179
4.5.5	RQ5 To what extent does CoRT enhance the detection performance compared to the feature-based approach?	181
4.5.6	RQ6 How well does CoRT generalize on unseen projects?	184
4.6	Implications and Limitations.....	190
4.6.1	Proxy Tasks	190
4.6.2	Detection Performance	190
4.6.3	Computational Efficiency	191
4.6.4	Comparative evaluation	192
4.6.5	Advantages for researchers and developers	192

4.7	Threats to Validity	193
4.7.1	Conclusion validity	193
4.7.2	Construct validity	193
4.7.3	Internal validity	195
4.7.4	External validity	196
4.8	Chapter Summary	196
CHAPTER 5 AUTOMATED DETECTION OF CLASS DIAGRAM SMELLS USING SELF-SUPERVISED LEARNING		199
5.1	Motivating Example	200
5.2	Related Work	203
5.3	Dataset Preparation	207
5.3.1	Unlabeled data collection	207
5.3.2	Labeled datasets construction	208
5.4	MoRT	212
5.4.1	Pre-training MoRT	212
5.4.2	Fine-tuning MoRT	218
5.4.3	Comparison models	220
5.5	Experimentation	221
5.5.1	Research questions	222
5.5.2	Evaluation metrics	224
5.5.3	Validation technique	224
5.6	Experimental Results	225
5.6.1	The effect of model architectures and hyperparameters	225
5.6.2	The detection performance of MoRT for detecting class diagram smells	227
5.6.3	Comparison with state-of-the-art approaches	233
5.7	Discussion	237
5.7.1	Pre-training tasks	237
5.7.2	Scalability and Adaptability	239
5.7.3	Advantages for researchers	240
5.7.4	Usage in industrial settings	240
5.8	Threats to validity	240
5.8.1	Conclusion validity	241
5.8.2	Construct validity	241
5.8.3	Internal validity	242
5.8.4	External validity	242
5.9	Chapter Summary	242

CHAPTER 6 SMELLYBOT: DESIGN AND IMPLEMENTATION OF AI-POWERED SOFTWARE BOT FOR CODE SMELL DETECTION.....	244
6.1 Related Work	246
6.2 Methodology.....	250
6.2.1 Overview	251
6.2.2 SmellyBot Building Blocks	253
6.2.3 Design Decisions.....	255
6.2.4 Implementation.....	256
6.3 Evaluation	257
6.3.1 Research Questions.....	257
6.3.2 Methods	258
6.3.3 Study Subjects	259
6.4 Results and Discussion.....	261
6.4.1 Developers' perception, recommendations, and expectations of using SmellyBot	262
6.4.2 The efficiency and accuracy of SmellyBot in code smell detection.....	267
6.5 Implications.....	271
6.5.1 Implications for researchers.....	271
6.5.2 Implications for practitioners.....	272
6.6 Threats to Validity	273
6.6.1 Construct validity	273
6.6.2 Internal validity	273
6.6.3 External validity.....	274
6.6.4 Conclusion validity	274
6.7 Chapter Summary.....	275
CHAPTER 7 CONCLUSION AND FUTURE DIRECTIONS	276
7.1 Contributions	278
7.2 Challenges and Limitations.....	279
7.3 Future Directions.....	280
APPENDIX A: THE SELECTED PRIMARY STUDIES	282
APPENDIX B: TYPES OF BAD SMELLS DETECTED USING DL.....	286
APPENDIX C: PUBLICLY AVAILABLE DATASETS FOR EACH BAD SMELL.....	289

APPENDIX D: DETAILED PROJECTS' DESCRIPTION	290
APPENDIX E: CROSS-PROJECT PERFORMANCE HEATMAP	297
REFERENCES.....	301
VITAE	320

LIST OF TABLES

Table 1 Targeted research questions.....	39
Table 2 Quality assessment checklist	45
Table 3 Data extraction form	46
Table 4 Granularity of dependent variables.....	57
Table 5 Main purpose of using DL in the PSs.....	60
Table 6 Learning approaches explored in the PSs.....	63
Table 7 DL models used by learning approaches	65
Table 8 Features used in DL-based bad smell detection.....	68
Table 9 Data preprocessing and techniques used	74
Table 10 Encoding techniques used in the PSs.....	75
Table 11 DL models used for predictive classification	77
Table 12 DL models used for representation learning.....	79
Table 13 Methods used in each training hyperparameter	84
Table 14 Validation methods used by the PSs.....	89
Table 15 Evaluation metrics used by the PSs	90
Table 16 Baseline techniques used in the PSs	93
Table 17 Source code availability	95
Table 18 Data used by the PSs.....	97
Table 19 Programming languages of the datasets used	98
Table 20 Labeling methods used to create the oracles	100
Table 21 Software tools used for feature extraction	101
Table 22 Dataset availability	101
Table 23 Summary of previous ML- and DL-based approaches.....	132
Table 24 Pre-training dataset description	136
Table 25 Fine-tuning datasets description	139
Table 26 Create pseudo labels	146
Table 27 Hyperparameter optimization search space	148
Table 28 Pre-trained model's best hyperparameters	170
Table 29 Data Class code smell detection performance	173
Table 30 God Class code smell detection performance.....	174
Table 31 Feature Envy code smell detection performance	175
Table 32 Long Method code smell detection performance	176
Table 33 Comparison with previous studies.....	177
Table 34 Comparison with state-of-the-art pre-trained models.....	178
Table 35 Pre-training time	180
Table 36 Fine-tuning time.....	180
Table 37 Summary of previous approaches for design smell detection	206
Table 38 Fine-tuning datasets description	210

Table 39 Pre-trained models performance	225
Table 40 Pre-trained model’s best hyperparameters (MoRT, ANN, CNN, and LSTM)	226
Table 41 Design smell detection performance.....	229
Table 42 Comparison with previous work.....	232
Table 43 Pre-training time (MoRT, ANN, CNN, and LSTM)	232
Table 44 Fine-tuning time (Blob, FD, SC, and SAK)	233
Table 45 Cross-project evaluation	239
Table 46 Related work summary on software bots.....	249
Table 47 Participants background information.....	260
Table 48 Selected projects for evaluating SmellyBot.....	261
Table 49 Modified Technology Acceptance Model (mTAM) scores.....	262
Table 50 Expected features and recommendations.....	266
Table 51 Detection performance of SmellyBot [176].....	270
Table 52 List of primary studies	282
Table 53 Primary studies venues and publication type.....	284
Table 54 Types of bad smells detected using DL	286
Table 55 Publicly available datasets	289
Table 56 Pre-training dataset description	290
Table 57 Fine-tuning datasets description	293

LIST OF FIGURES

Figure 1 The general structure of DL models	30
Figure 2 Search process	44
Figure 3 Distribution of primary studies over the year	49
Figure 4 Bad smell categories	54
Figure 5 Types of bad smells detected using DL	55
Figure 6 Learning approach vs. purpose of DL usage	64
Figure 7 Features used in DL-based bad smell detection	70
Figure 8 DL models used for predictive classification	78
Figure 9 DL models used for representation learning	81
Figure 10 PS distribution over the training considerations	85
Figure 11 Number of datasets used by the PSs	96
Figure 12 Detection performance with respect to the datasets	104
Figure 13 Detection performance with respect to the bad smells	106
Figure 14 Detection performance with respect to the DL models for predictive classification	108
Figure 15 Detection performance with respect to the DL models for representation learning	108
Figure 16 Detection performance with respect to the encoding techniques	110
Figure 17 Detection performance with respect to the preprocessing techniques	112
Figure 18 Detection performance with respect to the features	114
Figure 19 Overview of CoRT	142
Figure 20 CoRT model architecture	153
Figure 21 Pre-trained models accuracy	164
Figure 22 Pre-trained models loss	164
Figure 23 Effect of training hyperparameters on model loss	168
Figure 24 Effect of model size on model loss	169
Figure 25 Detection performance boxplots of CoRT compared to baseline and feature-based	183
Figure 26 Cross-project performance heatmap for Data Class	186
Figure 27 Cross-project performance heatmap for God Class	187
Figure 28 Cross-project performance heatmap for Feature Envy	188
Figure 29 Cross-project performance heatmap for Long Method	189
Figure 30 Online shopping class diagram	202
Figure 31 Fine-tuning datasets construction	211
Figure 32 Overview of MoRT	213
Figure 33 Detection performance boxplots of MoRT compared to baseline and feature-based	238

Figure 34 SmellyBot architecture	252
Figure 35 The processing time of SmellyBot	268
Figure 36 Cross-project performance heatmap for Data Class	297
Figure 37 Cross-project performance heatmap for God Class.....	298
Figure 38 Cross-project performance heatmap for Feature Envy	299
Figure 39 Cross-project performance heatmap for Long Method	300

LIST OF ABBREVIATIONS

A	:	Number of Self-Attention Heads
ACC	:	Abstract Concrete Class
AE	:	Autoencoder
ANN	:	Artificial Neural Network
AS	:	Anti-Singleton
AST	:	Abstract Syntax Trees
AST-LSTM	:	AST-based Long Short-Term Memory
AST-RNN	:	AST-based Recursive Neural Network
ASTNN	:	AST-based Neural Network
AUC	:	Area Under Curve
Acc	:	Accuracy
Avg	:	Average
B	:	Batch Size
b	:	Bias
BDG	:	Bytecode Dependency Graph
BH	:	Broken Hierarchy

BM	:	Broken Modularization
BPTS	:	Backpropagation Through Structure
Bi-GRU	:	Bidirectional Gated Recurrent Unit
Bi-LSTM	:	Bidirectional Long Short-Term Memory
CAC	:	Concrete Abstract Class
CBOW	:	Continuous Bags of Words
CC	:	Complex Conditional
CDM	:	Cyclic D Modularization
CDSP	:	Data Should Be Private
CF	:	Creeping Featurism
CFG	:	Control Flow Graph
CG	:	Conjugate Gradient
CH	:	Cyclic Hierarchy
CM	:	Complex Method
CMT	:	Comments
CNN	:	Convolutional Neural Networks
CSD	:	Code Semantic Dependency

DAE	:	Denoise Autoencoder
DC	:	Data Class
DCH	:	Divergent Change
DCL	:	Data Clumps
DE	:	Deficient Encapsulation
DEC	:	Dead Class
DFC	:	Duplicated Features in Classes
DFG	:	Data flow graph
DFH	:	Duplicated Features in Hierarchy
DH	:	Deep Hierarchy
DL	:	Deep Learning
DNN	:	Deep Neural Networks
DR	:	Dropout Rate
DSL	:	Domain-Specific Language
DT	:	Decision Tree
DU	:	Duplication
E	:	Embedding Dimension

EA	:	Excessive Alternation
ECB	:	Empty Catch Block
EMF	:	Eclipse Modeling Framework
F1	:	F1-score
FD	:	Functional Decomposition
FE	:	Feature envy
FS	:	feature selection
GA	:	genetic algorithm
GAN	:	generative Adversarial networks
GAT	:	Graph attention networks
GC	:	God Classes
GCN	:	Graph convolutional network
GGNN	:	Gated graph neural networks
GMN	:	Graph Matching Networks
GNN	:	Graph Neural Network
GP	:	God Package
GRU	:	gated recurrent units

H	:	Number of Units of Hidden Layers
HLM	:	Hub-like Modularization
IA	:	Imperative Abstraction
IM	:	Insufficient Modularization
JDT	:	Java Development Tools
KNN	:	K-Nearest Neighbors
L	:	Number of Layers
LC	:	Large class
LDA	:	Linear Discriminate Analysis
LF	:	Lava Flow
LM	:	Long Method
LOC	:	lines of code
LPL	:	Long Parameter List
LR	:	Learning Rate
LSTM	:	Long Short-Term Memory
LZC	:	Lazy Class
MA	:	Multifaceted Abstraction

MC	:	Misplaced class
MCC	:	Matthews Correlation Coefficient
MDD	:	Model-Driven Development
MH	:	Missing Hierarchy
ML	:	machine learning
MLP	:	Multilayer perceptron
MM	:	Middle-Man
MMO	:	Missing Modularization
MP	:	Multiple Personality
MPH	:	Multipath Hierarchy
MSE	:	Mean Square Error
Max	:	Maximum
Min	:	Minimum
Mins	:	Minutes
NA	:	Not Available
NB	:	Naive Bayes
NL	:	Natural Language

NLP	:	Natural Language Processing
NOC	:	Number of Classes
NOM	:	Number of Methods
NTN	:	Neural Tensor Network
OO	:	Object-oriented
P	:	Precision
PACE	:	Position Aware Character Embedding
PCA	:	Principal Component Analysis
PDG	:	Program Dependency Graph
PEU	:	Perceived Ease of Use
PG	:	Poltergeists
PIH	:	Parallel Inheritance Hierarchies
PL	:	Programming Language
PU	:	Perceived Usefulness
PV-DM	:	Paragraph Vector Distributed Memory
R	:	Recall
RAE	:	Recursive Autoencoders

RB	:	Refused Bequest
RBFN	:	Radial Basis Function Networks
RCR	:	Redundant Container Relation
RF	:	Random Forest
RFE	:	Recursive feature elimination
RH	:	Rebellious Hierarchy
RMSPROP	:	Root Mean Square Prop
RNN	:	Recurrent Neural Networks
ROC	:	Receiver Operating Characteristic
RPR	:	Refused Parent Request
ReLU	:	Rectified Linear Unit
RtNN	:	Recurrent Neural Network
RvNN	:	Recursive Neural Network
SAK	:	Swiss Army Knife
SBFL	:	Spectrum-Based Fault Localization
SBMO	:	Sequential Bayesian Model-based Optimization
SC	:	Spaghetti Code

SG	:	Speculative Generality
SGD	:	Stochastic Gradient Descent
SLR	:	Systematic Literature Review
SMOTE	:	Synthetic Minority Oversampling Technique
SP	:	Specters
SS	:	Shotgun Surgery
ST-Tree	:	Statement Tree
SVD	:	Singular Value Decomposition
SVM	:	Support Vector Machine
SW	:	Spider's Web
SWS	:	Switch Statement
T-LSTM	:	Tree Long Short-Term Memory
TB-CNN	:	Tree-Based Convolutional Neural Networks
TCN	:	Temporal Convolutional Networks
TF-IDF	:	Term Frequency-Inverse Document Frequency
TPE	:	Tree-Structured Parzen Estimator
UA	:	Unnecessary Abstraction

UE	:	Unexploited Encapsulation
UF	:	Undue Familiarity
UNA	:	Unutilized Abstraction
VAE	:	Variational Autoencoder
W2V	:	Word2Vec
WH	:	Wide Hierarchy
W	:	Weight

|

ABSTRACT

Full Name : Amal Abdulrahman Ali Alazba

Thesis Title : A Framework for Software Code and Model Smell Detection through
Self-Supervised Learning

Major Field : Computer Science

Date of Degree : December 2023

Bad smells negatively impact software quality metrics such as understandability, reusability, and maintainability. Reduced costs and enhanced software quality can be achieved through accurate bad smell detection. While various approaches, including traditional machine learning (ML) and deep learning (DL) methods, have been proposed and shown to detect bad smells effectively, they require extensive manually labeled data, which is expensive and challenging to scale. This research aims to propose a new bad smell detection approach that utilizes self-supervised learning to learn intermediate representations using unlabeled data and then fine-tune these representations on multiple tasks. The primary motivation behind the self-supervised learning approach is to harness the vast amount of data now accessible for unsupervised semantic feature learning, reduce the model's reliance on labeled data, and consequently enhance the generalizability of deep learning models. In this research, we conducted a systematic literature review where we analyzed 67 studies on DL-based bad smell detection and obtained quantitative and qualitative data. Also, we introduced a Code Representation with Transformers (CoRT) model to learn the semantic and structural features of the source code by training transformers to recognize masked reserved words that are applied to the code given as input. We extended our approach to detect bad smells at the model level. We introduced

the MoRT (Model Representation with Transformers) model that learns semantics and the structural features of the UML class diagrams by training Transformers to recognize masked keywords. We empirically demonstrated that the defined proxy tasks for CoRT and MoRT provide a powerful method for learning semantic and structural features. We exhaustively evaluated CoRT on four downstream tasks: detection of the Data Class, God Class, Feature Envy, and Long Method code smells. In addition, we assessed MoRT using four bad smells that can be detected through class diagrams, namely the Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife. We compared our results with those of supervised learning and a feature-based approach. Finally, we conducted a cross-project experiment to evaluate the generalizability of our method to unseen labeled data. The proposed method has been shown to detect bad smells at the code and model levels effectively. Compared with the baseline model and feature-based approaches, our method achieved better detection performance and had a high capability to detect bad smells in unseen datasets. Automating code smell detection through a software bot significantly improves efficiency, code quality, and developer productivity. A software bot named SmellyBot was developed to automate the detection of code smells. We seamlessly integrated SmellyBot within the GitHub framework, incorporating automated reporting to enhance code smell detection. Our evaluation involved 43 developers to assess user perception and recommended features, alongside an analysis of SmellyBot's performance in six real-world projects to examine its efficiency and effectiveness. Also, developers perceive SmellyBot as highly useful and easy to use. |

ملخص الرسالة

الاسم الكامل: أمل بنت عبدالرحمن علي العذبه

عنوان الرسالة: إطار عمل الكشف عن العيوب في النصوص البرمجية والنماذج من خلال التعلم الخاضع للإشراف الذاتي

التخصص: علوم الحاسب الآلي

تاريخ الدرجة العلمية: ديسمبر 2023

تؤثر العيوب البرمجية سلبًا على مقاييس جودة البرامج مثل سهولة الفهم وإعادة الاستخدام والصيانة. يمكن تحقيق خفض التكاليف وتحسين جودة البرامج من خلال الكشف الدقيق عن هذه العيوب. في حين تم اقتراح أساليب مختلفة، بما في ذلك أساليب التعلم الآلي التقليدية (ML) والتعلم العميق (DL)، وثبتت فعاليتها في اكتشاف العيوب البرمجية، إلا أنها تتطلب بيانات ضخمة مصنفة يدويًا، وهي مكلفة ويصعب استخدامها على نطاق واسع. يهدف هذا البحث إلى اقتراح نهج جديد للكشف عن العيوب البرمجية باستخدام التعلم الخاضع للإشراف الذاتي لتعلم التمثيلات المتوسطة باستخدام البيانات غير المُعلّمة، ثم ضبط هذه التمثيلات في مهام متعددة. الدافع الأساسي وراء نهج التعلم الخاضع للإشراف الذاتي هو تسخير الكمية الهائلة من البيانات التي يمكن الوصول إليها الآن لتعلم الميزات الدلالية غير الخاضعة للإشراف، وتقليل اعتماد النموذج على البيانات المصنفة، وبالتالي تعزيز إمكانية تعميم نماذج التعلم العميق. في هذا البحث، أجرينا مراجعة منهجية للأدبيات حيث قمنا بتحليل 67 دراسة حول الكشف عن العيوب البرمجية باستخدام التعلم العميق وحصلنا على بيانات كمية ونوعية. كما قدمنا نموذج تمثيل الكود مع المحولات (CoRT) لمعرفة السمات الدلالية والهيكلية للنصوص البرمجية من خلال تدريب المحولات على التعرف على الكلمات المقنعة التي يتم تطبيقها على النصوص البرمجية المدخلة. كما قمنا بتوسيع نهجنا لاكتشاف العيوب البرمجية على مستوى النموذج. قدمنا نموذج MoRT (تمثيل النموذج مع المحولات) الذي يتعلم الدلالات والميزات الهيكلية لمخططات الفئة من خلال تدريب المحولات على التعرف على الكلمات الرئيسية المقنعة. لقد أثبتنا تجريبيًا أن المهام المحددة لتدريب CoRT و MoRT توفر طريقة قوية لتعلم الميزات الدلالية والهيكلية. لقد قمنا بتقييم CoRT بشكل شامل في أربع مهام فرعية: ال Data Class، و God Class، و Feature، و Env، و Long Method. بالإضافة إلى ذلك، قمنا بتقييم MoRT باستخدام أربع عيوب برمجية يمكن اكتشافها من خلال مخططات الفئة، وهي Blob، و Functional Decomposition، و Spaghetti Code، و Swiss Army Knife. قمنا بمقارنة نتائجنا مع نتائج نموذجين: التعلم تحت الإشراف والنهج القائم على الميزات. أخيرًا، أجرينا تجربة مشتركة بين المشاريع لتقييم إمكانية تعميم طريقتنا على البيانات المصنفة غير المرئية. لقد أثبتت الطريقة المقترحة فعاليتها في الكشف عن

الروائح الكريهة على مستوى النصوص البرمجية ومستوى النماذج. بالمقارنة مع النموذج الأساسي والأساليب القائمة على الميزات، حققت طريقتنا أداء كشف أفضل ولديها قدرة عالية على اكتشاف العيوب البرمجية في مجموعات البيانات غير المرئية. توفر أتمتة اكتشاف رائحة التعليمات البرمجية من خلال روبوت برمجي فوائد كبيرة من حيث الكفاءة وجودة التعليمات البرمجية وإنتاجية المطور. تم تطوير روبوت برمجي، يُسمى SmellyBot، لأتمتة اكتشاف العيوب البرمجية. لقد قمنا بدمج SmellyBot بسلاسة في إطار عمل GitHub، ودمجنا التقارير الآلية لتحسين اكتشاف العيوب البرمجية. شمل تقييمنا 43 مطورًا لتقييم تصور المستخدم والميزات الموصى بها، إلى جانب تحليل أداء SmellyBot في ستة مشاريع في العالم الحقيقي لفحص كفاءته وفعاليته. تشير النتائج إلى أن SmellyBot مفيد للغاية وسهل الاستخدام من وجهة نظر المطورين. |

CHAPTER 1

INTRODUCTION

Software bad smells is a general term used to refer to the presence of internal design [1] and coding [2] problems. Bad smells were first introduced by Fowler et al. [2] in 1999, which refers to poor design and implementation choices of the source code that breach important principles. Bad smells impose a negative impact on software quality attributes like [3], [4]: understandability, reusability, and maintainability. A bad smell is not considered a fault or problem by itself. Instead, it is an indication of a problem that results from bad practices and may affect software maintenance and evolution. Bad smell detection is the process of identifying bad smells.

Making poor implementation and design decisions that violate the main design principles leads to the need for code cleaning (i.e., refactoring). Software engineers consider the refactoring process when the software is being used for a long time and has the potential for additional development [5]. Refactoring is defined as the process of improving the software's internal structure quality without altering the system functionality [2]. Even though refactoring is considered a time-consuming and labor-intensive process, many benefits are gained from the refactoring process, such as improved software quality and design, increased software readability and understandability, locating software defects, and facilitating the software development process [2]. However, refactoring is tricky and raises some challenges since it requires changes that might introduce new bugs and alter the

software's internal behavior [6]. Therefore, a key element to the refactoring process's success is identifying the code or model fragments that need to be changed (i.e., bad smells).

1.1 Problem Statement and Motivation

Complex organizations have thrived through the use of software systems to automate various tasks, such as handling transactions and managing the payroll. Also, they have contributed to defining new advanced ways to create business-to-consumer relationships through social networks and cloud computing. This has led to improving organizations' efficiency and effectiveness. Over time, software systems become increasingly large, complicated, and hard to deal with. With the high spread of information technologies, there is an increasing demand for software systems that meet non-functional requirements such as maintainability, reliability, and efficiency in addition to cost [7]. As a result, not only is software creation more complicated than ever, but even the maintenance process to correct and improve the systems has become very problematic due to the stringent schedules and deadlines. The cost of maintaining a software system is almost one hundred times higher than the costs of production [8].

In a standard example, a software system's source code undergoes many regular revisions to adjust to new requirements or patch urgent glitches [9]. Under tight deadlines, it is often challenging for software engineers to devote sufficient time to implement changes that are fully aligned with well-known design principles. This situation can vary depending on several factors, such as project complexity, available resources, and the specific practices of an organization.. The modifications are completed in a hurry, resulting in poor design

and implementation choices that might diminish the original design and possibly introduce the so-called bad smells [2]. For any reasonable-sized program, manually traversing the source code or class diagram and finding bad smells is a tedious, difficult, and time-consuming task for developers. Furthermore, developers may be unaware of the different bad smells and whether they exist or not. Smell detectors can help increase software efficiency on a wide scale by automating this operation.

Several scholars have been attentive in identifying and studying the characteristics of source code influenced by bad smells and their impact on maintainability [10]. Moreover, many empirical studies have explored the evolution and longevity of bad smells [11], [12], developers' perspective of bad smells [13], and their relationship with fault-proneness [14]. All these experiments prompted the academic community to devote time and resources to the development of techniques that can automatically detect bad smells in source code or from UML models, such as class diagrams, to assist developers in identifying the parts of the code or model that are most likely to be compromised. The first step towards identifying software quality issues for improving maintainability is detecting bad smells, whereas appropriate refactoring techniques can be applied.

Various methods to detect bad smells have been developed such as metrics-based and rule-based approaches [15]. Most of the metrics-based suggested detecting bad smells by using a series of thresholds dependent on structural parameters (i.e., software metrics) to classify core symptoms that describe specific bad smells [16]. Another approach was proposed using the rule-based to detect bad smells by combining multiple structural parameters and formulating a rule that varies depending on the form of the bad smells [17]. One major limitation of these methodologies is that their performance depends heavily on the

threshold values and/or the combination rules, and until now, there is no agreement on the standard values and rules [18]. Moreover, the rule-based approaches can be inflexible, failing to adapt to the evolving nature of programming practices and languages.

To mitigate these limitations, other methodologies, such as machine learning and deep learning, have sparked the interest of several researchers in recent years [19] due to the capabilities of ML and DL models in recognizing complex patterns that are not easily captured by simple metrics or predefined rules. In addition, ML and DL models can improve over time with more data, adapting to new coding practices and languages. Although these techniques are promising, one major limitation is that they need a vast amount of manually annotated datasets [18]. However, there is a lack of publicly available datasets that can be leveraged in such methodologies [19]. The manual annotation is a tedious, time-consuming, and error-prone process. For instance, in one study [20], 600 code smell instances were manually annotated by five people, and it nearly cost 100 hours per person. To mitigate the error in the manual annotation, researchers usually attempt to use strategies like the majority vote [20], which requires that at least three individuals annotate each data point. Also, in terms of money, the labeling is costly, whereas image labeling costs \$6.4 per image, according to a data labeling firm called Scale¹. Considering this scenario, a dataset of images with tens of thousands of instances may cost around one million dollars. Using machine learning techniques with labeled datasets is known as supervised learning [21]. The problem with supervised learning is not solely that it needs a labeled dataset; it also suffers from generalization issues [22], where the model is trained on only a subset of the data (the labeled ones), hence failing to detect unseen instances.

¹ <https://scale.com/pricing>

This research aims to mitigate these limitations by introducing a new detection approach that utilizes unlabeled data.

Detecting bad smells at the early stages of the development process helps in identifying design problems earlier. It also allows for performing quality assessment at early stages, hence reducing maintenance costs [23]. However, most deep learning-based studies have detected bad smells at the code level [24]; only a few studies detected bad smells from UML design models. Compared to the implementation phase, it is more difficult to detect bad smells at the design stage because of the lack of information in the models compared to the code [25]. In this research, we investigated the detection of bad smells from structural UML design models using unlabeled data, which is the class diagram, which represents the structure of the programs. The main goal is to mitigate the limitations of existing ML and DL approaches that depend heavily on labeled datasets.

Another limitation is the lack of automated tools for detecting bad smells. The use of autonomous bots for automatic support of software activities is becoming more popular [26] [27]. For example, bots have been utilized for automatic code refactoring [28] [29], refactoring documentation [30], repair bugs [31], and code reviews [32] [33]. One main benefit of software bots is that they reduce the workload of repetitive activities [34]. Wyrich and Bogner [28] highlighted the value of refactoring bots and encouraged them to be adopted in practice as they are expected to increase efficiency and effectiveness. Moreover, software bots are also expected to enhance both project quality and developer productivity, as discussed by Storey and Zagalsky's [26]. A recent study has shown that software bots are perceived as useful and generally accepted by developers [35]. Although software bots are promising, we can observe that most of the existing ML techniques have

not been automated in a tool that can support the developers in detecting code smells. Therefore, there is a necessity for automating the detection process via the design and development of a software bot.

The main motivation of this approach is to mitigate the above limitations. This research introduces a new bad smell detection approach using unlabeled data. Current deep learning-based approaches depend heavily on manual annotation of unlabeled data; hence, training an efficient deep learning model requires a vast amount of labeled data. However, there is a lack of large-scale datasets in the context of bad smells, making it challenging to train deep-learning models. This leads to a vicious cycle, whereas self-supervised learning can be leveraged to break this cycle.

1.2 Research Objectives

The goal of this research is **to introduce a new approach for model and code smell detection through self-supervised learning**. The main objectives of this research include:

1. Conduct an intensive review of state-of-the-art approaches for bad smell detection using deep learning.
2. Introduce a code smell detection approach that uses unlabeled data to learn intermediate representations of the source code and validate the proposed approach in detecting a set of code smells.
3. Introduce a model smell detection approach that uses unlabeled data to learn intermediate representations of the class diagrams and validate the proposed approach in detecting a set of model smells.

4. Extend and build two large datasets for bad smell detection, for the detection at the code level and model level.
5. Conduct a comparative study between the proposed approach and other existing approaches, including supervised learning and feature-based approaches.
6. Evaluate the generalizability of the proposed approach by conducting a cross-project evaluation.
7. Automate the detection process by designing and implementing a software bot.

In this research, we formulated the high-level research questions to be answered during the research:

- **RQ 1.** What are the most representative bad smells detected using DL approaches?

This question aligns with the first objective, which is to conduct an intensive review of state-of-the-art approaches for bad smell detection using deep learning. It aims to identify the most prevalent or significant bad smells that are detectable through current DL methodologies, thereby establishing a foundation for further research and development in this area. All sub-RQs in Chapter 3 are related to Objective 1.

- **RQ 2.** How can we detect bad smells from source code through self-supervised learning?

This question addresses multiple objectives - specifically, objectives 2, 4, 5, and 6. It explores the development and validation of a code smell detection approach utilizing self-supervised learning (Objective 2: RQ1, RQ2, RQ3, and RQ4 in Chapter 4), extending and building large datasets for bad smell detection at the code level (Objective 4), conducting comparative studies with existing methods

(Objective 5: RQ5 in Chapter 4), and evaluating the generalizability of the approach across different projects (Objective 6: RQ5 in Chapter 4).

- **RQ 3.** How can we detect bad smells from the class diagram through self-supervised learning?

This question addresses objectives 3, 4, 5, and 6. This research question delves into developing a model smell detection approach for class diagrams, again leveraging self-supervised learning. This aligns with introducing and validating a novel approach for model smell detection (Objective 3: RQ1 and RQ2 in Chapter 5), contributing to the expansion of datasets for model-level bad smell detection (Objective 4), enabling comparative analysis with other methodologies (Objective 5: RQ3 in Chapter 5), and assessing the cross-project applicability of the proposed model (Objective 6 RQ3 in Chapter 5).

- **RQ 4.** How can we fully automate the detection of bad smells by developing a software bot?

Directly addresses Objective 7, this question focuses on the technological advancement and practical application aspect of the research. It involves the design and implementation of a software bot to automate the bad smell detection process, thereby encapsulating the end-goal of creating a user-friendly, efficient, and automated solution for detecting bad smells in software development. All sub-RQs in Chapter 6 are related to Objective 7.

1.3 Research Methodology

To achieve our research objective, the research methodology to be followed in this research has the following phases:

- 1) Conduct a literature review to identify state-of-the-art approaches for bad smell detection using deep learning. The initial phase of this research involved a comprehensive and systematic review of the current and relevant literature in the field. This process was crucial to understand the state-of-the-art, identify gaps in existing research, and establish the foundation for the study. The following steps outline the approach taken in this review:
 - a. Search Strategy and Paper Selection: A set of databases is selected and a systematic search strategy is employed, involving a combination of relevant keywords and phrases linked to the research. Initial screening based on titles and abstracts is conducted, followed by a more detailed review of full texts to ensure the selected papers are closely aligned with my research objectives.
 - b. Classification and Analysis: The selected papers are then classified based on various factors such as the bad smells type, DL model, and the features used.
 - c. Identification of Gaps and Limitations: the gaps in the current literature are identified, particularly in the areas of DL-based bad smell detection. Limitations of existing approaches are highlighted which informed the direction and focus of our subsequent research.

- d. Synthesis and Documentation: The findings from this comprehensive literature review were synthesized to build a coherent understanding of the current state of research in the field.
- 2) Design and implement a new approach for code smell detection. After a thorough review of the existing literature and identification of gaps in current methodologies, the next phase involved designing and implementing a novel approach for code smell detection. This approach was guided by the limitations and challenges identified in previous. The steps are as follows:
- a. Formulate the pretext tasks: pretext tasks are pre-designed tasks for the deep learning model to solve.
 - b. Construct large-scale benchmarking code smell datasets.
 - c. Build and train a code smell detection model through self-supervised learning.
 - d. Apply on the downstream task: the application of the pre-trained model for detecting specific code smell. Small, labeled instances will be used to fine-tune the model.
 - e. Conduct a comparative study to investigate to which extent the proposed approach will enhance the performance of code smell detection over supervised and feature-based approaches.
- 3) Design and implement a new approach for model smell detection. After a thorough review of the existing literature and identification of gaps in current methodologies, the next phase involved designing and implementing a novel approach for bad smell

detection at the model level. This approach was guided by the limitations and challenges identified in previous studies. The steps are as follows:

- a. Formulate the pretext tasks: pretext tasks are pre-designed tasks for the deep learning model to solve.
- b. Construct large-scale benchmarking model smell datasets.
- c. Build a model smell detection model through self-supervised learning.
- d. Learn a proper feature representation: by exploiting some property of data that can be useful.
- e. Apply on the downstream task: the application of the pre-trained model for detecting specific model smell. Small, labeled instances will be used to fine-tune the model.
- f. Validate the effectiveness of the proposed approach: Conduct a comparative study to investigate to which extent the proposed approach will enhance the performance of model smell detection over supervised and feature-based approaches.
- g. Collect and analyze the results: conduct a comparative analysis of the performance of the two approaches in detecting bad smells.

4) Developing a software bot to automate the detection process. Developing a software bot to automate the detection process of the proposed method. This bot would automate the identification of code smells, making the process more efficient and user-friendly. The steps are as follows:

- a. Design the bot by specifying its inputs, outputs and components.

- b. Implement the bot and integrate its components with the proposed approach.
- c. Evaluate the developers' perception of the designed bot.

1.4 Dissertation Organization

This dissertation is composed of seven chapters as follows:

- Chapter 2 discusses background information on bad smells, deep learning, and software bots.
- Chapter 3 provides an extensive systematic literature review (SLR) on existing studies about deep learning-based bad smell detection. In this chapter, we summarize and synthesize the studies that used DL techniques for bad smell detection, identify the bad smell types, DL techniques, features used as inputs, evaluation methods of the DL models, and datasets and systems used in the DL model training. Also, we explore the detection performance of DL models in relation to other factors (i.e., datasets, features, encoding techniques, preprocessing techniques, and bad smell types).
- Chapter 4 introduces the detection of code smells using a Code Representation with Transformers (CoRT) model to learn the semantic and structural features of the source code by training transformers to recognize masked reserved words that are applied to the code given as input. The main framework and the proposed pipeline with a description of the components are discussed. Also, it presents a comprehensive evaluation of the proposed method on four code smells, including Data Class, God Class, Feature Envy, and Long Method.

- Chapter 5 introduces MoRT (Model Representation with Transformers) to learn semantics and the structural features of the UML class diagrams by training Transformers to recognize masked keywords from XMI representation. The main framework and the proposed pipeline with a description of the components are discussed. Also, it presents a comprehensive evaluation of the proposed method on four model smells that can be detected from the UML class diagram, including Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife.
- Chapter 6 presents the automation of the detection of bad smells by designing and developing a software bot named SmellyBot. Also, we discuss the integration of the bot within the GitHub framework, incorporating automated reporting to facilitate the detection. Finally, we present the evaluation of the bot to assess developers' perception alongside an analysis detection performance in production.
- Chapter 7 concludes the research work by summarizing the key findings, emphasizing the contributions, and highlighting the main challenges and limitations. |

CHAPTER 2

BACKGROUND

In this chapter, we present background information. This chapter contains (1) the definition and types of bad smells, (2) a description of various machine learning techniques, including supervised learning and unsupervised learning, (3) an introduction to deep learning, and (4) definitions of software bots.

2.1 Bad Smells

Software bad smells is a general term used to refer to the presence of internal design [1] and coding [2] problems. Code smells were first introduced by Fowler et al. [2] in 1999 and refers to poor implementation choices of the source code that breach essential principles. They negatively impact software quality attributes like [3], [4]: understandability, reusability, and maintainability.

2.1.1 Code level smells

Fowler's book [2] is prominent in the literature on identifying 22 code smells focusing on object-oriented systems. In Fowler's book, code smells are defined in an informal style and associated with a practical guide to refactoring techniques to resolve them. Wake [36] broadened the original catalog by considering some problematic aspects identified by software developers in practice.

The 22 code smells defined by Fowler et al. [2] are well-known and adapted mainly by the researchers. In code smell detection, God Class, Code Clones, Long Method, Feature Envy,

and Data Class are the most investigated code smells [15]. In this section, we briefly describe each smell.

Data Class – is a class-level code smell in which the class only holds the data (attributes) with only getters and setters without proper functionalities. Such classes are usually utilized by other classes; however, they don't manipulate their own data. Having data classes might be an indication of a lack of abstraction. To eliminate this code smell, the developer can look for functionalities better placed in these classes and then use the Extract Method refactoring technique.

God Class – is a class-level code smell that refers to a class with many responsibilities and numerous attributes/functions. Having a large class hinders the system's growth as maintenance becomes difficult and requires more time and effort. Large classes violate the principle of the *Single-Responsibility* [37], which states that a class should do exactly one thing (i.e., have one responsibility). This code smell can be removed by applying refactoring techniques such as Extract Class and Extract Interface.

Long Method – is a method-level code smell that occurs when a method is overly long and has many lines of code. Long methods are usually difficult to read, understand, and accommodate new changes. Like Large Class smell, this code smell violates the *Single-Responsibility Principle* [37]. One refactoring technique to resolve this code smell is to break down the function into smaller components that can be called from the primary function.

Feature Envy – is a method-level code smell that exists when a method is more interested in the functionalities of another class than its enclosing class. Alternatively phrased, it uses

data and accesses methods from another class. This code smell can be an indication of a strong coupling between the method and the other class. One refactoring strategy to eliminate the Feature Envy smell is to move the method to the class that is strongly coupled with the method.

Code Clones – also known as duplicated code, refers to having identical codes spread across the source code due to copying and pasting code snippets. Duplicated code makes the modification more complex and error-prone since a single change may require modification in multiple locations in the source code. This code smell breaks the principle of the *Don't-Repeat-Yourself* [38], which states that a code fragment that already exists should not be rewritten elsewhere. We can lessen the code clones by extracting the standard functionality where it can be called and used in different locations.

2.1.2 Design level smells

Brown et al. [39] described 40 antipatterns related to the design and implementation of object-oriented systems. They categorized the antipatterns into three categories: development, architecture, and project management antipatterns. Gamma et al. [40] identified design patterns, solutions, and best practices often used by developers. In this section, we briefly describe the most well-known design smells described in Brown et al. book [39], including the Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife.

Blob – refers to a class characterized by its complexity, containing a large number of attributes and methods. This class often depends on surrounding data classes, forming a highly coupled relationship. Within the Blob class, there is typically low cohesion among

its methods. It is common for a Blob to encapsulate the entire functionality of an application, resembling a main program. Even when represented using object-oriented notations and languages, it still exhibits characteristics of a procedural design. The Blob is often identified by a class diagram consisting of a single complex controller class surrounded by simpler data classes. This design smell may result from inappropriate requirements allocation or iterative development processes.

Functional Decomposition – this smell arises when classes are designed with the intention of performing single functions. These main classes are associated with smaller classes, which declare numerous private fields but implement only a few methods. This design defect is typically found in class diagrams created by developers with little experience in object-oriented programming. It results in classes with single actions resembling structures commonly found in procedural languages. Functional Decomposition may emerge when developers are accustomed to calling numerous subroutines from the main routine, ignoring class hierarchy.

Spaghetti Code – it represents a software structure characterized by its lack of organization. It often results from coding and progressive extensions that compromise the clarity of the software structure. It has global variables with large method implementations that do not have parameters. The execution path of the code is often linear and lacks complexity or dynamic interactions between different parts of the code. This predictability in the code's flow is a characteristic of Spaghetti Code, where each step typically leads to the next in a straightforward manner. This will make software maintenance and extension challenging, with limited opportunities for reusing objects and modules in other systems. Even when

developed using object-oriented languages, it exhibits characteristics of poor software structure, resembling code written in procedural languages like Pascal or FORTRAN.

Swiss Army Knife – This smell is associated with a class that implements a substantial number of interfaces to offer a wide range of functionalities. Classes linked to the Swiss Army Knife provide public interfaces to access these functionalities. Unlike the Blob, which encapsulates the entire functionality of a system, the Swiss Army Knife exposes high complexity to address various needs within a part of the system. Several Swiss Army Knives may coexist in a system, and they may include utility classes.

2.1.3 Categories and classification

Categorizing software smells helps to understand their relationships better and is required to attain better comprehension [41]. Sharma and Spinellis [42] proposed four desirable proprieties when developing classification schemes for software smells. A classification has to be (1) *Exhaustive*, which means it should include all existing smells; (2) *Simple*, where the scope and granularity of the classification scheme allow categorizing the smells effortlessly; (3) *Consistent* when applying the classification by multiple individuals it should produce the same classification, and (4) *Coherent* it must produce distinct categories with no overlaps between them.

Many attempts have been made to classify the software smells based on various aspects such as the effect of the smells, the granularity and object-oriented principles [42]. Mäntylä et al. [41] categorized software smells into six categories based on how they impact the activities in software development. Their classification includes the following categories:

Bloaters – represent the types of code smells that became enormously larger over time and hindered the understandability and maintainability of the code. This category contains class-level smells, such as God Class and Data Clumps, and method-level smells, like Long Method and Long Parameter List.

Object-Orientation Abusers – includes the code smells that violate the design principles of object-oriented programming. Examples of such code smell include Switch Statements, Refused Bequests, and Temporary Fields.

Change Preventers – contain the type of code smells that hinder the development of the program where a single modification may require the modification of multiple parts of the source code. Divergent Change and Shotgun Surgery are smells that belong to this category.

Dispensables – represent the types of code smells that are unnecessary or irrelevant where deleting these parts would result in a clean and understandable code. This includes Comments, Duplicate Code, and Dead Code.

Encapsulators – contain a set of opposite code smells where the decrease of one smell may imply the increase of another. The smells in this category are Message Chains and Middle Man.

Couplers – include the types of smells that have high coupling with other components. Feature Envy and Inappropriate Intimacy are examples of the couplers category.

Ganesh et al. [43] classified the smells based on the object-oriented principle that is violated by each smell. Their classification has four main categories:

Abstraction smells – this group covers the code smells that violate the abstraction principle. An example of this group is the Data Class smells, which is considered a passive abstraction smells in which the class only holds the data without functions.

Modularization smells – this category includes the code smells that violate the principle of modularity. Middleman is an example of a code smell that belongs to this group since it depends heavily on other classes, i.e., most of its functions are delegated to different classes.

Encapsulation smells – includes the set of smells that do not adhere to the encapsulation principle. Inappropriate intimacy belongs to this category because it is a class-level abstraction that accesses the private component of another class.

Hierarchy smells – contains the smells that violate the principle of hierarchy. For instance, the Switch Statements smell requires explicit checks when a change is made. Hence, it does not use the polymorphism in the hierarchy.

A granularity-based classification was introduced by Wake [36], where the code smells are categorized based on the level of granularity. Two main categories were introduced:

Smells within Classes – this category contains the code smells within a class and can easily recognized with software metrics such as long parameter lists, long methods, and god classes. It also includes unnecessary code snippets that need to be eliminated, like dead code and duplicated code.

Smells between Classes – this category represents the smells that can be found between the classes, such as envy, refused bequest, message chains, and parallel inheritance hierarchies.

Perez [4] suggested another granularity-based smell classification. However, the classification levels were designed according to the level of the problem. The smell cataloging contains the following categories:

Low-level smells – covers the set of smells related to low-level problems (i.e., problems in the code). This category includes smells like Large Class and Long Method.

High-level smells – include the smells associated with high-level and complex problems (e.g., architectural or design problems). These smells can evolve by combining multiple low-level smells, such as the Blob smell.

Moha et al. [17] proposed a smell classification at two levels. The first level classifies the smells as inter-class and intra-class, similar to the Wake [36] classification. Then, at the second level, the smells are categorized based on the detection technique that can be used as follows:

Structural smells – covers the smells detected from the code's structure using static analysis methods. This includes Data Class, and Message Chain code smells.

Lexical smells – are the smells where we can leverage natural language processing techniques to detect them like Comments.

Measurable smells – consist of the smells that can be detected using metrics such as Long Method, Large Class, and Long Parameter List.

2.2 Machine learning techniques

Machine learning classifiers are computational models that can predict classes or make decisions by learning complex patterns from the data [44]. There are three main categories for the machine learning paradigm [45]: supervised, unsupervised, and reinforcement learning. In supervised learning, the algorithms are trained using labeled data where each instance has a known input and output. However, when the learning attempts to draw patterns from input without labeled output data, the learning becomes unsupervised. In the third learning paradigm, the output is a sequence of steps learned by interacting with an environment.

2.2.1 Supervised learning

Supervised learning [21] is the process of learning the mapping function $Y = f(X)$ that maps between the input X to the output Y . The input consists of a set of features called independent variables, while the output consists of labels, known as dependent variables, that are assigned to the inputs. The outputs (i.e., labels) are usually assigned manually. The learning algorithm is trained using the inputs and outputs to approximate the function f . Once the approximation has been established, the trained model can predict the label of new data points. It is called supervised because the ground truth labels guide the learning process. There are mainly two types of supervised learning problems: regression and classification. A regression problem is a set of problems that aims to predict a real number value, for instance, an individual's age or income. On the other hand, the classification problems aim to predict a category, for example, if a patient belongs to a healthy or unhealthy category.

Code smell detection is a binary classification problem, with the output being either smelly or not. Numerous supervised learning algorithms exist, and they mainly differ in how they approximate the function f . We give a brief description of a set of well-known supervised learning algorithms:

Decision Tree (DT) [46] is a data classification and regression algorithm that produces a tree structure for the input data. The constructed tree has a root node representing the entire data, internal nodes dividing the data into two or more sets based on a test attribute, and leaf nodes providing the class label for each path. The decision tree algorithm has many implementations, viz. ID3, C4.5, C5.0, and CART. In their study, decision trees are built using the CART algorithm.

Support Vector Machines (SVM) [47] is a classification and regression algorithm that creates a hyperplane in n -dimensional space where n represents the number of features plotted in the space. The hyperplane segregates the data points of each class such that the data points are distinctly separated. A good separation of the data points is achieved by maximizing the margin between the classes. SVM algorithm utilizes various types of kernel functions such as linear, polynomial, sigmoid, and radial basis function networks (RBFN).

Naïve Bayes (NB) [48] is a simple classification algorithm based on probabilistic. It utilizes Bayes' theorem to estimate the probability of a given feature set belonging to a particular class. It makes naïve assumptions on the input where it expects the features to be independent of each other and assumes that all features have the same importance, i.e., they have an equal effect on the outcome. NB algorithms differ in the algorithm's probability distribution (e.g., Bernoulli, Gaussian, and Multinomial).

Logistic Regression (LR) [44] is a probabilistic classifier used to predict whether an instance belongs to a certain class. It uses the logistic function as an activation function, giving output values between 0 and 1. Then, a threshold value is specified to classify the data input into a class.

Multilayer Perceptron (MLP) [49] is an Artificial Neural Network (ANN) consisting of multiple layers starting from an input layer, one or many hidden layers, and an output layer. The input layer passes the input data into the network, then the hidden layer tries to separate the data linearly based on the class label, and the output layer gives the prediction results approximated by the MLP.

Stochastic Gradient Descent (SGD) [50] is a linear data classification algorithm that estimates the gradient of a function based on random samples selected at each iteration. *SGD* uses other linear algorithms (e.g., LR and SVM) as a convex loss function. *SGD* will be built using LR.

Gaussian Process (GP) [51] is a nonlinear classifier that solves classification and regression problems. GP is a stochastic process that finds a probability distribution over a function $f(x)$ consistent with the input data. To produce a final distribution, it starts with an initial distribution updated as data instances are observed. GP uses a kernel function to calculate the similarity of a new feature set to other instances to predict its class.

K Nearest Neighbors (KNN) [52] is a data classification and regression algorithm that classifies a new instance based on k number of instances (neighbors) that are closest in distance to that instance. The new instance is classified by choosing the majority vote among its k nearest neighbors.

Linear Discriminate Analysis (LDA) [53] is a linear classifier for classification and regression problems. It calculates statistical properties such as the mean and the variance of a given feature set for each class. Then, the statistical properties are fed into the LDA classifier to make predictions by estimating the probability (using Bayes' theorem) that a new feature set belongs to each class. The class with the highest probability is assigned to the set.

2.2.2 Unsupervised learning

Unsupervised learning [54] is the process of discovering hidden patterns and associations between the data where it solely takes the inputs without corresponding outputs. The main objective of unsupervised learning is to find hidden relationships in the data, and it is usually utilized for exploratory data mining, for instance, inferring that two items are typically bought together in a supermarket. It is called unsupervised learning because the learning process uses only a set of features without proper labeling that guides the process. Unsupervised learning problems can be divided into clustering and association problems. A clustering problem concerns grouping the data points that are similar to each other, such as grouping students by their background information. However, the association problem is concerned with discovering rules that link large portions of the data points; for instance, students who fail in subject X also tend to fail in subject Y. Many unsupervised learning algorithms have been proposed. We briefly describe some popular clustering algorithms:

K-Means clustering [55]: was proposed by Stuart Lloyd, a clustering algorithm that divides n data points into k clusters. Each data point is assigned to the nearest cluster, which can be calculated by metrics such as the Euclidean distance. Each cluster has a central data point called the centroid, which is basically the mean of all data points belonging to the

cluster. The number of clusters k is allocated manually by the user. The learning procedure usually minimizes within-cluster variances while maximizing between-cluster variance.

Hierarchical clustering [56]: is a clustering algorithm that partitions the data point into clusters with predominant ordering. There are two hierarchical clustering techniques: agglomerative clustering and divisive clustering. Agglomerative clustering is a bottom-up approach that starts by assigning each data point to a cluster. Then, it merges the two similar clusters into one cluster. The algorithm converges when all data points belong to one cluster. Divisive clustering is a top-down approach that starts with assigning all data points into one cluster and then splits the cluster into two least similar clusters.

Principal component analysis (PCA) [57]: is a linear feature learning, a.k.a. feature representation, technique that is used to reduce the dimensionality (i.e., the features) of the data. It is considered unsupervised learning because it takes an unlabeled set of n features and generates a new set of features with a smaller dimension. Transforming the features always comes with a reduced accuracy; however, PCA tries to preserve the essential information in the original features.

2.3 Deep learning

The deep Learning [58] is an emerging branch of machine learning that has recently gained researchers' attention in many areas, such as Natural Language Processing and Computer Vision. Deep learning is built upon the concept of Artificial Neural Network (ANN) [59]. This section describes the basic concepts of ANN and deep learning with examples of deep learning algorithms.

2.3.1 Concepts and terminologies

Deep Learning was introduced by Hinton et al. [60]; they defined deep learning as an Artificial Neural Network (ANN) with multiple hidden layers. These hidden layers provide feature learning where more representative features are learned from the original features. Before presenting the general architecture of a deep learning model, it is essential to understand its principle (i.e., ANN). The concepts of ANN were first proposed by Mcculloch and Pitts [59] inspired by the principle of neurons in biological systems. An (artificial) neural network is a learning algorithm where the function approximation is implicitly in the weights stored in the neurons. Although the calculations used to build a neural network are relatively straightforward, how the neurons are connected is complex. A basic ANN has the following main components [61]: neurons, inputs, outputs, weights, biases, and activation functions.

Neurons – Neural networks are made up of several interconnected neurons, and they are organized into layers. A neuron is the central building unit of a neural network. Each neuron has two significant functions: the transfer function and the activation function. The transfer function is a linear combiner that computes the summation of the products of the inputs and their weights as follows:

$$z = \sum_{i=0}^n w_i x_i + b_i$$

where x is the input vector, w is the weight, and b is the bias. The summation is then passed to an activation function to produce the final output.

Inputs – The inputs of a single neuron consist of 1) the features associated with a weight and 2) the bias. The input layer is built up with several neurons where the number of neurons is equivalent to the number of features.

Outputs – The output layer returns the prediction value y , which is estimated by the transfer function as well as the activation function using the formula:

$$y = f(z)$$

The form of the output depends on the type of activation function, but it can be a real value (regression) or probability (classification).

Weights – The weight (w) is an intrinsic parameter that the neural network model adjusts to amplify or minimize a neuron's input signal. The weights are initialized randomly and then adjusted based on the model performance until it reaches a stopping criterion.

Biases - The bias (b) is a scalar value adjusted throughout the learning. The main purpose of the biases is to add an element of uncertainty to the neural network model. This will allow the model to be generalized and adapted by new (unseen) inputs.

Activation functions – The activation function is a nonlinear function that governs the behavior of the neuron. The most used activation functions are Sigmoid, Softmax, and ReLu. Sigmoid returns a value between 0 and 1; it is usually used for binary classification problems. In a multiclass classification, Softmax can be used, which returns the probability distribution of a data point over all the classes. ReLu gives value ranges from 0 to infinity by simply comparing the input with zero and taking the maximum.

2.3.2 Architectures and Algorithms

Deep learning is an ANN with multiple hidden layers [61]. The multiple layers in a deep learning model allow the model to learn feature transformation and extraction with numerous higher levels of representation [58]. Deep learning can be used in supervised learning (i.e., for regression and classification) or unsupervised learning (e.g., clustering) manners. The behavior of a deep learning model is characterized by its architecture [62], and the architecture is defined by (1) the number of neurons and how they are connected, (2) the number of layers, (3) the learning algorithm used to adjust the weights, and (4) the activation functions. Figure 1 shows the general architecture of a deep learning model. A classical DL model has three types of layers:

Input layer – the input layer passes the preprocessed data points into the model.

Hidden layers – between the input and output layers, there are multiple hidden layers. These layers are responsible for generating a high-level feature abstraction.

Output layer – takes a high-level representation of the features and returns the final prediction results.

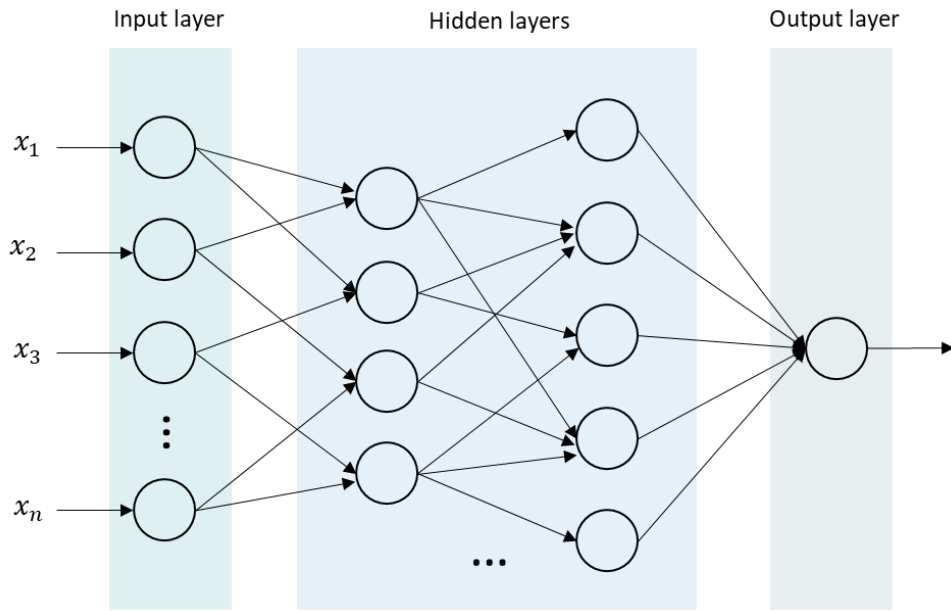


Figure 1 The general structure of DL models

Numerous deep-learning models have been used in the context of software engineering tasks [62]. We briefly describe the most commonly used DL models:

Convolutional Neural Networks (CNNs) [58]: is a feedforward ANN mainly used for image preprocessing and classification. Any CNN has convolutional layers followed by pooling and fully connected layers. Convolution layers filter the inputs and then pass the filtered input to a pooling layer. The pooling layers are used to reduce the dimensionality of the data. The fully connected layer is a multi-layer perceptron neural network (MLP).

Recurrent Neural Networks (RNNs) [63]: is derived from feedforward ANN and usually used for sequential data such as speech recognition and Natural Language Processing (NLP). The structure of an RNN is a directed graph with an internal memory that is used to store the computation of a previous layer. It is called recurrent due to using internal memory to handle a more extended history of computational records that are used recurrently.

Long Short-Term Memory (LSTM) [63]: is a variant of RNN that utilizes (1) a loop to store information of previous computations and (2) gates to control the data passed to the input and the output. This structure allows the model to store long-term information.

Autoencoders (AE) [63]: is an ANN that is usually used for unsupervised learning such as dimensionality reduction. It takes an input and then learns its representation to produce compressed data while maintaining the underlying representations. It consists of an encoder and a decoder. The encoder maps the input into a code (a.k.a. latent space representation) while the decoder takes the code and tries to reconstruct the original input.

2.4 Bots

Bots were first introduced in 1950 by Alan Turing, who proposed that machines could think [64]. There is no universally agreed definition to describe bots; however, they have been defined by researchers and practitioners in terms of their particular context and usage [65]. Bots are software programs that automate repetitive tasks [26]. It has also been defined as a program that performs tasks a human can execute without any human interventions [66]. The recent advancement of artificial intelligence and machine learning have facilitated the deployment of bots in various domains such as education, including students' engagement [67], self-learning [68], and tutoring [69]. They have also been used in other domains, such as in business and e-commerce [70], and in providing customer services [71]. Moreover, bots have been developed as intelligent personal assistants by well-known companies, such as Apple's Siri [72] and Google Assistant [73], which use conversational interfaces to assist users with various personal tasks.

CHAPTER 3

DEEP LEARNING APPROACHES FOR BAD SMELL

DETECTION: A SYSTEMATIC LITERATURE REVIEW

The term “bad smell” is generally used to refer to the presence of problems related to internal design [1] and code [2]. Bad smells negatively impact software quality attributes [4] [3] such as understandability, reusability, and maintainability. A bad smell is not considered a fault or a problem by itself. Rather, it is an indication of a problem stemming from bad practices that may affect software maintenance and evolution [3]. Automatic detection of bad smells will reduce the cost of the manual detection of bad smells in terms of time and resources. Deep learning (DL), which is an emerging area of Machine Learning (ML) that utilizes neural networks, has recently been extensively adopted for research on bad smell detection.

The present SLR focuses on DL detection approaches for design and code smells. Given the rapid growth of DL techniques, we believe that reviewing and analyzing the current body of knowledge would facilitate the development of new techniques and help researchers identify research gaps in this area. DL models are complex compared to traditional ML models; hence, building them requires the configuration of various training hyperparameters that can affect model performance. Previous reviews considered general detection approaches or focused on traditional ML detection approaches. Therefore, a study

that concentrates on DL-based bad smell detection approaches and presents an in-depth analysis of the current state of the DL model performance is necessary.

The main objective of this SLR is to identify and analyze the DL models utilized in bad smell detection. We analyze DL model performance in relation to other aspects, such as the model's purpose, bad smells detected, datasets used for training, feature(s) used in the model, preprocessing technique(s), and encoding technique(s) used for feature transformation. We also consider all design and code smells. Our findings will allow researchers to design and implement DL models based on the most up-to-date information and practices collated from various past investigations. The results will also assist practitioners in making informed selections about the best DL models for their specific context. The following contributions are made by the present SLR to this field.

- A set of 67 studies that address DL-based bad smell detection and were published up to October 2021 are identified; these studies can be used as the foundation for future DL-based bad smell detection research.
- Contextual details of the DL models used in the evaluated studies are summarized: this can enable other researchers to select, design, implement, and evaluate DL models appropriate for their context.
- The detection performance of the current state-of-the-art DL models in bad smell detection is evaluated: this evaluation is based on obtaining and compiling quantitative data (i.e., DL model performance reported in these studies in terms of the precision, recall, and f-score performance metrics).
- Opportunities for future work that would benefit researchers and professionals who plan to work in this research area are identified.

Many studies have used DL for bad smell detection since 2016. However, only a few studies compared the study methodologies available in the literature, evaluated the performance of these methods, and discussed possible limitations and future research directions. [19] presented a systematic literature review (SLR) on code smell detection techniques using ML studies published between 2005 and 2018. However, the main objective of their SLR is different from ours because they focused on traditional ML and did not include papers that used DL. None of the papers included in their SLR were considered in this work. Moreover, their SLR considered papers published only until 2018. More recently, [74] reviewed the application of DL in Code Clone detection. They included 21 studies published until 2020. In comparison, various types of code smells are considered in the present review, and performance analysis is performed from various perspectives; thus, 67 papers published until October 2021 are included herein.

The rest of this chapter is organized as follows: Section 3.1 reviews the existing SLRs and surveys; Section 3.2 describes the methodology used to conduct this SLR; Section 3.3 presents the results of the research questions; Section 3.4 discusses the detection performance analysis; Section 3.5 discusses the implications; Section 3.6 presents threats to validity, and Section 3.7 presents the chapter summary.

3.1 Related Work

This section presents an overview of the previous surveys and SLRs that investigated bad smell detection and the reviews that focused on the use of ML in detecting bad smells. Various smell detection methods have been proposed [15] thus far. Many of these methods suggest detecting code smells using a series of thresholds dependent on structural

parameters (i.e., software metrics) to classify the core symptoms describing specific code smells [16]. Another approach detected code smells by combining multiple structural parameters and formulating rules that vary depending on the code smell form [17]. One major limitation of these methodologies is that their performance heavily depends on the threshold values and/or combination rules. Moreover, no standard threshold values and rules have been set in this regard [18]. Accordingly, other methodologies that use ML and DL approaches have been proposed to mitigate these limitations [19]. Previous reviews investigated automatic code smell detection [75] [76], some of which focused on design smell detection [77] [23].

Recent reviews investigated ML techniques for code smell detection and DL for clone detection. [75] conducted a systematic review focusing on object- and service-oriented systems by identifying and analyzing 78 papers published between 2000 and 2017 to address the evolution of code smells, detection approaches, and research trends. [76] surveyed papers related to code smell detection that were published between 2006 and 2020 and discussed code smells, detection approaches, and automatic and semiautomatic tools for detection and refactoring. Moreover, they presented the challenges faced during code smell detection and suggested improvements to increase detection performance.

[77] reviewed the literature focused on design smell detection. They identified 395 papers between 2000 and 2017 and discussed the design smell types, detection approaches, detection tools, supported programming languages, and validation methods. An SLR proposed by [23] focused on design smell detection, particularly on UML model smells. Their survey covered papers published between 1990 and 2017 and reviewed the detection

techniques, validation methodology, and measurement used to evaluate detection performance.

AbuHassan et al. [15] published an SLR on the approaches used for code and design smell detection. They identified papers published up to 2018 and included 145 studies. Their review focused on the detection techniques, type of design and code smells, validation, and datasets used to evaluate detection performance.

A few SLRs and reviews investigating ML for bad smell detection have recently been proposed. [78] conducted an SLR that reviewed code smells, ML models, independent variables, datasets used, evaluation methods, and detection performance of models. Their study identified and analyzed papers published between 2000 and 2017. They found JRip and Random Forest to be the most efficient models and concluded that the use of ML for bad smell detection is limited, and its detection performance must be improved. They also found that most studies used code metrics as input for ML models.

[79] reviewed code smells and explored the ML models used for bad smell detection along with their detection performance. They identified 14 papers published between 1999 and 2016 and concluded that the genetic algorithm (GA) is the most widely used technique. Their findings show that although the investigated models perform comparably, Decision Tree, K-nearest Neighbor, and Random Forest are among the best performing models.

Al-Shaaby et al. [19] reviewed ML models used for code smell detection. They analyzed 17 papers published between 2005 and 2018 and addressed the ML models used, the most common code smells, performance measures, datasets, and ML tools. They reported that

16 various ML models were used. Of these, J48 and Random Forest performed better than the others.

A more recent review conducted by [80] discussed the use of simple and hybrid ML models for code smell detection in object-oriented systems. Their review included papers published between 2005 and 2020. They addressed the code smell, ML models, performance measures, and datasets used and found that Support Vector Machines and Decision Trees are the most common models used in the reviewed studies.

Another SLR proposed by [81] investigated artificial intelligence techniques used for code smell detection. They addressed the predictors used as input, ML/AI methods, code smell types, datasets used, and performance metrics used to evaluate the models. Their review included 45 papers published between 1998 and 2020, and their results show that most researchers used source code metrics as model input. They also found that Decision Tree and Support Vector Machines are the most widely used.

A recent review by [74] investigated the use of DL for Code Clone detection. Their review covered 21 papers published between 2016 and 2020 but was limited to type 3 and 4 clones. They discussed the limitations, challenges, and recommendations for using DL in Code Clone detection.

In the present SLR, we focus on DL detection approaches and consider both design and code smell. DL models are complex compared to conventional ML models; hence, building them requires the configuration of various training hyperparameters that can affect model performance. Moreover, unlike ML models that are trained using structured data such as numerical data, DL models can be trained using complex types of data such as images,

texts, and a combination of data types. Also, ML models are trained using supervised learning (i.e., a labeled dataset must be provided); however, DL models can be trained using supervised and unsupervised learning. Finally, ML models require an expert to perform feature extraction, while the process is fully automated in DL models due to their ability to learn high-level features from the data. Previous reviews considered general detection approaches or focused on conventional ML detection approaches. Our review concentrates on DL-based bad smell detection and presents an in-depth analysis of the current performance state of DL models.

3.2 Methodology

We followed the systematic method proposed by [82] to review works on DL-based bad smell detection. This section discusses the research questions, search strategy design, selection criteria, quality assessment, and data extraction and synthesis.

3.2.1 Research Questions

This SLR aims to explore DL models used in bad smell detection. Table 1 presents the research questions considered herein.

Table 1 Targeted research questions

Research question	Rationale
RQ 1. What is the current status of deep learning-based bad smell detection? RQ 1.1. What is the distribution of the published research each year?	This RQ presents the distribution of the selected primary studies (PSs) over the years.
RQ 2. Which bad smells have been detected using deep learning? RQ 2.1. Which types of bad smells have been detected using deep learning-based techniques?	In this RQ, we explore the types of bad smells and the software development level at which the bad smells were detected (e.g., code level or model level). Additionally, we

RQ 2.2. What is the granularity of dependent variables that have been used in deep learning-based bad smell detection?	discuss the granularity of the detection, i.e., whether it was detected at a file, class, or method level.
RQ 3. What is the purpose of using deep learning for bad smell detection?	The main purpose of using DL techniques in the primary studies is discussed and categorized to distinguish between problems that have been resolved using DL approaches and those that remain to be resolved. In this RQ, we present the purpose of using DL and the learning approach that was followed by the PSs reviewed.
RQ 4. Which input data have been used in deep learning-based bad smell detection? RQ 4.1. Which features have been extracted and used as input data? RQ 4.2. How were the data used by the PSs preprocessed? RQ 4.3. How are the data encoded into formats that are consumable by deep learning models?	In this research question, we explore the types of features that have been used as inputs for the DL models, how the data have been handled and preprocessed, and how the data were transformed into a format that is consumable by DL models.
RQ 5. Which deep learning models are used for bad smell detection? RQ 5.1. Which deep learning models are used as predictive classification models? RQ 5.2. Which deep learning models are used for representation learning? RQ 5.3. Which training hyperparameters are considered in building deep learning models?	This RQ explores the DL models that have been used for bad smell detection in primary studies. We discuss the DL models used as predictive classification and the ones used for representation learning. Moreover, we discuss the training hyperparameters that have been considered for training the DL models.
RQ 6. How are deep learning models evaluated and implemented in bad smell detection? RQ 6.1. Which validation methods are used to evaluate the deep learning models? RQ 6.2. Which evaluation metrics have been more frequently used to report the accuracy of deep learning models? RQ 6.3. Which baseline techniques are used to evaluate the deep learning models? RQ 6.4. Is the source code of deep learning models publicly available?	This RQ explores how the DL models were evaluated, including the validation methods used to validate the models, the evaluation metrics considered, and the baseline approaches used to compare the proposed DL models. Additionally, we discuss the availability of the source code.
RQ 7. How are the bad smell datasets being constructed and used? RQ 7.1. Which systems and datasets are used in deep learning-based bad smell detection? RQ 7.2. Which programming languages are most frequently used to construct datasets? RQ 7.3. Which approaches have been used to create the dataset?	In this RQ, we focus on the datasets used in the selected studies. The datasets used in training DL models for bad smell detection are constructed using various systems, written in different programming languages, and of different sizes. Moreover, several methods have been used to create the labels (i.e., the label), and many tools were used to extract features. The ratio of the publicly available datasets is highlighted.
RQ 8. What is the current performance of bad smell detection using deep learning models?	This RQ shows the detection performance of the DL models with respect to the dataset, bad smells, DL models, encoding technique, preprocessing technique, and the features used.

3.2.2 Study Selection

A paper was selected if and only if it fulfilled the following inclusion criteria.

- IC1: used a DL model to detect bad smells.
- IC2: reported empirical results.
- IC3: a research paper that was peer-reviewed and published in a conference, journal, or workshop.

The paper that fulfilled at least one of the following exclusion criteria was excluded.

- EC1: not written in English.
- EC2: book chapters, work in progress, abstract, Master’s thesis, or Ph.D. dissertations.
- EC3: review study (e.g., SLR, survey, etc.).
- EC4: conference papers extended to a journal article of the same paper.

3.2.3 Search Strategy Design

We identified relevant studies by selecting six digital libraries—IEEE Xplore, ACM Digital Library, Scopus, Springer Link, Science Direct, and Wiley. The search was performed in October 2021, with the inclusion of studies published up to October 2021. We also considered studies available as “early access” in the given libraries. We formulated the search string by identifying the common keywords found in the research questions that were relevant to the focus of this SLR. We focused herein on DL-based bad smell detection; thus, we identified the synonyms of bad smells, detection, and DL techniques. A pilot search performed in all specified digital libraries revised the search string to the following.

- Bad smells: (“model smell*” OR “design smell*” OR “bad smell*” OR “code smell*” OR “design flaw*” OR “antipattern*” OR “code clone*” OR “misuse case*”) AND
- Detection: (“detect*” OR “predict*” OR “identif*”) AND
- DL Technique: (“deep learning” OR “transfer learning” OR “representation learning” OR “CNN” OR “RNN” OR “auto-encoder*” OR “convolutional neural network*” OR “Deep neural network*” OR “Recurrent neural network*”) AND Software

A total of 946 potential studies were obtained by applying the search string (Figure 2). Note that the search string was modified according to the search restriction in each library, and filters were used when necessary. The exact search string used in each digital library is provided online². We found approximately 2478 results on the Springer link; of these, most were irrelevant. Therefore, we filtered the results to computer science only, narrowing down the search results to 547 studies. We reduced the number of search terms used in ScienceDirect because it supports only eight (AND/OR) operators. Moreover, the asterisk (*) wildcard was limited to seven; thus, we changed the search terms accordingly.

The scanning process was conducted by the first author (a Ph.D. student). The inclusion and exclusion criteria were designed by all authors. Pilot papers were used to apply the criteria and discuss them. The search process followed for identifying relevant studies was as follows.

²<https://github.com/amalazba/Deep-Learning-Approaches-for-Bad-Smell-Detection-SLR>

1. The search string was used to select relevant studies from the listed libraries. A total of 946 potential studies were obtained.
2. The inclusion/exclusion criteria were applied against the titles and the abstracts. A total of 846 studies were excluded. The duplicates were then removed (38 studies). The inclusion/exclusion criteria (IC/EC) were then applied to the whole paper. Twelve studies were excluded by applying the following filters: IC1 application excluded nine studies, IC2 application excluded one study, and EC4 application excluded two studies.
3. Next, we performed forward snowballing at one level considering all citations from the 50 relevant studies using the *Publish or Perish* tool [83]. Approximately 1000 papers were thus retrieved. Through the snowballing procedure, an additional 17 relevant studies that met our inclusion criteria were identified.
4. Finally, we ended up with 67 primary studies evaluated through quality assessment. Appendix A presents the primary studies considered in this SLR following quality assessment with venues and the publication type. All papers with a quality assessment score less than the threshold were excluded from the selected studies. A total of 67 papers that passed the third round of selection were considered in the final set and used in the data extraction and synthesis.

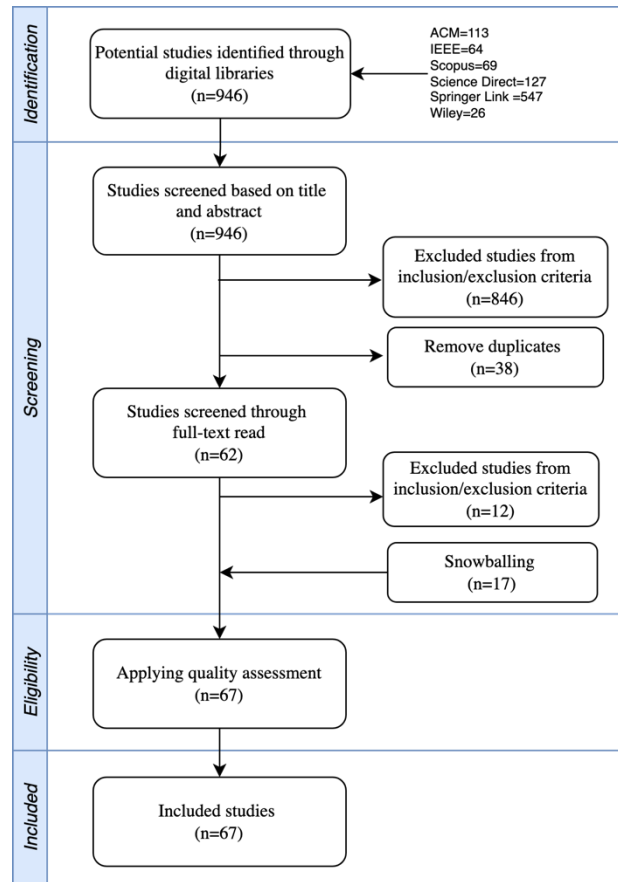


Figure 2 Search process

3.2.4 Quality Assessment

Assessing the quality of the selected studies is essential. Quality assessment is useful in interpreting main research data and assessing the extracted data quality. The quality evaluation substantially depends on the SLR type. There are no uniform definitions of major study quality metrics [84]. In this review, we considered different quality criteria, including clarity of research questions, performance measures, independent/dependent variables, validation method, datasets, experiment details, and threats to validity. The quality assessment application in this work mainly aims to assist in the selection of primary studies by assessing the extracted data quality (i.e., studies with sufficient data that could answer the formulated research questions).

Table 2 shows the quality assessment criteria used to assess the studies considered in this review, wherein the maximum score was 7. We specified a threshold of 50% (i.e., any study that scored less than 3.5 on the quality assessment was excluded), as used by [85]. The quality assessment was performed by one author, and all of the studies were verified by the other two authors. Conflicts in the quality assessment were discussed and resolved by the authors. No paper was excluded after the assessment. Detailed quality assessment scores are provided online³.

Table 2 Quality assessment checklist

No.	Quality assessment checklist	Score
QA1	Are the aims and questions of the research clearly defined?	(+1) Yes/ (+0) No
QA2	Are the performance measures used to assess the models in the study specified?	(+1) Yes/ (+0) No
QA3	Are the independent variables and dependent variable(s) clearly defined?	(+1) Yes/ (+0) No
QA4	Is the validation method specified?	(+1) Yes/ (+0) No
QA5	Are the datasets adequately described?	(+1) Yes/ (+0) No

³<https://github.com/amalazba/Deep-Learning-Approaches-for-Bad-Smell-Detection-SLR>

QA6	Are the details of the experiment method adequately described?	(+1) Yes/ (+0) No
QA7	Are the threats to the validity of the study specified?	(+1) Yes/ (+0) No

3.2.5 Data Extraction

We designed a data extraction form using Excel sheets to collect the data needed to answer the research questions. One of the authors was involved in the data extraction process. Accordingly, all studies were verified by the other two authors to ensure the correctness of the extracted data. Any conflicts in the extracted data were discussed and resolved. The data extraction form was structured in six sections: (1) metadata; (2) bad smells; (3) DL, (4) training considerations, (5) evaluation, and (6) datasets.

Table 3 provides a detailed description of the fields of each section in the data extraction form. This form was used to extract data from the final set of the collated primary studies.

Table 3 Data extraction form

Extracted data	Description	RQ
1. Metadata	Study ID, Title, Authors, Publication year, Publication type, Publication venue, and Summary	RQ1
2. Bad smells	Type (Code Clone, God Class...etc.), Level (code, model), and Granularity (class, method, file...etc.)	RQ2
3. Deep learning	Purpose (the goal of using DL), Features, Learning approach (supervised, unsupervised...etc.), Preprocessing (sampling, feature selection...etc.), Encoding (word2vec, code2vec...etc.), and DL models	RQ3, RQ4, RQ5.1, RQ5.2
4. Training considerations	Optimization algorithm, Loss function, Initialization, Learning rate, and Overfitting-handling	RQ5.3
5. Evaluation	Validation method, Evaluation metrics, Development environment, Source code availability, Code link, Baseline, and Model performance	RQ6, RQ8
6. Datasets	Systems/datasets, Programing language, Tools, Labeling method, Labeling tool, DS Availability, and DS link	RQ7

3.2.6 Data Synthesis

The data extracted in this SLR contained both qualitative and quantitative data (i.e., model performance). For the qualitative data, data tabulation was suitable for the collection,

comparison, and synthesis of information, as prescribed in the literature [86]. The data were extracted and stored in an Excel file. We then implemented an automated script using Python to compile the data and create the tables required to answer each research question.

3.3 Results

This section presents the results of this SLR to answer the research questions defined in Section 3.2.1. For each research question, the results are supported with opportunities for future work and open issues are discussed.

3.3.1 What is the current status of deep learning-based bad smell detection?

What is the distribution of the published research each year?

The selected PS years of publication ranged from 2016 to 2021. Figure 3 shows the distribution of papers published during these years. The last three years have witnessed a remarkable growth in the number of papers that used DL approaches for bad smell detection. The majority of PSs (85%) were published between 2019 and 2021; on average, approximately 19 papers per year have been published, demonstrating that the use of DL for bad smell detection is currently an active research area. White et al. [PS28] performed preliminary work in 2016. Between 2016 and 2017, DL was only applied to Code Clone smells. In 2018, other bad smells (e.g., God Class, Data Class, Feature Envy, and Long Method) were detected using DL.

DL requires a vast amount of data. Since 2016, more data on various bad smells have become publicly available. This explains why DL has been applied in this field fairly recently despite being successfully and widely used in other fields since a long time [87].

In 2014, [88] provided a benchmark open-access dataset for Code Clone, the BigCloneBench dataset. Further, [89] and [90] provided open-access datasets for various bad smells. These datasets have motivated researchers to investigate DL approaches using them.

RQ1.1 summary. The use of DL for bad smell detection was introduced in 2016. Few studies were conducted between 2016 and 2018. Between 2019 and 2021, considerable research interest has been extended to the use of DL approaches for bad smell detection. The Code Clone smell was the first smell investigated using deep learning. Other smell types have been explored since 2018.

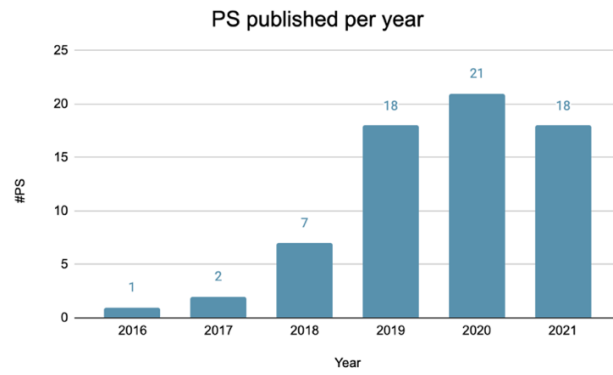


Figure 3 Distribution of primary studies over the year

3.3.2 Which bad smells have been detected using deep learning?

Which types of bad smells have been detected using deep learning-based techniques?

We categorized bad smells into two types based on the terms used by the authors in the PSs to describe the bad smells they are exploring. The authors in two papers referred to the smells as an “implementation smell” [PS23, PS53], whereas those in another paper called the investigated smells a “linguistic smell” [PS48]. We categorized them as code smells and design smells based on the following two definitions.

1. Code smells [91]: poor implementation choices of the source code that breach important principles, thereby negatively affecting understandability, maintainability, and reusability.
2. Design smells [92]: structures in the design that suggest a violation of the primary design principles, which might negatively affect design quality.

Figure 4 shows that code smells were investigated in 63 PSs, whereas design smells were investigated in 10 PSs. Figure 5 shows the type of bad smells detected using DL (a list of PSs is presented in Appendix B). The authors of the PSs used different smell names for the same problem (e.g., Blob and God Class). We decided to use the same names in this work as used in the original PSs. Among the various code smells studied, Code Clone is at the top of the list—it was studied in 58 papers out of the selected 67 PSs. The Feature Envy and Long Method code smells are next on the list, with the former being studied in nine PSs and the latter in five PSs. God Class was the most investigated design smell having been explored in four of the PSs. Blob, Deficient Encapsulation, Insufficient

Modularization, and Multifaceted Abstraction design smells were investigated in two of the PSs.

The Code Clone prevalence might be attributed to the following reasons. 1) Availability of benchmark datasets with millions of labeled instances. 2) The cost of dataset construction being less expensive than that for other smells; some studies (e.g., [PS8, PS10, PS24, PS52]) considered two code fragments that solve the same problem as clone and nonclone instances. 3) Availability of tools that can be used to find clones in the source code [18]. 4) Detection of such bad smells across multiple programming languages (e.g., [PS11, PS18, PS24, PS35, PS45]).

The second most recurring smells (i.e., Feature Envy, Long Method, and God Class) may have attracted research attention for the following reasons.

- 1) Publicly available datasets [89], [90].
- 2) The existence of tools that can be used to generate labels, such as JDeodorant [93], iPlasma [94], and DÉCOR [17]. The analysis of this research question shows that further studies must focus on code smell types other than the three specified above and further investigate design smells.

Detecting bad smells in the early stages of the software development cycle has many advantages, including reduction in cost and effort. Nevertheless, bad smell detection at the model level is more difficult than at the code level [23]. This explains the lack of studies investigating bad smell detection at the model level. Moreover, using DL requires a large number of labeled datasets, which are not widely available [19]. This also contributes to the scarcity of such studies at the model level. Future research should thus be directed

toward creating benchmark datasets and exploring DL for bad smell detection at the model level.

Many DL models have been used to detect various bad smells; however, some of them were used to detect a specific type of bad smell. For instance, CNN and LSTM have been extensively utilized to detect Code Clone, Feature Envy, and God Class smells, while autoencoders (i.e., AE and VAE) and feedforward neural networks (i.e., DNN and ANN) have been used to detect Long Method and Large Class smells. The prevalence of the CNN and LSTM might be due to their ability to extract deep semantical relationships. CNN can learn the local features of the input data, while LSTM can learn dependencies among a sequential input. All three smells (i.e., Code Clone, Feature Envy, and God Class) require semantics information for detection. Moreover, Long Method and Large Class depend on the lexical and structural information that can be detected with DL models simpler than CNN and LSTM, such as autoencoders (i.e., AE and VAE) and feedforward neural networks (i.e., DNN and ANN). A final observation is that although CNN and LSTM can extract semantics, CNN has been more popular. One possible reason for this is the training time, whereby CNN is well-suited for parallel computation, which reduces the training time.

RQ2.1 summary. We discovered that some bad smells received more research attention in the scientific literature than others. Bad smells under the code smell category are the most investigated smells. Among them, Code Clone, Feature Envy, and Long Method are the most recurring smells. The other most common smells are of the design smell category, with God Class at the top of the list. Further studies with more focus on other types of smell

are needed. Only few studies have explored design smell detection. Hence, exploring DL for detecting bad smells at the model level is necessary.

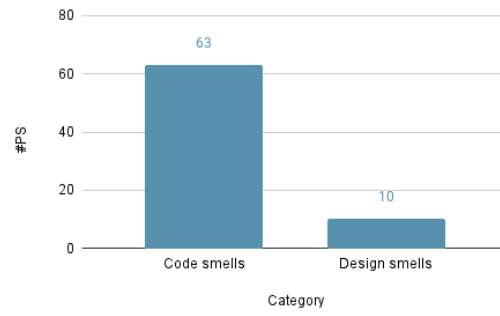


Figure 4 Bad smell categories

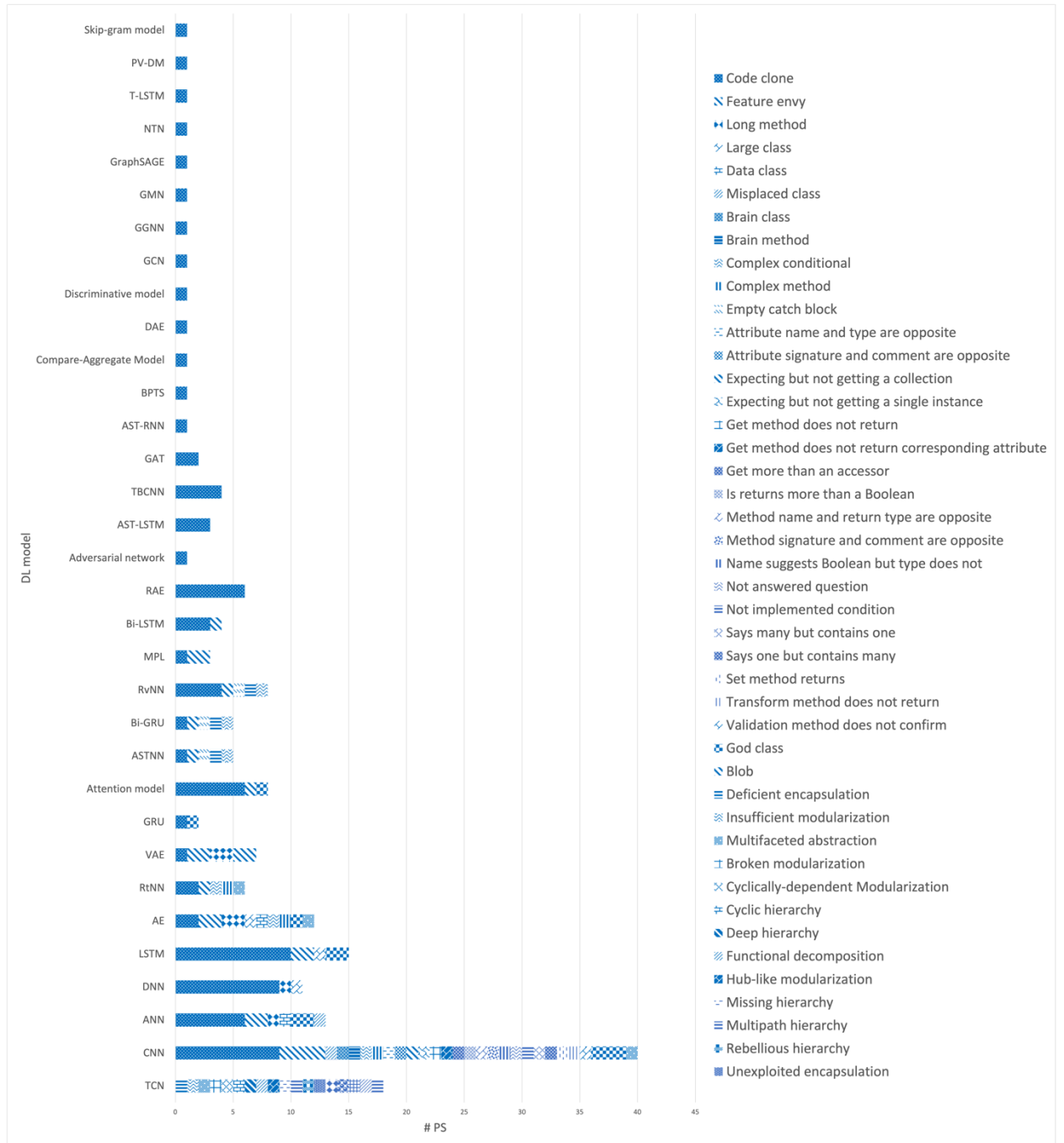


Figure 5 Types of bad smells detected using DL

What is the granularity of dependent variables that have been used in deep learning-based bad smell detection?

Each bad smell type can be detected on a relative scale or level of detail (i.e., granularity). Six granularities can be found in the PSs: attribute, method, class, file, project, and code fragment. An attribute is a subset of a method or a class, a method is a subset of a class, a class is a subset of a file, and a file is a subset of a project (i.e., an application). Code fragments refer to pieces of code that occur in a program. They can be a method or part of a method. They can also mean a class or part of a class.

Table 4 presents the granularities with the total number of primary studies. Most of the studies (62% of the PSs) detected bad smells at the method level. Next, 18% and 9% of the PSs detected bad smells at the class and file levels, respectively. The code fragment, project, and attribute levels were the least investigated granularities in the literature, with only 5%, 4%, and 1%, respectively, of the PSs having detected bad smells at these levels. Finally, one study detected clones at multiple granularity levels, but this was not specified in the original paper.

Code Clone and Feature Envy are the only two bad smells detected at various granularity levels. Code Clone was detected at all levels except for the attribute level, whereas Feature Envy was detected at the class and method levels. An obvious reason why the remaining types of bad smells were detected at a single level is that, by definition, some bad smells can appear at one granularity level. For instance, the *Attribute name and type are opposite* is an attribute-level bad smell that cannot be detected at other granularities. Finally, we

notice that no obvious relationship existed between the granularity and the selected DL models, wherein CNN was the most used at the method, class, file, and attribute levels.

RQ2.2 summary. Most studies (i.e., 62% of the PSs) detected bad smells at the method level. Next, 18% and 9% of the PSs detected bad smells at the class and file levels, respectively. The code fragment, project, and attribute levels were the least investigated granularities in the literature, with only 5%, 4%, and 1% of the PSs having detected bad smells at these levels, respectively. Future studies should investigate detection performance at various granularity levels other than the method level.

Table 4 Granularity of dependent variables

Granularity	DL models*	#PS
Method	CNN (14): [PS31, PS34, PS23, PS27, PS3, PS48, PS37, PS15, PS2, PS21, PS50, PS12, PS57, PS10], DNN (9): [PS27, PS60, PS45, PS16, PS18, PS65, PS36, PS17, PS55], LSTM (9): [PS31, PS27, PS51, PS1, PS9, PS50, PS24, PS40, PS49], Attention model (7): [PS51, PS40, PS1, PS43, PS50, PS67, PS47], ANN (6): [PS4, PS41, PS40, PS66, PS44, PS24, PS32], RAE (6): [PS60, PS29, PS16, PS42, PS39, PS20], AE (4): [PS5, PS4, PS28, PS23], TBCNN (4): [PS14, PS47, PS66, PS54], AST-LSTM (3): [PS40, PS62, PS56], Bi-LSTM (3): [PS1, PS41, PS13], GCN (3): [PS67, PS40, PS42], MLP (3): [PS31, PS50, PS51], RvNN (3): [PS9, PS53, PS20], ASTNN (2): [PS9, PS53], Bi-GRU (2): [PS9, PS53], RtNN (2): [PS28, PS23], VAE (2): [PS30, PS46], Adversarial network (1): [PS56], AST-RNN (1): [PS49], BPTS (1): [PS28], Compare-Aggregate Model (1): [PS13], DAE (1): [PS57], Discriminative model (1): [PS1], GAT (1): [PS51], GGNN (1): [PS33], GMN (1): [PS33], GraphSAGE (1): [PS43], GRU (1): [PS58], NTN (1): [PS43], RvNN (1): [PS28], Skip-gram model (1): [PS63]	53
Class	CNN (6): [PS26, PS38, PS25, PS34, PS23, PS27], AE (3): [PS5, PS4, PS23], LSTM (3): [PS38, PS25, PS27], RvNN (1): [PS53], ANN (3): [PS4, PS38, PS6], ASTNN (1): [PS53], Attention model (1): [PS38], Bi-GRU (1): [PS53], DNN (1): [PS27], GGNN (1): [PS33], GMN (1): [PS33], GRU (1): [PS25], RAE (1): [PS29], RtNN (1): [PS23], TCN (1): [PS64], VAE (2): [PS30, PS46]	15
File	TBCNN (3): [PS14, PS47, PS54], LSTM (2): [PS1, PS52], AST-LSTM (2): [PS62, PS56], Attention model (2): [PS1, PS47], RvNN (2): [PS28, PS52], T-LSTM (1): [PS52], Bi-LSTM (1): [PS1], Discriminative model (1): [PS1], Adversarial network (1): [PS56], RtNN (1): [PS28], BPTS (1): [PS28], AE (1): [PS28]	8
Code fragment	ANN (1): [PS7], Bi-LSTM (1): [PS35], DNN (1): [PS7], CNN (1): [PS22], VAE (1): [PS35], GAT (1): [PS59]	4
Project	Bi-LSTM (1): [PS61], PV-DM (1): [PS11], RtNN (1): [PS19]	3
Attribute	CNN (1): [PS48]	1
Not specified	Bi-LSTM (1): [PS8]	1

*A PS may belong to one or more fields.

3.3.3 What is the purpose of using deep learning for bad smell detection?

We address this research question by manually identifying the purpose of using DL in primary studies. These aims were not predetermined. They were derived from a careful reading of how each study implemented DL for bad smell detection. We identified the following three purposes.

1. **Predictive classification:** In DL, the models comprise many layers. The first few layers try to learn a useful representation of the features (implicitly) from the input. The last few layers then transform the last representation and output the predictions [63]. The main goal of using DL in the PSs under this category is to predict the bad smell classification. The output of the DL models is usually in the form of a binary value (i.e., 0 or 1) representing the presence or absence of a smell. The inputs are fed to the DL model, which directly outputs the classification results.
2. **Explicit representation learning:** Representation learning is described as the process of extracting useful information by learning an efficient and informative representation to improve the classification or regression model performance [95]. In this category, DL is used to transform the input into a useful representation, wherein it takes the original features as an input and outputs a feature vector. The new feature vector is then used to predict the classification using a specific measure, an algorithm, or conventional ML models that give a classification value for each bad smell. Hence, the main goal of using the DL model for bad smell detection in the PSs belonging to this category is to learn relevant input data representations rather than making predictions.

3. **Predictive classification and explicit representations:** Two different DL models can be used in this category. The first model is used to learn the representation. It outputs a feature vector, which is then fed into another DL model to make predictions. Note that more than one DL model can be used to learn the representation. The feature vectors are then concatenated and fed to the predictive model. For this category, we use the name classification and representation for conciseness in the rest of the chapter.

Table 5 shows the three main purposes of using the DL models for bad smell detection. DL models are mostly used for explicit representation learning (i.e., 43% corresponds to 29 papers). Utilizing DL as a predictive classification comes next, with 33% (22 papers) of the PSs falling under this aim. Bad smell detection using DL for representation learning and predictive classification is the least explored purpose, including only 24% or 16 papers. The studies that used DL models exclusively for explicit representation used other methods to make predictions. Some PSs used similarity measures such as cosine similarity [PS10, PS14, PS22, PS33, PS35, PS37, PS42, PS45, PS47, PS52, PS54, PS58, PS59], Euclidean distances [PS39, PS63], and Manhattan distance [PS61]. Some algorithms were also considered, including the weighted L1 distance algorithm [PS3], locality sensitive hashing [PS20, PS63], near-neighbor querying algorithm [PS20], and dynamic time warping [PS42]. Finally, some PSs implemented ML classifiers, such as K-Nearest Neighbors [PS11], Random Forest [PS29], Logistic Regression [PS30, PS46, PS67], and Multilayer Perceptron (MLP) [PS31, PS40, PS50, PS51] to make predictions.

The prevalence of using DL in bad smell detection for representation learning may be attributed to the ability of selecting an appropriate DL model to represent various input

formats. For example, in [PS31], they used two DL models for representation learning, LSTM for textual information and CNN for software metrics. Another example is [PS28], where RtNN and RvNN were utilized to learn useful representations from the source code and abstract syntax tree (AST), respectively. Their DL model selection was based on the fact that RtNN can model sequences of terms, such as the source code, whereas RvNN is well suited to model structures such as AST. Another possible reason behind the immense research interest in representation learning is the ability to learn a representation without supervision (i.e., the dataset does not have to be labeled), which will be discussed in the following subsection. Using DL in bad smell detection for representation learning and predictive classification has not been widely considered. An explanation for this might be the time and space complexity required to build multiple DL models [96]. This category lacks studies; hence, future research could focus on this purpose to show the tradeoffs between using DL models for representation and classification vs. using the DL model solely for either representation or classification.

Table 5 Main purpose of using DL in the PSs

Purpose	PS	#PS
Explicit representation	[PS63, PS47, PS33, PS35, PS52, PS67, PS11, PS45, PS59, PS29, PS40, PS39, PS22, PS20, PS54, PS10, PS58, PS3, PS61, PS46, PS14, PS62, PS5, PS30, PS37, PS42]	26
Predictive classification	[PS18, PS23, PS34, PS65, PS55, PS15, PS64, PS7, PS49, PS27, PS2, PS17, PS25, PS26, PS44, PS6, PS8, PS12, PS19, PS48, PS21, PS36]	22
Predictive classification and explicit representation	[PS43, PS32, PS24, PS41, PS56, PS31, PS38, PS51, PS1, PS16, PS60, PS57, PS50, PS53, PS66, PS13, PS9, PS28, PS4]	19

The learning approaches of ML are grouped into many categories depending on the nature of learning. We classified the PSs based on four learning approaches: supervised, unsupervised, semi-supervised, and self-supervised. Categorization was predetermined

because they are well-known and broadly used. This classification is necessary because it will ease the identification of the nature of the problem, examine the resources, and devise a viable solution. Each learning approach is described below.

1. **Supervised** [21]: the process of learning a mapping function that maps between the input and the output. The input comprises a set of features, whereas the output comprises labels assigned to the inputs. The outputs (i.e., labels) are usually manually assigned.
2. **Unsupervised** [54]: the process of discovering hidden patterns and associations between the data, where it solely takes the inputs without corresponding outputs. It is called unsupervised learning because the learning process uses only a set of features without the proper labeling that guides the process.
3. **Semi-supervised** [63]: a learning approach that takes the benefits of supervised and unsupervised learning. It utilizes both labeled and unlabeled data to train the model, in which a small amount of labeled data with a large amount of unlabeled data during training are used.
4. **Self-supervised** [22]: framing unsupervised learning problems into supervised problems and then implementing a supervised learning technique to solve them. Supervised learning techniques are used to solve a pretext task formulated from the unlabeled data. The resulting model from solving the pretext task is then used in the solution of the original (actual) modeling problem. After the training process, some labeled data are used to merely fine-tune the model.

Table 6 shows the learning approaches used by the selected PSs. Supervised learning was the most adopted learning approach in DL-based bad smell detection, representing 75%

(50 papers) of the selected studies. This is followed by semi-supervised and unsupervised learning, with 13% (nine papers) and 10% (seven papers) of the total PSs having adopted these methods, respectively. Self-supervised learning was rarely used. Only 2% (one paper) of the selected studies followed this learning approach. Supervised learning was extensively used because bad smell detection is considered a binary classification problem usually tackled using this approach. However, the lack of labeled datasets urged the research community to explore alternative learning approaches. DL techniques also facilitated the adaptation of other learning approaches [97]. Note that all studies utilizing the self-supervised or unsupervised learning approaches focused on detecting the Code Clone smell. In addition, 77% (seven out of nine papers) of the PSs that used a semi-supervised learning approach were also for Code Clone detection. Therefore, we recommend future studies to focus on detecting bad smells other than Code Clone and use learning approaches other than supervised learning.

One other remark is that the learning approaches are somehow related to the purpose of DL usage. Figure 6 illustrates the purpose of using DL in each learning approach. The figure clearly shows that all studies that used DL for predictive classification have followed the supervised learning approach. Conversely, in unsupervised and self-supervised learning, the DL models were mainly used for representation learning. Moreover, semi-supervised learning usually uses DL models for classification and representation. This outcome exposes a serious gap because detecting bad smells using learning approaches other than supervised learning for predictive classification should attract more research attention.

Table 7 presents the various DL models used by each learning approach. CNN and LSTM were frequently used for supervised learning, whereas ANN was mostly used in semi-supervised learning. RAE was used in unsupervised learning, while the TBCNN and the Attention model were used in self-supervised learning.

Table 6 Learning approaches explored in the PSs

Learning approach	PS	#PS
Supervised	Predictive classification : [PS25, PS48, PS8, PS7, PS19, PS26, PS2, PS34, PS44, PS15, PS12, PS21, PS27, PS6, PS36, PS49, PS17, PS23, PS64, PS18, PS55, PS65], Classification and Representations: [PS1, PS28, PS53, PS43, PS41, PS66, PS13, PS31, PS51, PS50], Explicit representations : [PS59, PS40, PS67, PS3, PS14, PS54, PS37, PS33, PS52, PS30, PS62, PS29, PS58, PS63, PS46, PS61, PS35, PS45]	50
Semi-supervised	Classification and Representation: [PS9, PS38, PS16, PS56, PS32, PS57, PS24, PS4], Explicit representations: [PS5]	9
Unsupervised	Classification and Representation: [PS60], Explicit representation: [PS39, PS42, PS22, PS11, PS10, PS20]	7
Self-supervised	Explicit representations: [PS47]	1

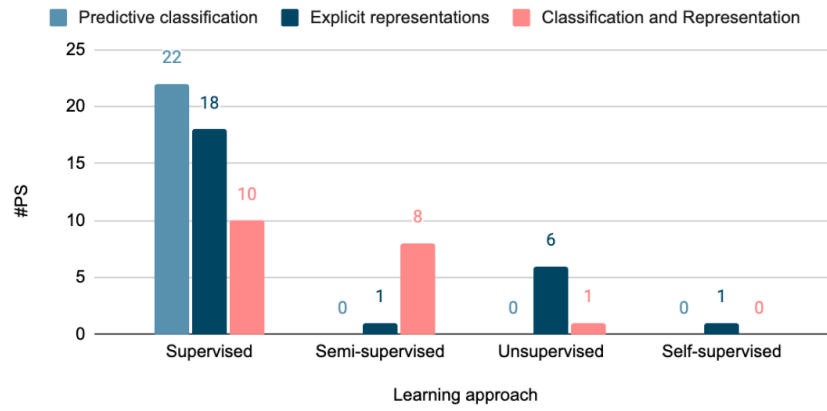


Figure 6 Learning approach vs. purpose of DL usage

Table 7 DL models used by learning approaches

Learning approach	DL models
Supervised	CNN (14): [PS3, PS25, PS48, PS37, PS15, PS2, PS21, PS31, PS50, PS23, PS12, PS26, PS27, PS34], LSTM (9): [PS51, PS25, PS52, PS1, PS31, PS50, PS27, PS40, PS49], DNN (8): [PS45, PS7, PS18, PS27, PS65, PS36, PS17, PS55], ANN (6): [PS41, PS7, PS66, PS44, PS40, PS6], Bi-LSTM (6): [PS41, PS61, PS1, PS35, PS13, PS8], Attention model (6): [PS1, PS43, PS50, PS67, PS40, PS51], MLP (3): [PS31, PS50, PS51], RNN (3): [PS19, PS28, PS23], RvNN (3): [PS28, PS53, PS52], VAE (3): [PS30, PS46, PS35], TBCNN (3): [PS14, PS54, PS66], AE (2): [PS28, PS23], AST-LSTM (2): [PS40, PS62], GAT (2): [PS51, PS59], GCN (2): [PS67, PS40], GRU (2): [PS25, PS58], AST-RNN (1): [PS49], ASTNN (1): [PS53], BPTS (1): [PS28], Bi-GRU (1): [PS53], GGNN (1): [PS33], GMN (1): [PS33], GraphSAGE (1): [PS43], NTN (1): [PS43], RAE (1): [PS29], T-LSTM (1): [PS52], TCN (1): [PS64], Skip-gram model (1): [PS63], Discriminative model (1): [PS1], Compare-Aggregate Model (1): [PS13]
Semi-supervised	ANN (4): [PS4, PS38, PS32, PS24], LSTM (3): [PS9, PS38, PS24], AE (2): [PS5, PS4], CNN (2): [PS38, PS57], AST-LSTM (1): [PS56], ASTNN (1): [PS9], Bi-GRU (1): [PS9], DAE (1): [PS57], DNN (1): [PS16], RAE (1): [PS16], RvNN (1): [PS9], Attention model (1): [PS38], Adversarial network (1): [PS56]
Unsupervised	RAE (4): [PS60, PS39, PS42, PS20], CNN (2): [PS22, PS10], DNN (1): [PS60], GCN (1): [PS42], PV-DM (1): [PS11], RvNN (1): [PS20]
Self-supervised	TBCNN (1): [PS47], Attention model (1): [PS47]

RQ3 summary. We identified three purposes of using DL for bad smell detection, one of which was more frequent than the others. The top aim is using DL for explicit representation learning. The DL model output is typically a feature vector. Utilizing DL models as predictive classifiers is the second most expressed purpose. Finally, using DL for both aims ranked last as a purpose. Future research should focus on showing the tradeoffs between using DL models for representation and classification vs. using them exclusively for either representation or classification. Supervised learning was the most adopted learning approach in DL-based bad smell detection, representing 75% (50 papers) of the selected PSs. Semi-supervised and unsupervised learning followed, with 13% (nine papers) and 10% (seven papers) of the considered PSs having used them, respectively. Self-supervised learning was rarely used, with only 2% (one paper) of the selected studies

following this learning approach. The outcome exposes a serious gap considering that bad smell detection using learning approaches other than supervised learning for predictive classification should attract more research attention.

3.3.4 Which input data have been used in deep learning-based bad smell detection?

Which features have been extracted and used as input data?

Various features have been utilized for bad smell detection using DL models. Figure 7 and Table 8 summarize the feature types used in the selected primary studies. The use of graph-based features was largely explored, and we found that 50% of the PSs used at least one graph type. The most frequently used graph representation is the AST. Almost one-third of the PSs (33%) have employed the AST as an input. The control flow graph (CFG) is the second most frequently used graph representation in the PSs, representing 11%.

Text- (24%) and numeric-based (20%) features rank second after the graph type. Text-based features mainly include the source code and the textual information extracted from the source code or other references. Some examples of textual information are the identifiers name [PS29], method and class names [PS31, PS41], comments [PS48], and text from the API documentation [PS11]. Numerical features include various software metrics calculated from the code [PS38, PS18, PS41, PS12, PS64, PS4, PS27, PS26, PS31, PS46, PS34, PS5, PS55, PS25, PS7, PS65] or from the model [PS6]. Binary code and images are rarely used as inputs in DL-based bad smell detection. In fact, only 4 and 1% of the PSs employed binary code and images, respectively.

We observed that the majority of papers (73% PSs) utilized only one feature, while the rest (27% PSs) used a combination of at least two features [PS7, PS8, PS12, PS15, PS25, PS27, PS28, PS29, PS31, PS32, PS35, PS38, PS39, PS40, PS41, PS44, PS46, PS50]. Using multiple features is highly recommended because various features capture different information types. For instance, the AST can capture the structural features of the source code, but textual information is needed to capture the semantics information. Combining multiple features is effective in increasing the DL model performance compared to relying solely on one feature [PS25]. Therefore, we highly recommend that future studies explore the use of multiple features for bad smell detection. Future research might also examine the feature type well suited for each smell type considering that some bad smells can be detected on the structure level (e.g., Long Method), while others need semantics information (e.g., Feature Envy).

Among the utilized DL models, the CNN and the LSTM are the most frequently used on graph-based features. The LSTM was frequently used for text-based features, while the CNN was mostly used for numeric- and image-based features. Finally, the RAE was used for binary code features. The feature type must be considered when selecting DL models because each model has characteristics better suited for a specific type of data. For instance, the CNN is useful with data that have a spatial relationship like images and speech, whereas the LSTM is designed to work with sequential data, such as text. The graph neural network (GNN) is the most suitable model for graph-based features like AST, CFG, and PDG; however, only a very few studies have considered using it (GGNN: [PS33], GraphSAGE: [PS43], GMN: [PS33]) because it is still evolving. We recommend that future research to

explore and develop GNN-based DL modes (e.g., gated graph sequence neural networks [98]) to better represent the graph-based features.

RQ4.1 summary. The use of graph-based features was largely explored. We found that 50% of the PSs used at least one graph type. The text- (24%) and numeric-based (20%) features ranked second after the graph-based features. Binary code and images were rarely used as inputs in DL-based bad smell detection. Only 4 and 1% of the PSs employed binary code and images, respectively. Future research might explore the feature type well suited for each smell type considering that some bad smells can be detected on the structure level (e.g., Long Method), while others need semantics information (e.g., Feature Envy).

Table 8 Features used in DL-based bad smell detection

Type	#PS	Features	PS *	#PS
Graph	43	Abstract syntax trees (AST)	CNN (6): [PS12, PS15, PS2, PS37, PS3, PS57], TBCNN (4): [PS66, PS47, PS54, PS14], ANN (4): [PS66, PS44, PS40, PS24], RAE (4): [PS39, PS16, PS60, PS29], LSTM (3): [PS49, PS52, PS24], AST-LSTM (3): [PS62, PS56, PS40], Attention model (3): [PS47, PS40, PS67], Bi-LSTM (3): [PS8, PS35, PS13], VAE (3): [PS46, PS35, PS30], DNN (2): [PS60, PS16], RvNN (2): [PS28, PS52], GCN (1): [PS67], Adversarial network (1): [PS56], AE (1): [PS28], AST-RNN (1): [PS49], BPTS (1): [PS28], Compare-Aggregate Model (1): [PS13], DAE (1): [PS57], GGNN (1): [PS33], GMN (1): [PS33], Skip-gram model (1): [PS63], T-LSTM (1): [PS52]	30
		Control flow graph (CFG)	ANN (3): [PS40, PS32, PS44], Attention model (2): [PS43, PS40], GCN (2): [PS40, PS42], RAE (2): [PS42, PS29], DNN (1): [PS36], GAT (1): [PS59], GraphSAGE (1): [PS43], GRU (1): [PS58], NTN (1): [PS43], RtNN (1): [PS19]	10
		Program dependency graph (PDG)	CNN (2): [PS12, PS15], Attention model (1): [PS51], GAT (1): [PS51], LSTM (1): [PS51], MLP (1): [PS51]	3
		Statement tree (ST-Tree)	ASTNN (2): [PS9, PS53], Bi-GRU (2): [PS9, PS53], LSTM (2): [PS1, PS9], RvNN (2): [PS9, PS53], Attention model (1): [PS1], Bi-LSTM (1): [PS1], Discriminative model (1): [PS1]	3
		Data flow graph (DFG)	ANN (1): [PS32]	1
		Code semantic dependency (CSD)	Attention model (1): [PS50], CNN (1): [PS50], MLP (1): [PS50]	1

		Bytecode dependency graph (BDG)	CNN (1): [PS15]	1
Text	20	Source code	CNN (4): [PS23, PS22, PS10, PS25], Bi-LSTM (3): [PS8, PS35, PS61], AE (2): [PS23, PS28], ANN (2): [PS7, PS40], DNN (2): [PS7, PS17], LSTM (2): [PS40, PS25], RtnN (2): [PS23, PS28], Attention model (1): [PS40], BPTS (1): [PS28], GRU (1): [PS25], RAE (1): [PS39], VAE (1): [PS35]	12
		Textual information	LSTM (4): [PS31, PS50, PS38, PS27], CNN (3): [PS38, PS27, PS48], MLP (2): [PS31, PS50], Attention model (2): [PS38, PS50], RAE (1): [PS29], PV-DM (1): [PS11], DNN (1): [PS27], Bi-LSTM (1): [PS41], ANN (1): [PS38]	8
Numeric	17	Software metrics	CNN (7): [PS38, PS12, PS31, PS25, PS27, PS34, PS26], ANN (5): [PS41, PS38, PS4, PS6, PS7], DNN (5): [PS55, PS7, PS65, PS18, PS27], LSTM (3): [PS38, PS27, PS25], AE (2): [PS4, PS5], Attention model (1): [PS38], GRU (1): [PS25], MLP (1): [PS31], TCN (1): [PS64], VAE (1): [PS46]	17
Binary code	4	Bytecode	RAE (2): [PS20, PS29], ANN (1): [PS32], RvNN (1): [PS20]	3
		LLVM IR	DNN (1): [PS45]	1
Image	1	Image of source code	CNN (1): [PS21]	1

**A PS may belong to one or more fields.*

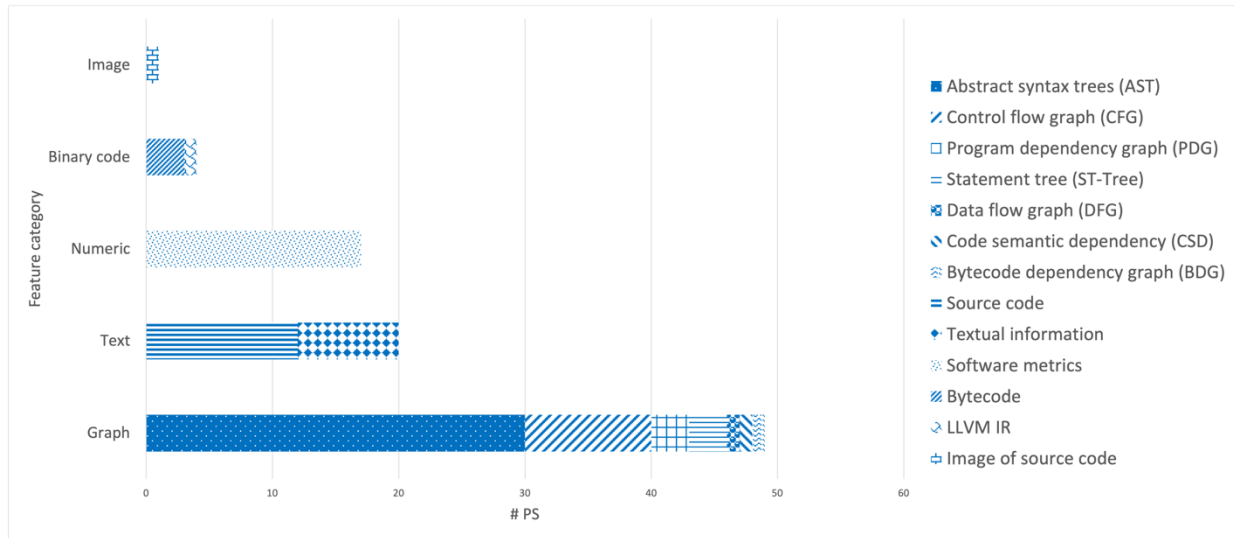


Figure 7 Features used in DL-based bad smell detection

How were the data used by the PSs preprocessed?

Many studies addressed data preprocessing steps, including data balancing, feature selection (FS), tokenization, data normalization, filtering, and data transformation, to handle data issues. Table 9 describes these data preprocessing steps along with the techniques used in the selected primary studies. A precisely stated technique name was presented along with the study. Three studies [PS53, PS49, and PS1] did not address the technique name, but mentioned to have used data balancing and normalization techniques; hence, we reported the techniques as not available (NA). Accordingly, 54% of the selected PSs used at least one data preprocessing method, while 46% did not include data preprocessing methods.

Data Balancing. In most cases, bad smells are not uniformly distributed, and the number of smelly instances is considerably lower than the number of non-smelly ones. This problem is known as the class imbalance problem [99], where the minority class becomes difficult to detect when trained on imbalanced data because the model is biased and in favor of the majority class. A data sampling technique is typically used to change the distribution of smelly and non-smelly instances. Few studies addressed this issue, and various techniques were used to overcome it, including undersampling [PS23, PS44, PS21, PS66], upsampling [PS42, PS52], and Synthetic minority over-sampling (SMOTE) [PS40, PS31]. In undersampling, a subset of the majority class is randomly selected and then discarded to achieve a balanced proportion. Conversely, in upsampling, a random subset of the minority class instances is duplicated and added to the actual dataset. In SMOTE, new artificial instances of the minority class are generated based on the feature similarity.

Feature Selection. Irrelevant and redundant features might affect the performance of ML models [100]. Few studies used the FS technique to solve this issue. FS is the process of finding and eliminating duplicated and irrelevant features to have a final subset of features that yield better predictive performances. Applying feature selection is a reasonable way of handling data with many features. Various feature selection techniques have been used, including autoencoders [PS4], chi-square [PS6], gain ratio [PS46], and recursive feature elimination (RFE) [PS5].

Tokenization. Tokenization is the process of dividing a sequence of characters to a sequence of tokens. This preprocessing method was utilized in 11 PSs [PS7, PS21, PS23, PS25, PS61, PS40, PS35, PS28, PS17, PS8, and PS10].

Data normalization. The top two normalization techniques used are special symbol removal and comment removal. Five studies normalized the source code by removing special symbols like '<', '>', ':' ... etc. [PS19, PS28, PS48, PS66, and PS67], while four PSs excluded comments [PS18, PS7, PS44, and PS66].

Filtering. Various filtering techniques were employed to filter out odd or trivial instances from the data and prevent model overfitting. Data filtering by removing instances based on the methods' lines of code (LOC) was considered in two PSs [PS17 and PS40]. Meanwhile, [PS40] excluded instances that had less than or equal to five LOC. Instance removal based on the number of tokens was performed in [PS28], which excluded instances that had less than 50 tokens.

Data transformation. Some studies transformed the source code by adding or modifying the identifier names and the literal values by replacing variables and values by their type,

such as $\langle \text{int} \rangle$, $\langle \text{string} \rangle$, ... etc. [PS12, PS29, PS7, PS42, PS66, PS44, PS39], tagging each token with its part of speech [PS48] or splitting individual terms into one or more terms [PS48].

One study [PS21] concluded that using the undersampling data balancing technique yielded better and more reliable results. However, we cannot derive a solid conclusion because only a few studies have explored the class imbalance issue. Comprehensive and empirical studies are needed to investigate the impact of various data balancing and feature selection techniques on the DL models for bad smell detection. Most studies did not specify whether or not they handled data-related issues, which might be due to the fact that DL models are relatively robust to noise compared to conventional ML approaches [101]. Although some studies normalized the source code to minimize memory and processing consumption, the techniques used might have affected the detection performance. Further experiments are needed to show the feasibility of such techniques. For instance, the removal of special symbols is a well-known and -established normalization technique in NLP, but its application to the source code may not be suitable because unlike in natural language, special symbols have meanings in programming languages. Nevertheless, data issues can dramatically affect the ML model performance. Future studies related to DL-based bad smell detection should investigate the robustness of DL models.

RQ4.2 summary. Fifty-four percent of the PSs addressed the data preprocessing methods, while 46% did not include them. The frequently used techniques for class imbalance were the undersampling, upsampling, and SMOTE techniques. Autoencoders, chi-square, gain ratio, and recursive feature elimination (RFE) were employed to address the irrelevant and redundant feature issue. Other preprocessing methods like tokenization, data

normalization, filtering, and data transformation have been used, but further empirical studies are needed to investigate the impact of these techniques on the DL models for bad smell detection.

Table 9 Data preprocessing and techniques used

Data preprocessing	Techniques (#PS): *
Data balancing	Undersampling (4): [PS23, PS44, PS21, PS66], Upsampling (2): [PS42, PS52], SMOTE (2): [PS40, PS31], NA(2): [PS53, PS49]
Feature selection	Autoencoder (1): [PS4], Chi-square (1): [PS6], Gain ratio (1): [PS46], Recursive feature elimination (RFE) (1): [PS5]
Tokenization	Splitting source code to tokens (11): [PS7, PS21, PS23, PS25, PS61, PS40, PS35, PS28, PS17, PS8, PS10]
Data normalization	Special symbols removal (5): [PS19, PS28, PS48, PS66, PS67], Comment removal (4): [PS18, PS7, PS44, PS66], Trimming (2): [PS15, PS12], Min–max normalization (2): [PS15, PS64], Stop nodes removal (2): [PS29, PS63], Whitespace and newline removal (2): [PS66, PS39], Pretty-printing (2): [PS66, PS7], Stop words removal (1): [PS19], Lowercase (1): [PS25], Not specified: (1): [PS1]
Filtering	Methods' lines of code (LOC) (2): [PS17, PS40], Number of tokens (1): [PS28], Singular value decomposition (SVD) (1): [PS11], Action filter based on API call similarity (1): [PS18], Mean input size (1): [PS23], Removing data based on predefined assumptions (1): [PS27], Number of nodes in a tree (1): [PS49], Action filter based on methods similarity (1): [PS55], Size similarity filter (1): [PS55]
Data transformation	Transforming identifier names and literal values (7): [PS12, PS29, PS7, PS42, PS66, PS44, PS39], Part of speech tagging (1): [PS48], Splitting individual terms (1): [PS48]
Not mentioned	Not mentioned: (31): [PS2, PS20, PS50, PS57, PS24, PS34, PS51, PS56, PS43, PS30, PS45, PS32, PS47, PS3, PS60, PS62, PS37, PS41, PS14, PS9, PS38, PS33, PS54, PS65, PS36, PS22, PS26, PS16, PS58, PS13, PS59]

*A PS may belong to one or more fields.

How are the data encoded into formats that are consumable by deep learning models?

After feature selection and processing, some of the features must be encoded to fit the neural network requirements. Encoding techniques are used to transform features like graphs and words into appropriate input (i.e., numerical representation). Table 10 presents the encoding techniques utilized in the identified PSs. Thirty-seven percent of the selected PSs did not employ any encoding techniques, while 63% considered one or more encoding strategies. Most of the studies used text-based encoding (57%). Among them, word2vec was the dominant embedding technique, representing 32% of usage. This was followed by the term frequency–inverse document frequency (TF–IDF), one-hot, and Doc2vec

techniques representing 7, 6, and 4% of the studies, respectively. The graph-based embedding techniques were used in 6% of the primary studies.

Although many studies used graph-based features (RQ4.1), most of them used text-based encoding techniques for feature embedding. This shows the need for new embedding techniques dedicated to encoding the structural representation of the source code. Only one study utilized a code-specific encoding technique (i.e., code2vec [PS23]). The majority used conventional encoding techniques to encode natural language text. Embedding for natural language text usually employs a vocabulary size from thousands to millions. In programming languages, however, the vocabulary size is considered smaller. Therefore, further research for encoding the source code would be interesting.

RQ4.3 summary. Approximately 37% of the papers did not employ any encoding techniques, while 63% considered one or more. The majority of the studies used text-based encoding (57%). word2vec was the dominant embedding technique, with a percentage of 32%. Graph-based embedding techniques were used in 5% of the primary studies. New embedding techniques dedicated to source code encoding are needed.

Table 10 Encoding techniques used in the PSs

Category	Encoding	PS *	#PS
Graph-based	Ast2vec	[PS66]	1
	Graph embedding technique (HOPE)	[PS29]	1
	Graph2vec	[PS44]	1
	Tree-based skip-gram	[PS24]	1
Text-based	Code2vec	[PS23]	1
	Coding criterion	[PS30]	1
	Continuous bags of words (CBOW)	[PS61]	1
	Doc2vec	[PS11, PS22, PS40]	3
	One-hot	[PS28, PS43, PS51, PS20]	4

	Position aware character embedding (PACE)	[PS54]	1
	Sent2vec	[PS63]	1
	Skip-gram	[PS63]	1
	TF-IDF	[PS39, PS19, PS67, PS10, PS25]	5
	Word2vec	[PS9, PS53, PS60, PS13, PS31, PS58, PS38, PS18, PS48, PS67, PS49, PS44, PS40, PS2, PS37, PS39, PS16, PS41, PS57, PS27, PS50, PS62, PS56]	23
Not used	-	[PS36, PS55, PS35, PS14, PS4, PS52, PS1, PS5, PS34, PS12, PS59, PS33, PS47, PS32, PS64, PS3, PS17, PS45, PS46, PS42, PS15, PS8, PS26, PS21, PS7, PS6, PS65]	27

**A PS may belong to one or more fields.*

3.3.5 Which deep learning models are used for bad smell detection?

Which deep learning models are used as predictive classification models?

Table 11 and Figure 8 summarize the studies that employed DL models for classification in bad smell detection, categorized into the main DL family that each model belongs to [102], along with the primary studies. The DL models that did not belong to any category were specified as “others.” Feed-forward neural network-based DL models were the most widely explored category (48%), possibly due to network simplicity. It does not have additional layers like convolution or pooling layers. Convolutional network-based models represent the second most frequently used category (24%). Recurrent networks (18%) and autoencoders (8%) were the two least used categories. Among the 17 DL models, the three most frequently employed DL models were CNN (11), ANN (10), and DNN (9). The remaining DL models were explored in up to three primary studies.

RQ5.1 summary. Feed-forward neural network-based models were the most widely explored category (48%). The total number of DL models used in the PSs was 17 models. The CNN (11), ANN (10), and DNN (9) were the most popular DL models used to build

prediction classification models for bad smell detection. Many DL models have not been used; hence, future studies may explore new models like generative Adversarial networks (GANs) and radial basis function networks (RBFN).

Table 11 DL models used for predictive classification

DL family	DL model	PS *	#PS
Feed-forward neural networks	Artificial neural networks (ANN)	[PS66, PS7, PS38, PS32, PS40, PS44, PS24, PS6, PS41, PS4]	10
	Deep neural networks (DNN)	[PS18, PS27, PS7, PS16, PS55, PS65, PS60, PS36, PS17]	9
	Multilayer perceptron (MLP)	[PS50, PS51, PS31]	3
	AST-based neural network (ASTNN)	[PS53, PS9]	2
Convolutional networks	Convolutional neural network (CNN)	[PS26, PS27, PS15, PS25, PS23, PS2, PS34, PS48, PS21, PS57, PS12]	11
	Temporal convolutional networks (TCN)	[PS64]	1
Recurrent networks	Long short-term memory (LSTM)	[PS27, PS25, PS49]	3
	Recurrent neural network (RtNN)	[PS23, PS19]	2
	Gated recurrent unit (GRU)	[PS25]	1
	Bidirectional long-short term memory (Bi LSTM)	[PS8]	1
	AST-based recursive neural network (AST-RNN)	[PS49]	1
	Backpropagation through structure (BPTS)	[PS28]	1
Autoencoder	Autoencoder (AE)	[PS23]	1
Others	Discriminative model	[PS1]	1
	Neural tensor network (NTN)	[PS43]	1
	Adversarial network	[PS56]	1
	Compare-aggregate model	[PS13]	1

**A PS may belong to one or more fields.*

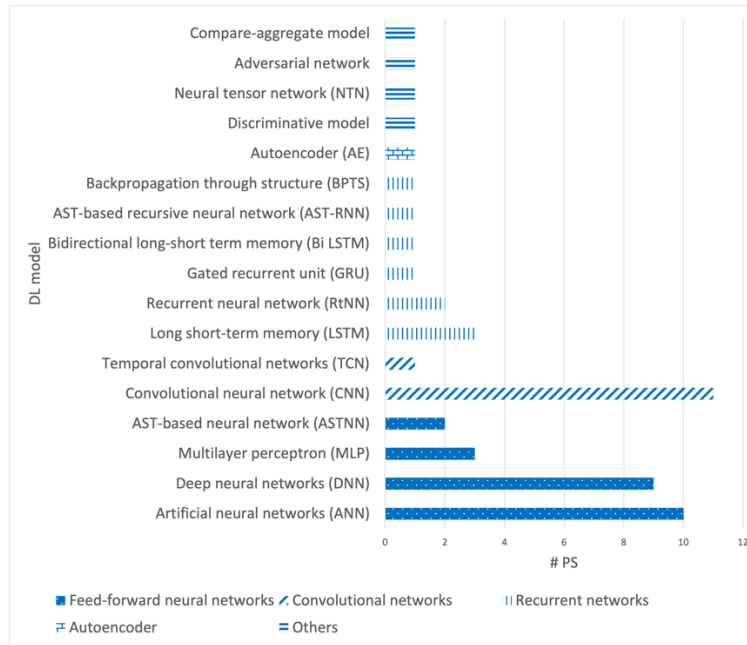


Figure 8 DL models used for predictive classification

Which deep learning models are used for representation learning?

Table 12 and Figure 9 present the studies that employed DL models for representation learning in bad smell detection. The recurrent network-based models were the most popular categories used to learn representation (38%). Convolutional networks (19%), autoencoder (18%), and attention networks (14%) were the second top categories. Neural (4%) and graph networks (4%) were not commonly used for representation learning. A total of 24 DL models were used, with the most popular models being the LSTM (9), Attention model (8), CNN (7), and RAE (6).

RQ5.2 summary. Recurrent network-based models were the most popular category used to learn representation (28%). A total of 24 DL models were used, with the most popular models being the LSTM (9), Attention model (8), CNN (7), and RAE (6). Future studies may explore new DL models for representation learning, such as the radial basis function networks (RBFN).

Table 12 DL models used for representation learning

Category	DL model	PS *	#PS
Recurrent networks	Long short-term memory (LSTM)	[PS1, PS50, PS31, PS38, PS40, PS52, PS9, PS24, PS51]	9
	Recursive neural network (RvNN)	[PS53, PS52, PS9, PS28, PS20]	5
	Bidirectional long short-term memory (Bi-LSTM)	[PS1, PS35, PS13, PS61, PS41]	5
	AST-based long short-term memory (AST-LSTM)	[PS40, PS62, PS56]	3
	Bidirectional gated recurrent unit (Bi-GRU)	[PS53, PS9]	2
	Gated recurrent unit (GRU)	[PS58]	1
	Recurrent neural network (RtNN)	[PS28]	1
	Tree long short-term memory (T-LSTM)	[PS52]	1
Convolutional networks	Convolutional neural network (CNN)	[PS50, PS31, PS10, PS3, PS37, PS38, PS22]	7
	Tree-based convolutional neural network (TB-CNN)	[PS54, PS66, PS47, PS14]	4
	Graph convolutional network (GCN)	[PS67, PS42, PS40]	3
Autoencoder	Recursive autoencoders (RAE)	[PS16, PS29, PS39, PS42, PS60, PS20]	6

	Variational autoencoder (VAE)	[PS35, PS46, PS30]	3
	Autoencoder (AE)	[PS5, PS28, PS4]	3
	Denoise autoencoder (DAE)	[PS57]	1
Attention networks	Attention model	[PS1, PS50, PS47, PS43, PS38, PS40, PS67, PS51]	8
	Graph attention networks (GAT)	[PS59, PS51]	2
Feed-forward neural networks	Artificial neural networks (ANN)	[PS32, PS41]	2
	Deep neural networks (DNN)	[PS45]	1
Graph networks	Gated graph neural networks (GGNN)	[PS33]	1
	Graph matching networks (GMN)	[PS33]	1
	Graphsage	[PS43]	1
Others	Paragraph vector distributed memory (PV-DM)	[PS11]	1
	Skip gram model	[PS63]	1

**A PS may belong to one or more fields.*

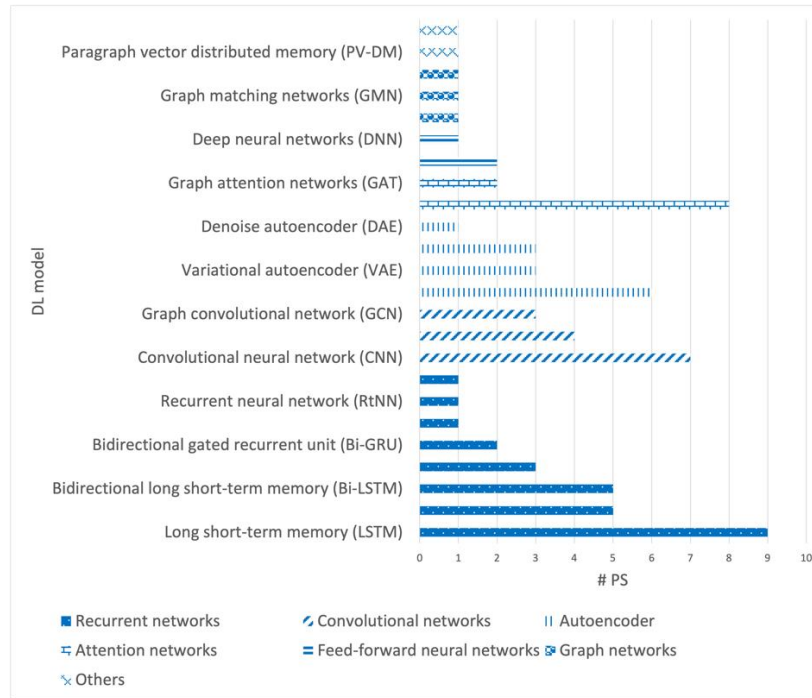


Figure 9 DL models used for representation learning

Which training hyperparameters are considered in building deep learning models?

Various hyperparameters are involved in the DL model development due to the model's complexity. The considerations for the hyperparameter training are explored in this research question. Table 13 summarizes all hyperparameter values used in the selected PSs. Hyperparameters are certain parameters that affect the learning process. Those considered in this review (i.e., initialization, learning rate, loss function, optimization algorithms, and overfitting handling) affect the DL model training. Not all reviewed studies mentioned the considered training hyperparameters. Figure 10 shows the percentage of papers that stated the method used in each training parameter. Initialization was the least considered parameter, being mentioned in only 15% of the PSs. The loss function was the most considered training parameter, with almost half of the PSs (55%) stating the used loss function. Overfitting handling, learning rate, and optimization algorithm were considered in 27, 36, and 40% of the PSs, respectively.

Initialization is the process of assigning the neural network weights. The most commonly used method for weight initialization is random initialization (six papers) [PS14, PS18, PS24, PS33, PS40, and PS55]. Other initialization methods include Xavier initialization [PS6, PS21], transfer learning [PS23], and Kaiming uniform method [PS51].

Learning rate controls the weight adjustment rate of the neural network based on the loss function used. The learning rate used in the selected PSs spanned from 0.00002 to 0.25, with 0.001 being the most commonly used (eight papers) [PS10, PS13, PS32, PS33, PS44, PS47, PS52, and PS66].

Loss function is used to compute the error (i.e., the difference between the predicted and actual values). Binary cross-entropy is the most popular loss function used in the reviewed studies (10 papers) [PS1, PS50, PS27, PS31, PS13, PS9, PS34, PS60, PS41, and PS51]. Cross-entropy is the second top loss function (eight papers) [PS10, PS58, PS3, PS32, PS25, PS64, PS44, and PS28]. The remaining functions were only used in five or fewer studies. Finally, three studies defined their own loss functions [PS59, PS35, and PS40].

Optimization algorithms are used to find optimal parameters (i.e., weights and biases) by minimizing the loss function. The most frequently used algorithm is the Adam optimizer, which was utilized in 12 studies [PS1, PS47, PS10, PS33, PS13, PS52, PS64, PS44, PS34, PS21, PS41, and PS51]. The second most common algorithm is stochastic gradient descent (SGD), which was employed in six PSs [PS18, PS54, PS55, PS64, PS39, and PS11].

Overfitting handling is the method used to overcome the overfitting of DL models known to be susceptible to this issue. Different methods can be used to overcome overfitting. The dropout method was the most frequently used, with 13 studies considering usage [PS18, PS15, PS10, PS32, PS16, PS25, PS55, PS23, PS60, PS67, PS57, PS19, and PS41]. Regularization techniques, such as L1 and L2 regularization, which were used in seven PSs [PS26, PS32, PS39, PS23, PS67, PS57, and PS19], were the second most popular method. Ensemble methods [PS26, PS27, PS29] and early stopping techniques [PS64, PS23] were not common.

Most training hyperparameters were neglected in the reviewed studies, despite being fundamental aspects of the DL models. Although initialization has a significant impact on neural networks, it was the least considered parameter [103]. We also emphasize herein

the need to explore other sophisticated methods not used in the selected PSs. For example, the Restricted Boltzmann Machine is an initialization method that can be used in future studies. Conjugate gradient (CG) is an optimization algorithm that can help speed up the training of DL models [104].

RQ5.3 summary. Initialization was the least considered parameter, being mentioned in only 15% of the PSs. The loss function was the most widely considered training parameter, with almost half of the PSs (55%) stating the usage of the function. Overfitting handling, learning rate, and optimization algorithm were considered in 27, 36, and 40% of the PSs, respectively. Other sophisticated methods for hyperparameter optimization that were not used in the selected PSs must be explored.

Table 13 Methods used in each training hyperparameter

Training consideration	Values/methods (#PS)
Initialization	[Random (6), Xavier Initialization (2), Transfer Learning (1), Kaiming Uniform Method (1), Not specified (57)]
Learning rate	[0.001 (8), 0.0001 (3), 0.01 (3), 0.0005 (2), 0.005 (2), 0.002 (2), 0.015 (1), 0.00002 (1), 0.0002 (1), 0.25 (1), 0.05 (1), 0.003 (1), Not specified (43)]
Loss function	[Binary Cross Entropy (10), Cross Entropy (8), Equation (5), Mean Squared Error (MSE) (4), User-defined (3), Relative Entropy (2), Euclidean Distance (2), Reconstruction Errors (1), Encode-Decode Loss (1), Adversarial Loss (1), Softmax Cross Entropy (1), Log Loss (1), Contrastive Loss Algorithm (1), Not specified (30)]
Optimization algorithm	[Adam (12), Stochastic Gradient Descent (6), Gradient Descent (4), Root Mean Square Prop (RMSPROP) (3), Adamax (2), Adagrad (2), Adadelata (1), 1BFGS (1), Nadam (1), Not specified (40)]
Overfitting handling	[dropout (13), regularization (7), ensemble methods (3), early stopping (2), Not specified (49)]

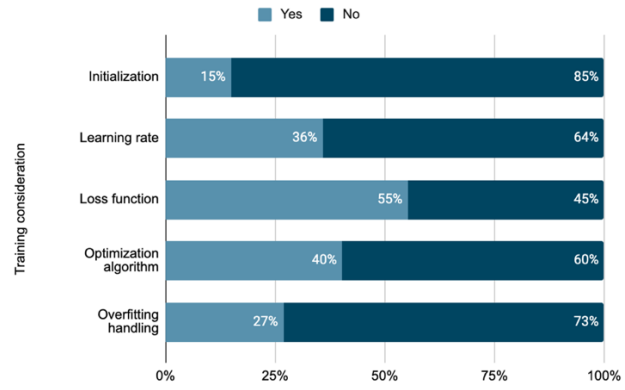


Figure 10 PS distribution over the training considerations

3.3.6 How are deep learning models evaluated and implemented in bad smell detection?

Which validation methods are used to evaluate the deep learning models?

The final phase of the DL model training (i.e., parameter learning phase) on the input data is the model validation in a real-world setting. Validation is a technique for assessing the performance of the trained model using previously unknown (i.e., unseen) data [105]. This phase is essential because it decreases overfitting, thereby increasing the generalization performance of DL models. It measures how well the model performed on previously unseen data. To achieve this goal, the dataset is usually split into two subsets: training and independent testing sets. When performing hyperparameter optimization, the testing set can no longer be considered an unbiased indicator of the model's generalization performance. Consequently, the training set is further separated into a third set known as a validation set used to select the optimum hyperparameter settings. Therefore, the testing set is used to objectively evaluate the model performance. Six validation methods were found in the primary studies briefly discussed below.

Train–test split. This category included studies performing a random split of the data into two sets: one used for training and another for testing. It also included studies that divided their data into training and testing sets and provided the number of instances used for training and testing without specifying the splitting ratio. The most common ratio was 80/20, where 80% of the data was used for training and 20% was utilized for testing [PS19, PS43, PS44, PS55, PS67, PS10]. The 70/30 followed, where 70% was used for training and 30% was used for testing [PS22, PS23, PS59]. One study considered 50/50 for training

and testing [PS34], while other studies did not specify a ratio [PS3, PS6, PS8, PS17, PS21, PS35, PS36, PS39, PS42, PS57, PS61, PS66].

Train–valid–test split. This category included studies that split their data into three sets as follows: a training set for training a model; a validation set for hyperparameter optimization on unseen data; and a test set for evaluating the model performance. The most common split ratio was 80/10/10 for training/validation/testing [PS1, PS10, PS14, PS33, PS37, PS52, PS54]. The next most common ratio was 60/20/20 for training/validation/testing [PS9, PS13, PS40]. One study considered a 70/20/10 split [PS53], while other studies divided their data into three sets without specifying the ratio [PS24, PS51, PS65, PS45, PS49].

K-fold cross-validation. This method involves partitioning the data into k equally sized folds. We iterate over the data for k times. In each iteration, $k-1$ folds and 1-fold are used as the training and test sets, respectively. The model performance is calculated by averaging the performance results of the k iterations. The most frequent value of k used in the primary studies was $k = 10$ [PS2, PS4, PS7, PS12, PS15, PS29, PS32, PS67], followed by $k = 5$ [PS30, PS31, PS40]. One study considered $k = 3$ [PS49]. Another considered $k = 7$ [PS50].

Leave-one-out cross-validation. This method is a specification of the k -fold cross-validation, where $k = n$. In this validation method, all instances are used for training except for one instance is used for testing (*leave-one-sample-out*). Another variation of this method is to use all subjects for training and only one subject for testing, regardless of the number of instances belonging to each subject or system (*leave-one-subject-out*). Two

studies used the *leave-one-sample-out* validation method [PS27, PS48], while three considered the *leave-one-subject-out* method [PS18, PS38, PS41].

Nested cross-validation. This method generates the train, validate, and test sets using two loops. The outer loop is used for error estimates, while the inner loop is employed for hyperparameter optimization. The inner loop divides the training set into two subsets: one is used for training, and another as a validation set. In this loop, the model is trained on the training set, and the hyperparameters minimizing the error on the validation set are selected. In the outer loop, the dataset is divided into many training and test sets. The errors are then averaged from each split to obtain a reliable model error estimate. This validation method was used in one study [PS64].

Table 14 shows the validation methods used in the selected primary studies. The train–test split (a.k.a., holdout) was the most commonly used validation method at 31%, followed by the train–valid–test split used in 23% of the PSs. Another common method found in the literature is the k-fold cross-validation representing 22% of the PSs. The leave-one-out (7%) and nested cross-validations (1%) were less popular. In the train–test and train–valid–test methods, the test set should be large, representative of the data, and not repeated in the training set. However, it is difficult for the testing set to have these features when the quantity of data available is limited. Hence, separating the data into various sets and only using a fraction for training can substantially impair the performance of data-hungry models, such as DL models. In these circumstances, cross-validation methods are preferred. However, cross-validation methods are time-consuming and require high computational power. This may be the reason behind the limited use of cross-validation methods. They take more time to train and evaluate ML models. Future studies might want

to explore the stability of the DL model performance when various validation methods are used, along with the time and computational complexity required to help researchers select the most appropriate method when validating their DL models in bad smell detection.

RQ6.1 summary. Approximately 17% of the PS did not mention the use of any validation method. The train–test split is the most commonly used validation method, with 31% percentage, followed by the train–valid–test split used in 23% of the PSs. Future studies might want to explore the stability of the DL model performance when various validation methods are used, along with the required time and computational complexity.

Table 14 Validation methods used by the PSs

Validation method	PS*	#PS
Train-test	[PS35, PS61, PS17, PS36, PS19, PS57, PS22, PS44, PS43, PS39, PS55, PS66, PS59, PS10, PS3, PS42, PS21, PS6, PS34, PS8, PS67, PS23]	22
Train-valid-test	[PS9, PS24, PS40, PS49, PS13, PS52, PS53, PS37, PS45, PS65, PS10, PS33, PS51, PS14, PS54, PS1]	16
K-fold cross-validation	[PS2, PS30, PS31, PS32, PS15, PS40, PS49, PS29, PS50, PS12, PS67, PS4, PS7, PS58, PS26]	15
NA	[PS46, PS60, PS28, PS56, PS20, PS47, PS16, PS25, PS5, PS63, PS62, PS11]	12
Leave-one-out	[PS48, PS27, PS41, PS18, PS38]	5
Nested cross-validation	[PS64]	1

*A PS may belong to one or more fields.

Which evaluation metrics have been more frequently used to report the accuracy of deep learning models?

Table 15 presents the types of evaluation metrics used by studies focusing on DL-based bad smell detection. Unsurprisingly, the majority of studies used metrics usually used in binary classification problems, such as precision, recall, F1-score, and accuracy. The top three evaluation metrics used were precision, recall, and F1-score, which corresponded to 85, 82, and 75% of the PSs, respectively. The accuracy measure used in 13% of the studies

followed. Some of the less common evaluation metrics included AUC (9%), MCC (4%), and ROC (3%). The C-score, confidence score, false alarm rate, mean squared error (MSE), successrate@t and false positive rate were used only in 1% of the PSs. Even though most papers employed two or more evaluation metrics, few of them merely used one metric [PS2, PS3, PS12, PS15, PS28, PS34, PS36, PS49, PS64]. Four of these studies used accuracy; two studies used precision; one used the f-score; one used recall; and one used the AUC.

Most of the studies employed precision and recall metrics, which are inadequate for assessing the effectiveness of a model because they rely on an arbitrarily chosen threshold and are not suitable for data with imbalance classes [106]. Using evaluation metrics that are threshold-independent (e.g., Brier score [107] and MCC [108]) is usually recommended for binary classification problems. Moreover, using evaluation metrics that are suitable in the case of imbalanced datasets (e.g., balanced accuracy and AUC [109]) is highly suggested.

RQ6.2 summary. Recall (85%), precision (82%), and f-score (75%) are the most commonly used evaluation metrics. Bad smell detection is considered a binary classification problem with imbalanced classes; thus, future research may consider other metrics, such as AUC, MCC, and Brier.

Table 15 Evaluation metrics used by the PSs

Evaluation metric	PS*	#PS
Precision	[PS35, PS40, PS61, PS58, PS17, PS52, PS26, PS1, PS7, PS2, PS9, PS19, PS57, PS32, PS62, PS60, PS13, PS22, PS29, PS44, PS37, PS47, PS53, PS56, PS39, PS55, PS14, PS66, PS54, PS4, PS46, PS28, PS59, PS18, PS63, PS5, PS10, PS51, PS65, PS38, PS30, PS31, PS41, PS42, PS27, PS21, PS6, PS48, PS50, PS16, PS25, PS33, PS24, PS8, PS67, PS23, PS11]	57

Recall	[PS35, PS40, PS61, PS58, PS17, PS52, PS36, PS26, PS1, PS7, PS9, PS19, PS57, PS62, PS32, PS60, PS13, PS22, PS29, PS44, PS37, PS47, PS53, PS56, PS39, PS55, PS14, PS66, PS54, PS4, PS46, PS59, PS18, PS63, PS5, PS10, PS51, PS65, PS30, PS31, PS41, PS42, PS27, PS21, PS6, PS48, PS50, PS16, PS25, PS33, PS24, PS8, PS67, PS23, PS38]	55
F-score	[PS35, PS40, PS61, PS58, PS52, PS64, PS26, PS1, PS7, PS9, PS19, PS57, PS62, PS32, PS60, PS13, PS22, PS29, PS44, PS37, PS47, PS53, PS56, PS14, PS66, PS54, PS4, PS46, PS59, PS18, PS63, PS5, PS10, PS51, PS30, PS31, PS41, PS42, PS27, PS48, PS6, PS50, PS16, PS25, PS33, PS24, PS8, PS67, PS23, PS38]	50
Accuracy	[PS19, PS15, PS41, PS3, PS21, PS12, PS34, PS5, PS45]	9
Area under the curve	[PS7, PS49, PS27, PS45, PS23, PS38]	6
Matthew's correlation coefficient	[PS48, PS27, PS38]	3
Receiver operating characteristic curve	[PS48, PS39]	2
C-score	[PS17]	1
Confidence score	[PS11]	1
False alarm rate	[PS21]	1
Mean square error (MSE)	[PS43]	1
Successrate@t	[PS11]	1
False positive rate	[PS39]	1

**A PS may belong to one or more fields.*

Which baseline techniques are used to evaluate the deep learning models?

A new model or approach proposed to tackle an existing problem must be compared with other baseline techniques to show the superiority of the proposed model. Therefore, an important step when evaluating the DL model performance is to compare the performance of the new model to that of available state-of-the-art models. In this research question, we investigated the baseline techniques used to compare the proposed DL models for bad smell detection. Comparisons are often gained by (1) re-implementing the models or (2) using the results from published models. The first approach is preferred, but due to the lack of source code availability (RQ6.4), some studies use the second approach. Nevertheless, the selected primary studies compared the proposed model with at least one of three baseline types: (1) other DL models used for bad smell detection; (2) other bad smell detection

approaches (e.g., metrics-based); and (3) conventional ML models such as LR and SVM. Table 16 presents the most frequent baseline techniques used in at least two PSs. Most PSs compared the proposed DL model with at least one DL-based detection approach (51%). Approximately 31% of the PSs used other detection approaches as a baseline comparison. Few studies compared their DL model with ML-based detection approaches (13%). Only 5% of all PSs did not compare the performance of their DL models with any other approach. Among the various DL-based approaches, CDLH [110] was the most frequently used baseline (14 papers), employing a hashing function to optimize the similarity between two vector representations of the AST and the Tree-LSTM models to encode the features represented as ASTs. Deckard [111] was the most widely used baseline approach (19 papers). It is a rule-based code clone detection approach that uses established rules to build characteristic vectors for each AST subtree, and then clusters them to detect the code clones. Sourcerercc [112] is another detection approach. It is a bag-of-words-based clone detection tool used as a baseline in 12 PSs. Among the list of ML-based approaches, the NB, RF, and SVM models were commonly used as the baselines. Two-thirds (39 papers out of 59) of the baselines that PSs used employed only one type of baseline, but considered more than one technique. One-third (20 papers out of 59) considered at least two baseline types. One paper used all three types of baselines [PS26]. Five used DL- + ML-based approaches [PS7, PS15, PS19, PS31, PS65]. Fourteen used DL-based + other approaches [PS32, PS38, PS39, PS41, PS43, PS44, PS50, PS52, PS57, PS59, PS62, PS63, PS66]).

Although most of the PSs compared their DL models with DL-based models, few studies considered conventional ML-based models. We recommend future studies to use both baseline types for comparison. Comparing DL models with conventional ML-based

models is as important as comparing them with DL-based models because “the least complex model that is able to learn the target function is the one that should be implemented,” as depicted in *Occam’s Razor Principle* [113].

RQ6.3 summary. Twelve percent of the studies did not report baseline comparisons. Approximately 66% compared their proposed model with the DL-based models, 46% compared theirs with other detection approaches, and 16% compared their models with the ML-based models. We recommend future studies to use ML- and DL-based baseline types for comparison.

Table 16 Baseline techniques used in the PSs

Baseline type	Baseline technique	PS*	#PS
DL-based detection approaches	CDLH	[PS39, PS60, PS66, PS56, PS52, PS14, PS67, PS32, PS16, PS54, PS33, PS44, PS40, PS9]	14
	DLC	[PS56, PS52, PS62, PS14, PS47, PS44, PS40]	7
	CNN	[PS61, PS52, PS19, PS53, PS23, PS64]	6
	Deepsim	[PS39, PS60, PS16, PS66, PS44, PS40]	6
	ASTNN	[PS67, PS58, PS33, PS44, PS13]	5
	Cclearner	[PS39, PS57, PS59, PS21, PS63]	5
	RtvNN	[PS66, PS67, PS32, PS58, PS33]	5
	DNN	[PS38, PS64, PS50, PS41]	4
	LSTM	[PS52, PS61, PS35, PS64]	4
	MLP	[PS31, PS15, PS19]	3
	Tree-LSTM	[PS52, PS35, PS13]	3
	Weighted-RAE	[PS60, PS16, PS67]	3
	Code2vec	[PS47, PS10]	2
	GGNN	[PS52, PS51]	2
	Oreo	[PS39, PS57]	2
	TBCCD	[PS37, PS51]	2
	TBCNN	[PS35, PS40]	2
	Word2vec	[PS10, PS22]	2
Other detection approaches	Deckard	[PS57, PS52, PS67, PS33, PS40, PS56, PS17, PS62, PS14, PS32, PS47, PS49, PS8, PS66, PS39, PS59, PS58, PS28, PS44]	19
	Sourcerercc	[PS57, PS56, PS17, PS52, PS62, PS14, PS67, PS47, PS8, PS44, PS40, PS55]	12

	Jdeodorant	[PS26, PS38, PS41, PS50, PS27]	5
	Nicad	[PS39, PS57, PS17, PS63, PS55]	5
	décor	[PS27, PS26]	2
	Jmove	[PS41, PS50]	2
	TACO	[PS27, PS30]	2
	CloneWorks	[PS39, PS55]	2
ML-based detection approaches	Naive Bayes (NB)	[PS7, PS15, PS25, PS12, PS48]	5
	Random Forest (RF)	[PS12, PS7, PS48, PS19]	4
	Support Vector Machine (SVM)	[PS26, PS12, PS48, PS3]	4
	J48	[PS31, PS12, PS48]	3
	Decision Tree (DT)	[PS26, PS3]	2
	Logistic Regression (LR)	[PS65, PS3]	2
No basslines	-	[PS2, PS6, PS29, PS34, PS36, PS42, PS45, PS46]	8

**A PS may belong to one or more fields*

Is the source code of deep learning models publicly available?

Table 17 presents the source code availability along with the PSs. Only 20 studies (30%) made the source code available online. The other 47 papers (70%) did not publish the source code. Nevertheless, having access to the source code provides many benefits, including the following: (1) it saves time for future researchers proposing new DL models; (2) it improves the motivation to replicate a paper's findings; and (3) it facilitates the baseline comparisons of DL models. Therefore, it is highly recommended to provide public access to the source code.

RQ6.4 summary. Only 20 studies (30%) made the source code available online; 47 papers (70%) did not publish the source code. Publishing the source code has many benefits; hence, it is highly recommended to make the source code available.

Table 17 Source code availability

Source code availability	PS	#PS
No	[PS35, PS61, PS49, PS58, PS52, PS20, PS12, PS36, PS64, PS62, PS1, PS2, PS19, PS57, PS60, PS13, PS22, PS53, PS56, PS37, PS55, PS14, PS66, PS4, PS46, PS28, PS59, PS5, PS10, PS45, PS30, PS31, PS15, PS41, PS3, PS42, PS21, PS6, PS48, PS50, PS34, PS16, PS25, PS33, PS8, PS67, PS38]	47
Yes	[PS40, PS17, PS26, PS7, PS9, PS32, PS44, PS29, PS47, PS43, PS39, PS54, PS18, PS63, PS51, PS65, PS27, PS24, PS23, PS11]	20

3.3.7 How are the bad Smell datasets being constructed and used?

Which systems and datasets are used in deep learning-based bad smell detection?

Table 18 presents the main datasets and systems used in the selected studies and datasets/systems used in more than two studies. The overall number of datasets considered in the PSs was 148 datasets. Of the total, 81% (120 datasets) were used solely in one study, 8% (11 datasets) were used in two studies; and 11% (17 systems) were used in 28 studies. Figure 11 shows the number of datasets utilized in the PS distribution. Most PSs (64%) evaluated the DL models using one or two datasets. Only one study used 30 datasets. The remaining 23 studies used three to 13 datasets.

The table shows that BigCloneBench is the most prevalent dataset presented in 42% of the selected studies. Ojclone is the second most common dataset, being used in 18% of the PSs. The datasets collected from Github repositories and Google code jam were used in 9% of PSs each. Among the software systems used as a dataset, JEdit is the most prevalent system, representing 12% of the PSs. ArgoUML is the second most popular system used in 9% of the PSs. The rest of the datasets listed in the table were used in three to four PSs. The top two frequent datasets were used for code clone detection. The remaining datasets

were employed in various bad smells, except for Google code jam and Atcoder, which were used solely for code clone detection.

Various datasets were used, and approximately 81% of these datasets were utilized only once. This denotes the lack of commonly accepted benchmarking datasets for bad smell detection studies. An exception is the code clone bad smell. Two benchmarking datasets existed and were widely used, explaining why the code clone is the most detected bad smell when using DL models. Moreover, the datasets presented in Table 18 have common characteristics: 1) they are created from open-source systems and are publicly available, and 2) they are long-lived systems and written in Java.

RQ7.1 summary. The total number of datasets considered in the PSs was 148 distinct datasets. Most PSs (64%) evaluated the DL models on one or two datasets. BigCloneBench (42%), Ojclone (18%), and JEdit (12%) were the most widely used datasets in the DL-based bad smell detection studies. Commonly accepted benchmarking datasets for bad smell detection are lacking.

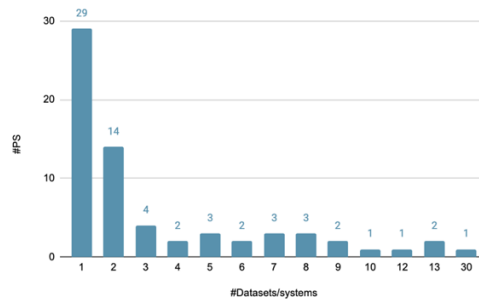


Figure 11 Number of datasets used by the PSs

Table 18 Data used by the PSs

Data used	PS*	#PS
Datasets		
BigCloneBench	[PS17, PS54, PS1, PS67, PS16, PS58, PS51, PS55, PS63, PS14, PS39, PS40, PS21, PS60, PS57, PS37, PS62, PS66, PS13, PS3, PS65, PS9, PS36, PS49, PS56, PS33, PS32, PS47]	28
Ojclone	[PS10, PS62, PS54, PS9, PS47, PS56, PS1, PS59, PS52, PS8, PS14, PS44]	12
GitHub repositories	[PS16, PS23, PS53, PS55, PS11, PS45]	6
Google code jam	[PS35, PS58, PS51, PS18, PS33, PS32]	6
Systems		
Jedit	[PS41, PS27, PS5, PS26, PS63, PS50, PS46, PS38]	8
Argouml	[PS5, PS28, PS22, PS48, PS26, PS34]	6
Junit	[PS41, PS48, PS27, PS50]	4
Weka	[PS41, PS42, PS27, PS50]	4
Pmd	[PS41, PS50, PS27, PS34]	4
Jhotdraw	[PS28, PS46, PS38]	3
Atcoder	[PS35, PS18, PS24]	3
Apache ant	[PS28, PS30, PS26]	3
Freeplane	[PS41, PS27, PS50]	3
Freemind	[PS38, PS46, PS34]	3
Eclipse	[PS5, PS22, PS48]	3
Areca	[PS41, PS27, PS50]	3
Apache lucene	[PS30, PS26, PS48]	3

*A PS may belong to one or more fields.

Which programming languages are most frequently used to construct datasets?

Most datasets were constructed from systems written in Java, C, and C++.

Table 19 shows the distribution of the programming languages considered in the selected primary studies. Java was the most commonly used programming language in 59 PSs, which may be due to the availability of open-source Java systems. C and C++ were the second and third most used programming languages (12 and 10 PSs, respectively). Other less common languages were C#, Python, Golang, and Scratch, which were used in four, four, one, and one PSs, respectively. One study detected bad smells at the model level [PS6].

Table 19 categorizes this study as programming language-independent (PL-indep).

Some primary studies explored cross-language bad smell detection and investigated the capability of DL models to detect bad smells across multiple programming languages. The studies that explored cross-language detection were [PS11] (C/C++, Java, Python, C#), [PS18] (Java, C#, Python), [PS23] (Java, C#), [PS24] (Java, Python), and [PS35] (Java, Python, C#, C/C++).

An interesting area of DL is transfer learning [PS23], where a DL model is trained on one programming language and then used for bad smell detection in other programming languages. This is beneficial, especially in the case of programming languages with low resources or datasets. It opens the door for designing tools that can be used universally, regardless of the system language.

RQ7.2 summary. Java was the most common programming language used in the 59 PSs, followed by C/C++ in 14 PSs. Datasets constructed from programming languages (e.g., C# and Python) are lacking.

Table 19 Programming languages of the datasets used

Programing language	PS*	#PS
Java	[PS27, PS17, PS28, PS54, PS64, PS1, PS18, PS30, PS67, PS29, PS7, PS38, PS16, PS31, PS58, PS43, PS51, PS55, PS19, PS15, PS63, PS34, PS12, PS14, PS23, PS39, PS40, PS5, PS2, PS22, PS48, PS21, PS60, PS57, PS11, PS37, PS62, PS66, PS41, PS13, PS42, PS3, PS53, PS65, PS9, PS26, PS36, PS49, PS56, PS33, PS4, PS50, PS32, PS25, PS46, PS47, PS35, PS24, PS20]	59
C	[PS62, PS54, PS9, PS47, PS56, PS1, PS59, PS11, PS52, PS8, PS14, PS44]	12
C++	[PS10, PS35, PS62, PS54, PS9, PS47, PS52, PS8, PS14, PS44]	10
C#	[PS35, PS18, PS23, PS11]	4
Python	[PS35, PS18, PS24, PS11]	4
Golang	[PS45]	1
Scratch	[PS61]	1
PL-indep	[PS6]	1

*A PS may belong to one or more fields.

Which approaches have been used to create the datasets?

Most studies tackled the problem of bad smell detection using the supervised learning approach (RQ3.2), which requires creating dataset oracles, which means labeling the extracted instances into smelly and non-smelly instances based on the bad smell being investigated. Accordingly, three main methods were used: (1) reference: using an existing dataset containing the labels from a reference; (2) automatic: using tools or rules for label creation; and (3) manual: manual inspection of the dataset and labeling them accordingly. Table 20 shows the approaches used by the primary studies. A total of 38 PSs used existing datasets; 21 papers automated the process, and 16 used the manual approach.

The labeling process automation was done using tools [PS7, PS23, PS42, PS48, PS53, PS55, PS64, PS65], similarity measure [PS8, PS10, PS18, PS24, PS35, PS43, PS45, PS52, PS61], semi-supervised labeling method [PS12], smell-introducing refactoring method [PS27], rules [PS34], or heuristics [PS65]. Most tools are used to detect code clone smells, such as Nicad [PS42, PS7], Sourcerercc [PS55, PS7], Ccfinderx [PS7], Cloneworks [PS7], Ctcompare [PS7], Deckard [PS7], Iclones [PS7], Simian [PS7], and Simcad [PS7]. Designite and Designite2 were utilized in [PS23, PS64] to extract the design smells, including complex methods, complex conditional, feature envy, and multifaceted abstraction.

The manual approach is usually combined with automatic and reference methods. Among the 61 studies, six merely used the manual method [PS28, PS2, PS11, PS50, PS6, PS29]; six first used an auto way to label the data and manually validated the extracted labels [PS42, PS23, PS64, PS65, PS48, PS7]; and four papers used an existing labeled dataset, but also manually validated the labels [PS57, PS63, PS3, PS40].

Most studies utilized an existing labeled dataset. Labeling is a time-consuming and labor-intensive process that might explain the extensive use of existing datasets and automatic approaches. Therefore, we highlight herein the need for publicly available labeled datasets and tools that can facilitate the dataset labeling process.

Table 20 Labeling methods used to create the oracles

Labeling method	PS*	#PS
Reference	[PS17, PS54, PS1, PS30, PS67, PS38, PS16, PS31, PS58, PS51, PS59, PS63, PS12, PS14, PS39, PS40, PS22, PS21, PS60, PS57, PS37, PS62, PS66, PS41, PS13, PS3, PS9, PS26, PS36, PS49, PS56, PS33, PS4, PS32, PS25, PS46, PS47, PS44]	38
Automatic	[PS10, PS61, PS27, PS64, PS18, PS8, PS7, PS43, PS55, PS19, PS34, PS12, PS23, PS48, PS24, PS42, PS35, PS53, PS65, PS45, PS52]	21
Manual	[PS42, PS23, PS3, PS40, PS28, PS2, PS64, PS48, PS65, PS57, PS63, PS11, PS50, PS6, PS29, PS7]	16

**A PS may belong to one or more fields.*

Various features were extracted and used (RQ4.1) to create the datasets. Table 21 presents the tools utilized by the selected primary studies for feature extraction. A broad range of tools can be used, but the table shows the tools used by at least two PSs. Twenty-four tools were found. Among them, 16 tools were used only once. Antlr (10 studies), Javalang (eight studies), and Pycparser (six studies) were the top three tools used for feature extraction. Antlr is mainly used to transform the source code written in C#, Python, and Java into abstract syntax trees (AST). Javalang is a python library used to extract the AST and tokenize the Java source code. Pycparser is used to generate the ASTs of the source code written in the C language. Java Development Tools (JDT) and Soot are used in four PSs. JDT is mainly used to extract method blocks and generate ASTs. Soot is used to extract the CFG.

Table 21 Software tools used for feature extraction

Tools	PS *	#PS
Antlr	[PS16, PS17, PS28, PS40, PS21, PS60, PS18, PS63, PS61, PS8]	10
Javalang	[PS53, PS9, PS56, PS33, PS37, PS62, PS25, PS14]	8
Pycparser	[PS62, PS56, PS9, PS14, PS52, PS59]	6
Java Development Tools (JDT)	[PS15, PS28, PS27, PS49]	4
Soot	[PS43, PS40, PS51, PS58]	4
Ast2bin	[PS16, PS39]	2
Codesplit	[PS53, PS23]	2
Eclipse Astparser	[PS3, PS17]	2

**A PS may belong to one or more fields.*

The ratio of the PSs that made their dataset publicly available was only 30% (20 papers) compared to 70% (47) that did not publish their dataset or used a publicly available dataset without re-publishing it (Table 22). Appendix C presents the publicly available datasets found in the primary studies. Most of the publicly available datasets were for the code clone smell, which was in alignment with the conclusion of the previous research questions. The code clone smell has many supported tools. Unlike other bad smells that need a careful inspection, labeling can easily be automated using similarity measures.

Table 22 Dataset availability

Dataset availability	PS	#PS
No	[PS10, PS61, PS17, PS28, PS20, PS1, PS30, PS67, PS8, PS38, PS16, PS31, PS58, PS55, PS19, PS15, PS52, PS59, PS63, PS34, PS6, PS12, PS5, PS2, PS22, PS21, PS60, PS57, PS37, PS62, PS66, PS41, PS13, PS35, PS42, PS3, PS53, PS36, PS49, PS56, PS33, PS4, PS45, PS50, PS25, PS46, PS47]	47
Yes	[PS27, PS64, PS54, PS18, PS29, PS7, PS43, PS51, PS14, PS23, PS39, PS40, PS48, PS11, PS24, PS65, PS9, PS26, PS32, PS44]	20

RQ7.3 summary. Most of the studies (38 PSs) used existing labeled datasets; 21 PSs automated the dataset labeling process, and 16 performed a manual inspection. Twenty-four tools were used for feature extraction. Among them, 16 tools were used only once.

Antlr (10 PSs), Javalang (eight PSs), and Pycparser (six PSs) were the top three tools used for feature extraction. We highlight herein the need for publicly available labeled datasets and tools that can facilitate the dataset labeling process.

3.4 Performance Analysis

We now analyze the DL model performance to answer RQ8. We have reported the performance for the dataset, bad smells, DL models, encoding techniques, preprocessing techniques, and features used. Synthesizing the data from several studies is difficult and many SLRs do not provide a synthesis [114]. This SLR followed the practices of [115] and [116] for data synthesis and analysis by combining and synthesizing our qualitative data. The main motivation behind using this approach is to generate a rich picture of the DL model performance for bad smell detection with respect to multiple factors. This was achieved by using boxplots depicting the DL model performance for various factors, such as dataset, features, bad smells, and encoding techniques. The boxplots contained the detection performance of three evaluation metrics: precision, recall, and f-score. We selected these measures because they were the most used evaluation metrics in the PSs. The number of studies shown in each boxplot was different from that reported in the previous section. This was because some studies used evaluation metrics other than precision, recall, and f-score; hence, they were excluded from the boxplots. For instance, the Bigclonebench dataset was used in 28 studies (Table 18). However, in Figure 12, the number of studies that used Bigclonebench was only 27 ($N = 27$) because they did not report the detection performance in terms of precision, recall, and f-score in [PS36]. Moreover, data were only included in each boxplot if at least three PSs utilized the same

technique/feature/dataset (i.e., N must be greater than or equal to 3). Finally, we only considered the best performance if a study reported the results of multiple configurations. It is worth mentioning that the performance analysis does not show strengths and weaknesses of the DL models as each PS conducted its experiment on a context that cannot be generalized to another context, rather it shows the current state of DL models performance in a rich picture, and research can be inspired by this analysis to explore new techniques to improve the detection performance.

3.4.1 Dataset

Figure 12 presents the detection performance relative to the dataset used. We observed a wide variety of reported model performances while using the Github repositories because they were usually collected from a wide range of systems. The Bigclonebench dataset showed the least variation in the detection performance and the best precision, recall, and F1 scores. BigCloneBench is a clone detection benchmark dataset that satisfies the benchmarking criteria such as 1) representative (it has all types of code clones), 2) experimentally verified (it has been evaluated), and 3) reusability (it is publicly available). There is a need for the development of high-quality benchmark bad smells datasets to be used in training and evaluating the DL model. The figure also illustrates that building models for certain systems may be more challenging than for others. Most datasets achieved higher recall scores compared to precision scores.

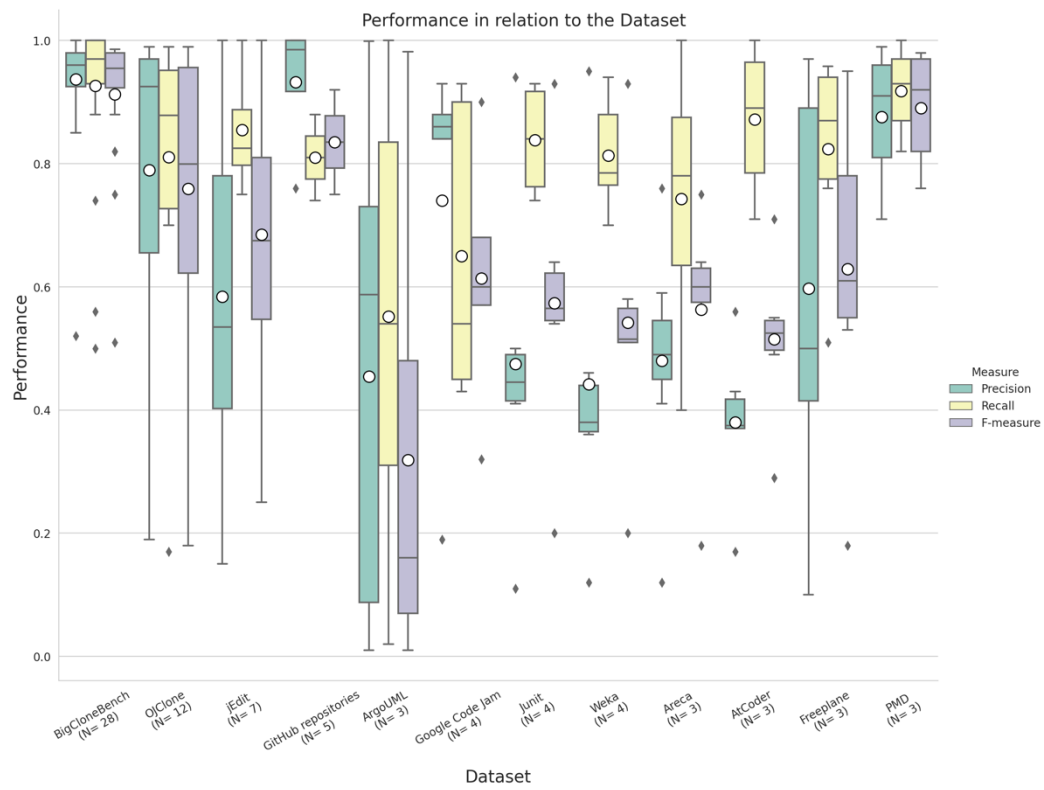


Figure 12 Detection performance with respect to the datasets

3.4.2 Bad smells

Figure 13 depicts the detection performance of the most used bad smells. The code clone smell achieved the highest detection performance compared to the other smells, which might be due to the large interest of the research community in code clone smell detection. Accordingly, new techniques were proposed to improve the detection performance. Feature Envy, Long Method, and God Class have achieved similar detection performances in terms of recall and nearly similar precision scores. Therefore, there is a need for new DL models, features, or encoding techniques to enhance the detection of performance of Feature Envy, Long Method, and God Class code smells.

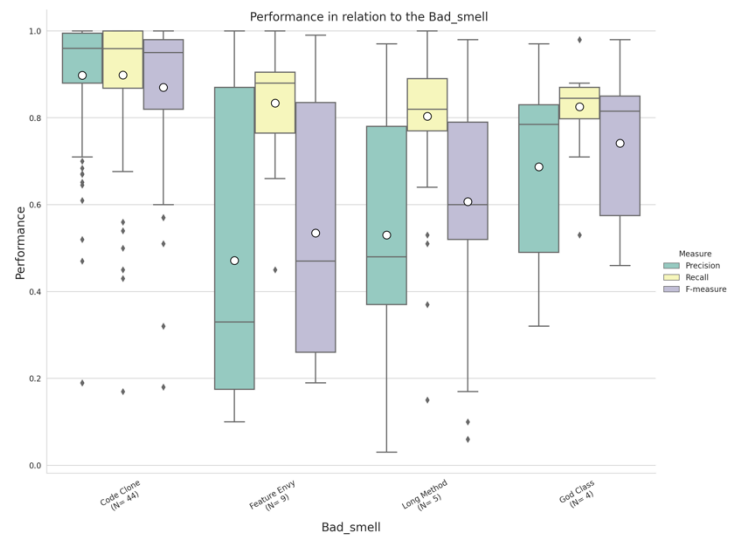


Figure 13 Detection performance with respect to the bad smells

3.4.3 Deep learning models

Figure 14 and Figure 15 show the detection performances of the DL models used for predictive classification and representation learning for bad smell detection, respectively. Figure 14 depicts the similar recall scores for the DL models. The precision scores varied between the models and the F1 scores. The CNN and ANN models showed the lowest precision scores. Moreover, most of the DL models had higher or nearly equal recall scores compared to their precision scores. An exception to that is the DNN models, which showed higher precision scores.

Among the DL models used for representation learning (Figure 15), RAE- and VAE-based models performed well. The TBCNN and AST-based LSTM models achieved high detection performances. The LSTM and the Bi-LSTM did not perform well in bad smell detection. RAE and VAE, which are autoencoder variations, is a good technique commonly used for representation learning [102]. The TBCNN is a CNN variation specially designed to encode the source code ASTs. The AST-based LSTM is a variation of the well-known LSTM model adapted to work with AST. This explains the good performance of these two models. The CNN and the LSTM are well suited and very effective in computer vision and NLP, respectively [87]. Using these models for bad smell detection must be further explored to enhance the detection performance. A general observation is that the DL models specifically designed to encode features for bad smell detection (e.g., TBCNN and AST-based LSTM) achieve higher scores; more experiments are needed to show the superiority of these models over other DL models.

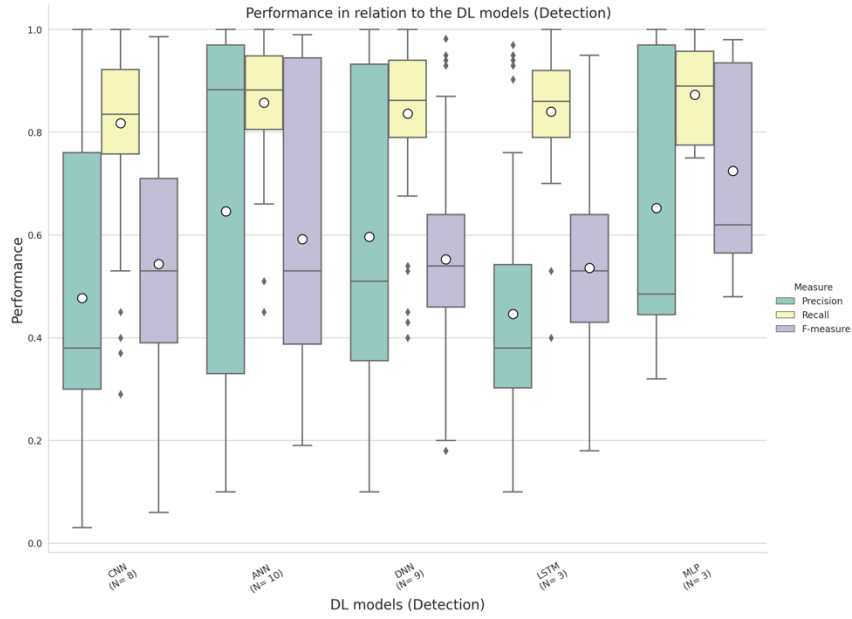


Figure 14 Detection performance with respect to the DL models for predictive classification

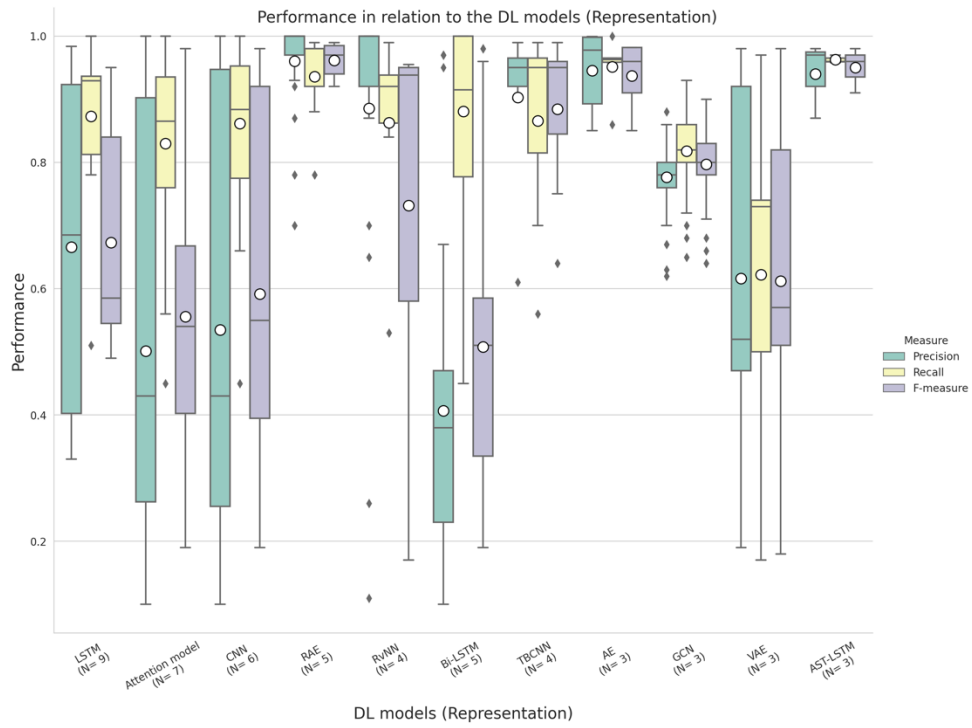


Figure 15 Detection performance with respect to the DL models for representation learning

3.4.4 Encoding techniques

Figure 16 presents the encoding techniques used in the PSs with respect to the detection performance. We show only studies that reported the use of either Word2vec or TF-IDF, along with those that did not use any encoding technique. The figure suggests that the detection performance of the DL models is not related to the employed encoding technique. The studies that did not employ any encoding techniques had nearly similar scores to word2vec encoding. TF-IDF showed less performance variation due to the low number of the PSs that used it (three studies) compared to the word2vec (23 studies) and no (22 studies) groups. Most of the studies used DL for representation learning, replacing the need for encoding. This did not dramatically affect the DL model performance.

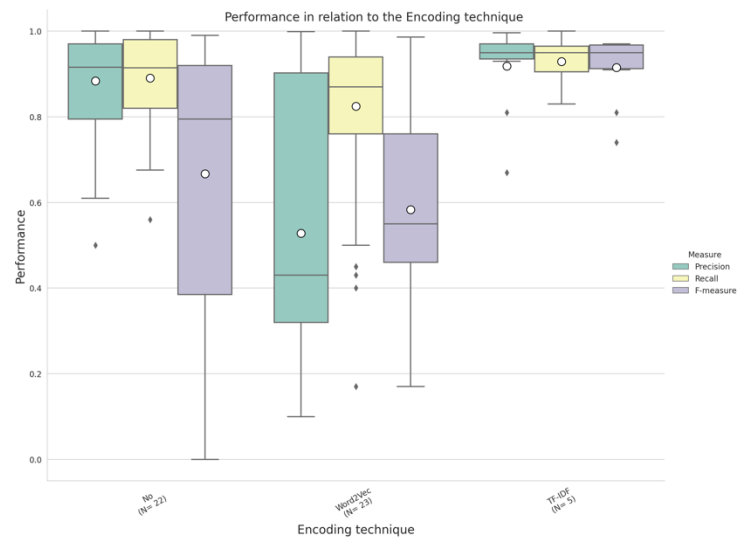


Figure 16 Detection performance with respect to the encoding techniques

3.4.5 Preprocessing techniques

Figure 17 illustrates the preprocessing techniques used in the PSs with respect to the detection performance. The figure shows that studies that utilized filtering and tokenization have less performance variation compared to feature selection, class imbalance, and transformation techniques. Studies that have employed normalization techniques have the highest scores with minimal variation in terms of precision and recall metrics.

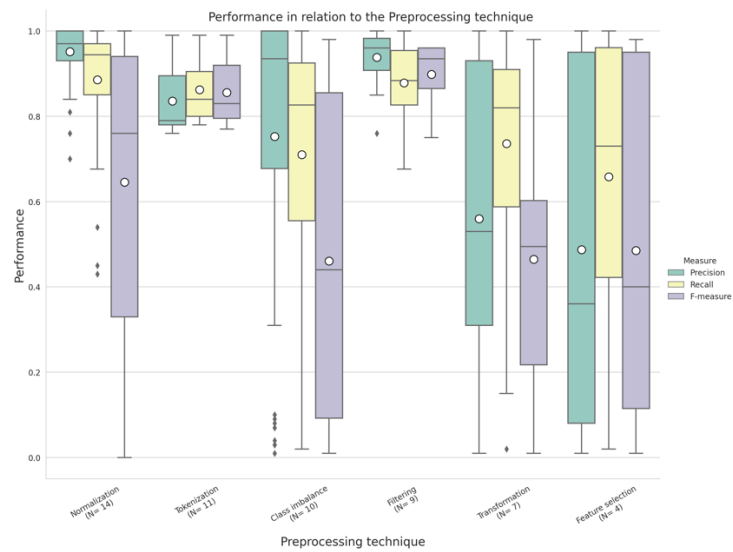


Figure 17 Detection performance with respect to the preprocessing techniques

3.4.6 Features

Figure 18 illustrates the detection performance relative to the features used in the PSs. We observe that using software metrics and statement trees as features have achieved higher performance scores compared to textual information. Future studies may explore other techniques that can be used to capture features extracted from textual data. A small performance variation was observed between models using other features (i.e., AST, source code, and CFG).

RQ8 summary. Many factors can affect the DL model performance, but the DL model selection has the highest impact. The DNN, autoencoders (RAE and VAE), TBCNN, and AST-based LSTM showed the best detection performances. Moreover, building DL models for certain systems may be more challenging than for others. The code clone smell achieved the highest detection performance among all smells. The DL model performance for bad smell detection is not related to the encoding technique used. Finally, a small performance variation was observed between models with respect to the features used. We highly suggest that future studies investigate the effectiveness of each factor on the DL model performance. This will help in deriving solid conclusions about the factors that are most influential and need to be considered when building DL models.

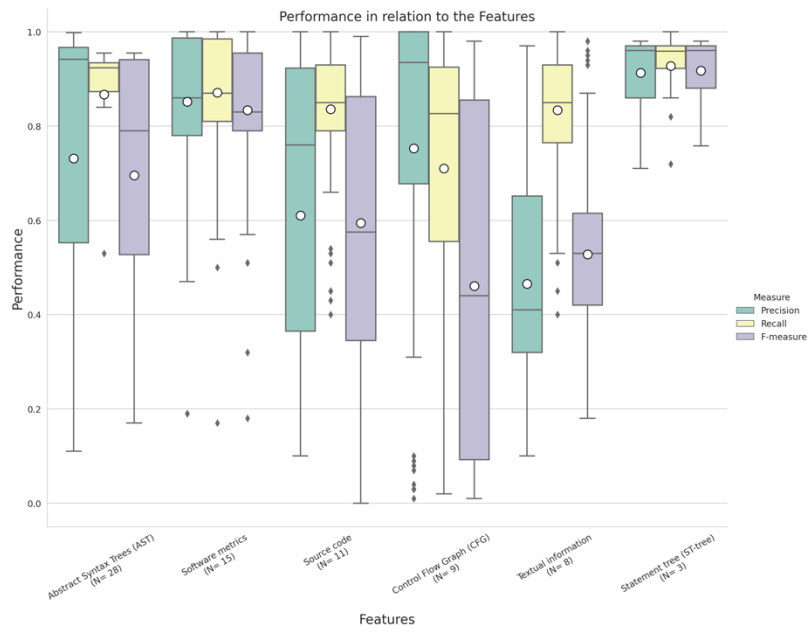


Figure 18 Detection performance with respect to the features

3.5 Discussion and Implications

This section discusses how DL-based bad smell detection may be further explored to drive toward a more accurate detection performance. We summarize the main findings here and pinpoint open issues and opportunities for future research.

3.5.1 Exploring other bad smells (RQ2)

Although 49 types of bad smells were detected using DL techniques, Code Clone was the most detected code smell in the PSs. This smell type might have been frequently detected owing to the following reasons: 1) availability of large-scale benchmarking labeled datasets; 2) availability of tools that can detect this smell type; 3) low cost of building a new labeled dataset; 4) availability of tools; and 5) nature of this bad smell, which facilitates the labeling process.

In terms of DL utilization for bad smell detection, we observed that most of the PSs focused on one smell type. Other bad smell types are also known to negatively influence the software quality [117], and they are found to frequently occur in software systems [118]. A research gap exists in the detection of bad smells other than the Code Clone smell using DL techniques. Therefore, we recommend further research to exploit and fill this gap. The Feature Envy, Data Class, God Class, and Long Method bad smells are frequently found in software systems [118], and they greatly affect the software quality [117]; however, very few studies have explored the usage of DL models to detect these smells. The capability of the Bi-LSTM, RtNN, RvNN, GCN, and GMN DL models has never been explored for detecting the Data Class, God Class, and Long Method bad smells, which can be investigated in future research.

3.5.2 Model-level detection (RQ2)

Bad smell detection at the early stages of the software development cycle reduces the maintenance cost. However, most of the PSs detected these smells at the code level. Only one study investigated design smell detection at the model level [PS6]. Despite the usefulness of detecting bad smells at the model level, it has received little attention in the context of DL-based detection.

In the future, researchers should investigate design smell detection by employing DL techniques. DL models should also be developed to detect bad smells at the model level using UML diagrams such as class and use case diagrams.

3.5.3 Exploring new learning approaches (RQ3)

Existing studies have tackled the problem of bad smell detection as a supervised learning problem. The supervised learning approach requires a huge number of labeled datasets; however, as discussed in RQ7, such datasets are lacking. The issue of lacking labels might be addressed by exploring other learning approaches. One learning approach that might tackle this problem is self-supervised learning, which has been used in other areas like natural language processing [119] and computer vision [120] and has shown promising results. The use of self-supervised learning for DL-based bad smell detection was investigated in one study [PS47]. However, self-supervised learning is worth further investigation. Future research may explore new pretext tasks directly derived from the source code rather than the AST. The self-supervised learning model scalability in various down-streaming tasks may also be investigated.

Another interesting research area to explore is transfer learning, which has been explored solely in one study [PS23]. They used this learning technique to detect bad smells across multiple programming languages. This learning approach is useful because most of the available datasets had been extracted from systems written in Java. The use of transfer learning is promising but requires more research. Future studies can investigate how to transfer the knowledge extracted from Java systems to other programming languages that lack datasets, such as Python, C#, and Golang. Moreover, researchers can also investigate the feasibility of transferring knowledge from one task (e.g., code clone detection) to another (e.g., Feature Envy, God Class, and Long Method detection).

3.5.4 From binary classification to multilabel (RQ3)

Current DL models output a single final prediction that indicates a smelly or a non-smelly instance. This is known as binary classification. The current architecture implies the need for a single DL model for each bad smell type. Most of the investigated studies used binary classification, and very few PSs used multi-classification [PS3, PS13, PS17, PS55, PS57], in which they did not solely predict whether or not a code fragment was a clone but also predicted the clone type. Nevertheless, multi-classification is useful in detecting multiple types of bad smells and can further be explored in the future.

However, using multi-classification remains an issue because a single instance may be affected by two smells at the same time. For example, a class can be a God Class and a Data Class. In this example, multi-classification is not enough. Multi-label classification permits the assignment of multiple labels to each instance. A recent study explored code smell detection using multi-label datasets and ML models [121]. Future research must consider utilizing DL models to detect various smell types in a single model and recognize

if an instance has two smell types. A possible method to approach this is by defining new activation functions other than the ReLU and Sigmoid functions at the output layer of a DL model (e.g., using softmax in the last neural network layer to return the probability of each class).

3.5.5 Exploitation of hyperparameter optimization (RQ4)

Most of the studies did not report the hyperparameters used in the DL models. Most of them also used default hyperparameters and did not consider hyperparameter optimization or a manual configuration of a set of hyperparameters. One study [PS64] performed a detailed hyperparameter optimization to optimize the learning rate, activation function, optimization algorithm, and the number of layers through Sequential Bayesian Model-based Optimization (SBMO). We emphasize herein the importance of the hyperparameter optimization of DL models because the development of DL models is an iterative and empirical process, and no one-size-fits-all hyperparameter configuration can be used. We recommend that the hyperparameter optimization for DL models in bad smell detection research should be considered as required rather than a choice. Manual configuration is also time-consuming. Future studies should utilize systematic optimization techniques like grid search, differential evolution, random search, and genetic algorithm [122].

3.5.6 Data preprocessing (RQ5)

Although data preprocessing is essential and can influence the DL model performance, many PSs did not explore most of the preprocessing techniques. Imbalanced labels and feature selection were utilized in a few PSs. A research gap exists in the usage of data preprocessing methods, such as data transformation, tokenization, data normalization, and filtering. These preprocessing methods have been explored, but many have not yet been

investigated in detail. For instance, log transformation, square root transformation, Z-score normalization, and decimal scaling are data transformation and normalization techniques that could be explored. The principal component analysis can also be used as a feature selection technique. Hence, more research on the use of data preprocessing techniques for DL-based bad smell detection is needed to fill the gap. Future research can investigate the effectiveness of these techniques on both the detection performance and the computational cost as well.

3.5.7 More attention on model evaluation (RQ6)

The use of multiple metrics for the DL model evaluation is important. Most of the PSs used at least two performance metrics, but most did not include the training time in the proposed DL model evaluation. Complex models may require additional resource consumption (e.g., storage and computational time cost). The model selection should consider all these aspects [123]. Therefore, we highly recommend that the training time be considered as an evaluation metric along with the performance metrics in DL-based bad smell detection.

Although most of the PSs compared their DL models with DL-based models, few studies considered conventional ML models. Comparing DL models with conventional ML models is as important as comparing them with DL-based models because, as stated by the *Occam's Razor Principle* [113], a simple model should be selected over a complex model when both can learn the target function. One study [96] applied the principle to the DL models. Surprisingly, the DL model failed to pass Occam's Razor Test. Future studies must use the two baseline types for comparative comparisons, that is, comparing the new DL models with previous and well-known DL models (e.g., CNN and LSTM) and with traditional ML models (e.g., DT and SVM).

3.5.8 Automated tools (RQ6)

We observed that existing DL models proposed for bad smell detection were not implemented in a software tool. In addition, the DL model's source code was not made publicly available. Automating the detection model can help researchers compare new models to state-of-the-art ones and assist in constructing new datasets considering that building datasets from scratch is costly and time-consuming. Only a few studies implemented their DL models into a software tool. For example, [PS16] developed a web application that can detect code clones by uploading code fragments. A command-line tool was implemented in [PS7] to detect code clones. In [PS45], their detection model for the code clone was implemented as a cloud-based web application and a command-line tool.

Future research on DL-based bad smell detection may consider implementing their DL models into automated tools. We suggest that they use more efficient methods other than the command line and web applications. Future works can also implement their detection models as plugin tools that can be used within existing IDEs, such as Eclipse. The use of autonomous bots for the automatic support of software activities is becoming popular in software engineering [26]. We recommend that future studies implement their detection model using software bots that can facilitate the detection process of bad smells.

3.5.9 Large-scale and publicly available datasets (RQ7)

Our SLR shows a lack of labeled bad smell datasets. Although we are not the first to highlight this issue, we emphasize herein the importance of datasets in the era of DL-based models because DL models are known to be data-hungry. Large-scale datasets are required to build robust DL models. Therefore, future research must consider constructing bad smell datasets and making them publicly available to researchers. Most publicly available

datasets are for the code clone smell. Therefore, future studies may want to focus on constructing datasets for other bad smell types, such as Feature Envy, God Class, Long Method, and Data Class. Constructing a dataset for various programming languages (e.g., Python, C, and C#) other than Java is also necessary because they are popular programming languages used by many companies. Finally, the labeling procedures used so far are not unified. Future studies may propose more systematic methods to label bad smells. For example, semi-supervised learning can be utilized to construct labeled datasets [124].

3.5.10 Deep learning model performance (RQ8)

The performance analysis performed in this SLR did not indicate that the investigated factors are necessarily affecting the DL model performance but rather provided insights about possible factors. We highly suggest that future studies investigate the effectiveness of each factor on the DL model performance. This will help derive solid conclusions about the factors that are most influential and need to be considered when in DL model construction.

3.6 Threats to Validity

SLRs are susceptible to various threats that may influence the findings and conclusions of a study [125]. Therefore, potential threats must be identified. The validity threats observed during this SLR are outlined below.

3.6.1 Construct validity

Construct validity refers to the connection between the research hypothesis and the results. Threats that belong to this category are typically research questions, databases used, and search terms.

Search bias. We selected six digital libraries that are well-known in computer science and software engineering. Covering all databases and venues is impractical. We further applied the snowballing procedure to mitigate the risk of missing related papers. We believe that by following these procedures, the most relevant studies were included in this SLR.

Incomplete search term. Constructing a comprehensive and informative search string is not trivial. A missing term might affect the search result. In the context of DL, most studies do not necessarily include the term “deep learning” in the title, abstract, or keywords. They alternatively use the technique name (e.g., LSTM or CNN). Therefore, we included the DL technique names to ensure the search string comprehensiveness. A pilot search was also performed with various search strings to mitigate the risk of missing important terms. Moreover, synonyms and spelling variations were considered in the search string construction.

3.6.2 Internal validity

Internal validity is concerned with the relation between treatment (i.e., set of primary studies included) and outcome. The possible threats related to this category are quality assessment and data extraction.

Subjective quality assessment. The answers to some of the quality assessment questions are not decisive; thus, a subjective assessment might be given to a study based on how the

evaluator interprets it. To mitigate this threat, one author conducted the quality assessment for all studies, and the other two verified the assessment by applying the quality assessment to all studies. The results were then compared. Any disagreement was discussed for resolution.

Bias in data extraction. We created a data extraction form to extract relevant data from the primary studies. A pilot data extraction was performed on 10 studies to ensure that the data extraction items were clear and reflective of the targeted research questions. The data extraction form was then refined. To mitigate the risk of data extraction bias, one author extracted the data, and the other two verified the extracted data of all studies. The data were compared, and any disagreements were discussed and resolved.

3.6.3 External validity

External validity refers to the extent to which the study findings can be generalized. The generalizability and the time span are two issues related to this validity.

Primary study generalizability. The outcome of this study cannot be generalized to other domains. The findings of this SLR might be inaccurate for a detection system relying on DL (i.e., different DL models might yield better detection performances than those analyzed in this SLR when applied to other domains).

Restricted time span. Researchers are usually unable to include related papers outside the time frame specified in the planning phase. DL is a fast-growing area, and new DL techniques are introduced every day. Consequently, a certain DL-based bad smell detection research that succeeded in producing a high detection performance may not have been included because it was not published at the time of writing.

3.6.4 Conclusion validity

Conclusion validity refers to threats that influence the ability to draw the same conclusions if the procedure is repeated. The main issue related to this category is the selection of included studies.

Bias in study selection. The selection of the primary studies might be subjective based on the researcher's judgment. The selection process was performed in two rounds to mitigate this threat. The first round was based on the title and the abstract of the papers. The second round was based on a full-text read. Another researcher was consulted if a decision cannot be made on a particular study. Moreover, a detailed description of the selection process was described, including the research questions, search string, search process, and exclusion and inclusion criteria. The questions related to the quality assessment and the data extraction form were provided.

3.7 Chapter Summary

A comprehensive literature review was conducted to attain the study's objectives and answer the research questions. A total of 67 studies were selected by following a systematic approach and analyzing the inclusion/exclusion criteria and the quality of the studies. Apart from the previously mentioned recommendations and the future research opportunities, the key findings from this SLR are given below based on the research questions:

Bad smells. Bad smells under the code smell category are the most addressed smells. Among them, Code Clone, Feature Envy, and Long Method are the most recurring smells. God Class is the most studied design smell. Most primary studies detected bad smells at

the code level, while a few detected it at the model level. Most studies detected bad smells at the method level.

Framework. The primary objective of using DL models is to explicitly learn the representation of the input feature. Typically, the DL model output is a feature vector. Supervised learning is the most adopted learning approach, followed by the semi-supervised and unsupervised learning approaches.

DL models. Most of the DL models were constructed using only one feature type. Graph-based features are the most common features (e.g., ASTs). Data-related issues are not considered in most of the studies. CNN, ANN, and DNN are the most popular DL models used to build prediction classification models, while the LSTM, Attention model, CNN, and RAE are the most popular DL models used for representation learning. The training hyperparameters are usually neglected.

Evaluation. The train–test split is the most frequently used validation method. Recall, precision, and f-score are the most widely used evaluation metrics. The majority of the studies compared the proposed model with the DL-based ones, while some compared the proposed model with the ML-based models. Tensorflow and Keras are the top two frameworks used for DL model implementation. The source code is usually not publicly available.

Datasets. The majority of the studies evaluated the DL models using one or two datasets. The BigCloneBench, Ojclone, and JEdit datasets are the most widely used datasets in DL-based bad smell detection studies. Java is the most common programming language used for dataset construction. Three main methods are used for dataset labeling: 1) using an

existing dataset containing labels from a reference; 2) automatically using tools or rules to create labels; and 3) manually inspecting the dataset and labeling them accordingly. Most studies used the first approach. The majority of the datasets are not publicly available, and the publicly available datasets are only for the Code Clone smell.

The DL model performance in bad smell detection is influenced by many factors. Specifically, the detection performance of the DL models is associated with the datasets used and the DL modeling techniques. Although most of the DL models achieved a comparative detection performance in terms of recall, most could only achieve low precision. DL-based bad smell detection still remains a challenge, and performance improvements are needed.

CHAPTER 4

CoRT: TRANSFORMER-BASED CODE

REPRESENTATIONS WITH SELF-SUPERVISION BY

PREDICTING RESERVED WORDS FOR CODE SMELL

DETECTION

Various methods to detect smells have been developed [15]. The majority suggested detecting code smells using a series of thresholds dependent on structural parameters (i.e., software metrics) to classify core symptoms that describe specific code smells [16]. Another approach has been proposed to detect code smells by combining multiple structural parameters and formulating a rule that varies depending on the form of the code smell [17]. One major limitation of these methodologies is that their performance depends heavily on threshold values and/or combination rules; until now, there has been no agreement on standard values and rules [18]. To mitigate these limitations, other methodologies, such as ML and DL, have sparked the interest of researchers in recent years [19]. However, one of the major limitations of the current DL-based approaches is that they depend heavily on the manual annotation of data; hence, training an efficient DL model requires a large amount of labeled data, which is both expensive and infeasible. In the context of code smells, large-scale datasets are lacking, which makes it difficult to train DL models.

This leads to a vicious cycle; self-supervised learning can be leveraged to break this cycle. Self-supervised learning is a framework designed to address unsupervised learning problems as a supervised one. The motivation behind self-supervised learning is to learn useful representations of data from unlabeled data for supervised tasks (e.g., classification and regression). The main goal of self-supervised learning is to eliminate the need for manual annotation of data and to improve the generalization of the trained model. This framework has been leveraged for natural language processing (NLP) and image processing. It has been shown that self-supervised learning can improve the performance of classical tasks such as question answering [119], NLP inferencing [87], and object detection [126].

In this chapter, we propose Code Representations with Transformers (CoRT) as a new code smell detection approach that utilizes the self-supervised paradigm. We defined a new proxy task for learning code representations by training transformers to recognize masked reserved words that were applied to the source code received as input. The main advantage of this approach is that it decreases the cost of constructing manually annotated datasets, thereby increasing efficiency, allowing software engineers to devote their time to more important tasks. Moreover, the proposed approach is expected to increase the generalizability of detection models trained using unlabeled data. This study makes the following contributions.

- Novel proxy task for extracting semantic and structural features from source code.
(RQ1)
- The effect of the hyperparameter optimization of the proposed method was assessed, and CoRT was compared with three different DL architectures: Artificial

Neural Network (ANN), Convolutional Neural Network (CNN), and long short-term memory (LSTM). (RQ2)

- An exhaustive evaluation of CoRT on various downstream tasks, that is, the detection of four code smells: Data Class, God Class, Feature Envy, and Long Method. (RQ3)
- The efficiency of CoRT was assessed by evaluating the time complexity over various stages, including pre-training and fine-tuning. (RQ4)
- Comparative analysis of CoRT with supervised learning and feature-based approaches. CoRT demonstrates state-of-the-art results compared to these approaches. (RQ5)
- A concrete illustration of CoRT generalization capabilities for the detection of code smells in unseen datasets was obtained by conducting a cross-project evaluation and comparing the CoRT results with the two paradigms. (RQ6)

The proposed approach enhances the detection performance of code smells compared with supervised and feature-based approaches. We aim to provide valuable insights into the effectiveness and applicability of the CoRT approach for detecting code smells by formulating a set of research questions. The research questions aimed to investigate specific aspects, such as the influence of model architecture and hyperparameter optimization (RQ1 and RQ2), detection performance for class- and method-level code smells (RQ3), time complexity of pre-training and fine-tuning (RQ4), the improvement compared to supervised and feature-based approaches (RQ5), and the generalization capability of unseen projects (RQ6). By examining these aspects, this study sought to validate that CoRT offers improved code smell detection.

The remainder of this chapter is organized as follows: Section 4.1 summarizes the related work. In Section 4.2, we present the dataset construction process. Section 4.3 provides a detailed description of our self-supervised methodology, data extraction, data preprocessing, pre-training, and fine-tuning. Section 4.4 presents the empirical evaluation of our method, including the research questions, dataset construction, and evaluation metrics used. Section 4.5 discusses the detection performance of the proposed approach in detail. Finally, Section 4.6 presents the implications and limitations of this study. Section 4.7 discusses the identified threats to validity and Section 4.8 summarizes the chapter.

4.1 Related Work

The identification of code smells in source code is an active research area in software refactoring. Various detection techniques have been proposed to identify code smells, including metric-based [127], rule-based [17], ML-based [128], and DL-based [24]. In this section, we review studies related to code smell detection. First, we shed light on code smell detection using ML and DL approaches, and then present existing pre-trained models for programming languages.

4.1.1 ML and DL studies

Table 23 summarizes previous ML- and DL-based studies compared with the proposed method. Existing ML-based- and DL-based approaches for code smell detection depend heavily on manually annotated code smell datasets. All previous ML-based studies and some DL-based studies addressed the problem of detecting code smells as a supervised learning problem by utilizing a labeled dataset to train the classifier. Manual annotation is time-consuming and error-prone. Moreover, the problem with supervised learning is that

it requires a labeled dataset and suffers from generalization issues [22], where the model is trained on only a subset of the data (labeled data); hence, it fails to detect unseen instances. Only a few DL-based studies have used DL techniques to learn the deep representations of features. The benefit of representation learning is that it captures the underlying structure and semantics of features. The extracted features were then fed into a typical supervised classifier. In this study, we propose a code smell detection approach using a self-supervised learning approach. Self-supervised learning is a new learning approach that uses unlabeled data. The main advantage of this approach is that it eliminates the need for manually annotated datasets, thus increasing detection efficiency and decreasing cost. Furthermore, it is expected to solve the generalization issues to some extent.

In addition to the learning approach, existing DL-based models have mainly used convolutional neural networks (CNN) or recurrent neural networks (RNN) with their variations of LSTM, GRU, and RvNN models, which have their own limitations. A CNN is known to be efficient and reduces the training time because of the outputs of Conv. The layers are computed in parallel. However, CNN is useful for data that have a spatial relationship, such as images and speech, but it does not perform well on textual data, such as texts and source code, because it fails to capture semantic information. Most studies utilized CNN to extract features from numerical data. Recurrent networks, such as recurrent neural networks (RNN), long short-term memory (LSTM), and gated recurrent units (GRU), are highly suitable for sequential data such as text because of their ability to extract semantic information. However, training recurrent networks is very time-consuming, and many previous DL-based studies have used recurrent networks on a few texts (e.g., method names and class names) rather than the source code.

Moreover, previous studies have extracted features manually using the source code, either by transforming the code into an AST or a statement tree, or by extracting numerical metrics such as software metrics or distance metrics. The main limitations of this approach are as follows: 1) it requires more effort, 2) it may require external tools, and 3) it makes the automation of the detection approach difficult.

Therefore, in this study, we utilized the Transformer model for code representation using a self-supervised learning approach. The Transformer is highly suitable for sequential data. Simultaneously, the outputs of the layers are computed in parallel. Moreover, we directly used the source code to train and evaluate the models.

Table 23 Summary of previous ML- and DL-based approaches

Ref	Code smells	Learning approach	Detection	Independent variables	Preprocessing	Embedding
[129]	Blob, FD, SC, SAK	Supervised	SVM	60 OO metrics	NA	NA
[130]	GC, FE, DC, LM	Supervised	SVM	OO metrics	NA	NA
[131]	Blob, FD, SC	Supervised	AIS	OO metrics	Data reduction	NA
[132]	LM	Supervised	BLR	4 Code metrics	NA	NA
[133]	Blob, AS, CDSP, CC, LC, LZC, LM, LPL, MC, RPR, SG, SAK	Supervised	DT, SVM, BBN	62 Software metrics	Feature selection	NA
[134]	GC, DC, LM, FE	Supervised	SMO, LibSVM, DT, RF, NB, JRip	63-84 OO metrics	Feature selection	NA
[135]	GC, DC, LM, FE	Supervised	SMO, LibSVM, DT, RF, NB, JRip	63-84 OO metrics	Feature selection	NA
[136]	GC, FE	Supervised	NN	6-7 Code metrics	Filtering	NA
[137]	GC, LC, FE, PIH, DC, LZC	Supervised	NN	8 OO metrics	NA	NA
[121]	LM, FE	Supervised	RF, J48	82 Software metrics	NA	NA
[138]	GC, DC, LM, FE	Supervised	NB, KNN, MLP, DT, LR, RF	61-82 Software metrics	Feature selection	NA
[139]	GC, DC, LM, FE	Supervised	J48, JRIP, RF, NB, SMO, Boosting, M-	Software metrics	Class imbalance	NA

				LR, RBFN, LIBSVM/SVR		
[140]	LC, LM	Supervised	DT, RF, LR, SVM, MLP, SGD	Software metrics	Feature scaling, Feature selection	NA
[141]	FE, LM, LC, MC	Supervised	CNN, LSTM, DNN	Textual, Software metrics	Filtering	W2V
[142]	CM, CC, FE, MA	Supervised	CNN, RNN, AE	Source code	Feature selection	NA
[143]	FE	Supervised	MLP, CNN, LSTM	Software metrics, Textual	Class imbalance	W2V
[144]	GC	Supervised	CNN	Software metrics	NA	NA
[145]	GC, DC, FE, LM	Supervised	CNN	Software metrics	Class imbalance	NA
[146]	GC, DC, FE, LM	Supervised, unsupervised	ANN, DAE	Software metrics	Feature selection	NA
[147] [148]	Blob, FE, LM	Unsupervised	LR, VAE	AST, Software metrics	NA	Coding criterion
[149]	GC	Supervised, unsupervised	LSTM, self-attention model, CNN	Textual, Software metrics	NA	W2V
[150]	FE	Supervised, unsupervised	ANN, Bi-LSTM,	Textual, Numerical	NA	W2V
[151]	FE	Supervised, unsupervised	MLP, CNN, LSTM, attention model	Textual, CSD	NA	W2V
[152]	IM, DE, FE, ECB	Supervised, unsupervised	Bi-GRU, RvNN, ASTNN	ST-tree	Class imbalance	W2V
This work	God Class, Data Class, Feature Envy, Long Method	Self-supervised	Transformer model	Source code	Tokenization, Truncation, Filtering, Vectorization, Data balancing	Word + positional embedding
Anti-Singleton (AS), Abstract Syntax Trees (AST), Class Data Should Be Private (CDSP), Complex Class (CC), Complex Conditional (CC), Complex Method (CM), Code Semantic Dependency (CSD), Data Class (DC), Deficient Encapsulation (DE), Empty Catch Block (ECB), Feature Envy (FE), Functional Decomposition (FD), God Class (GC), Insufficient Modularization (IM), Large Class (LC), Long Method (LM), Long Parameter List (LPL), Lazy Class (LZC), Multifaceted Abstraction (MA), Message Chains (MC), Object-oriented (OO), Parallel Inheritance Hierarchies (PIH), Refused Parent Request (RPR), Swiss Army Knife (SAK), Spaghetti Code (SC), Speculative Generality (SG), Statement tree (ST-tree), Word2Vec (W2V)						

4.1.2 Pre-trained Models for Programming Languages

In recent years, there has been growing interest in developing pre-trained models specifically tailored for programming languages. These models leverage large-scale code repositories to learn representations that capture the unique characteristics and semantics

of programming languages. In this subsection, we discuss several notable pretrained models for programming languages, including CodeBERT, CodeT5, and CodeGPT.

CodeBERT [153], proposed by Feng et al., focuses on learning the cross-modal representations of natural language (NL) and programming language (PL) code snippets. It utilizes a transformer-based encoder architecture and is pretrained on a large corpus of NL-PL code pairs. CodeBERT has shown promising results in various code-understanding tasks, such as natural language code search and code documentation generation. GraphCodeBERT [154] extends the CodeBERT model by incorporating dataflow sequences. This model demonstrated an improved performance in code-understanding tasks involving fine-grained structural analysis. Recently, ContraBERT [155], introduced by Liu et al., employed a contrastive learning framework to improve the representation learning of codes and natural languages. They conducted extensive experiments and demonstrated that ContraBERT achieved superior performance compared with other existing pretrained models on various code tasks, such as Code Translation and Code Search, highlighting the effectiveness of contrastive learning in code representation learning.

CodeT5 [156], proposed by Wang et al., was built on the T5 architecture and extended to the programming language. It is specifically designed for code generation and code-understanding tasks. CodeT5 is pretrained on a diverse range of code-related tasks, including code translation, code summarization, and code refinement. This has shown promising results in the generation of semantically correct and syntactically coherent code snippets. CodeT5+ [157] further enhances the capabilities of CodeT5. CodeGPT [158] utilizes a ((Generative Pre-trained Transformer) architecture to pre-train a language model

on a large corpus of code. This model focuses on capturing the sequential patterns and context in code snippets to enable tasks such as code completion and text-code generation. CodeGPT has demonstrated the ability to generate syntactically valid code snippets, and has shown promising performance in code-related tasks. CodeRL [159], proposed by Le et al., combines pretrained models with deep reinforcement learning techniques to enhance code generation. It leverages the power of reinforcement learning to improve the quality and diversity of the generated code.

Although these pretrained models have made significant advancements in code-related tasks, it is important to note that CoRT differentiates itself from these models in terms of its pre-training tasks and focus. CoRT is specifically designed for code smell detection and aims to learn representations that capture code smells at both the class and method levels. Unlike the aforementioned models, CoRT focuses on fine-tuning a pretrained model for code smell detection tasks to enhance its effectiveness in identifying various code smells. These pretrained models provide valuable contributions to the field of programming language understanding and generation. However, the unique design and pre-training task of CoRT highlight its potential to address the specific challenge of code-smell detection, providing a specialized solution that complements existing approaches.

4.2 Datasets Construction

In this section, we discuss the construction of our pre-training and fine-tuning datasets. Unlabeled data (Section 4.2.1) were used in the pre-training phase, whereas labeled data (Section 4.2.2) were used for fine-tuning. A detailed description of the pre-training and

fine-tuning datasets (i.e., per project) can be found in Appendix D. The constructed datasets are available in our online appendix⁴.

4.2.1 Collection of unlabeled datasets

We collected the source code from 103 open-source Java projects. We extracted the data as described in the previous subsection. Table 24 shows the total number of methods (NOM), number of classes (NOC), lines of code (LOC), and time taken to parse the project and extract its methods and classes. Pre-training was conducted with 16M LOC and the data extraction process took 78 h. To increase the generalizability, the data should be representative; therefore, the selection of the projects was based on a set of characteristics, including 1) industry diversity: projects are from different industries, sectors, or application domains to ensure diversity, 2) project size: projects are of varying sizes involve selecting both small-scale and large-scale projects, 3) project maturity: projects at different stages of development, including mature projects with a longer history and emerging projects with ongoing development, 4) open source projects: for transparency, accessibility to data, and the possibility of replication or extension of experiments, and 5) availability of data: the availability of labeled data.

Table 24 Pre-training dataset description

Num. of Projects	NOC	NOM	LOC	Time (Mins)
103	165K	1.3M	16M	78h

⁴ <https://github.com/amalazba/CoRT-Transformer-based-Code-Representations-with-Self-supervision-for-Code-Smell-Detection>

4.2.2 Collection of labeled datasets

We used labeled datasets that are publicly available from previous studies. The selection of the datasets was based on 1) the availability of one of the four code smells considered in this study and 2) the datasets must have a full file path along with the class/method name to match the labels with the extracted code. The labels were collected for subsequent studies.

- Tarek Alkhaeir; Bartosz Walter [160] utilized Eclipse plugin *inCode* for static code analysis that detects code smells; among the code smells, the ones related to our study are Data Class and Feature Envy⁵.
- José Pereira dos Reis et al. [161] manually annotated three code smells (Long Method, God Class, and Feature Envy) using hundred teams⁶.
- Sotto-Mayor et al. [162] utilized the *Organic* plugin to detect the existence of code smells in the source code and detected various types of smells, including Data Class, Large Class, Long Method, and Feature Envy⁷. It is worth mentioning that the original paper detected smells at the file level. To utilize their data, we considered files with only one class because we could easily map the label of the file with the class. In addition, for method-level smells, they provided the ratio of the existence of each method-level code smell within the file, which we labeled as follows:
 - For files with a ratio of 100%, all methods belonging to this file were labeled smelly.

⁵ <https://zenodo.org/record/3992730#.YvsvfuxBxEI>

⁶ <https://zenodo.org/record/6555241#.YvssBuxByi4>

⁷ <https://zenodo.org/record/4103861#.Yvsyu-xByi4>

- For files with a ratio of 0%, all methods that belonged to this file were labeled as non-smelly.
- All files with ratios between 0%-100% were excluded.

Construct new labeled datasets. We constructed fine-tuned datasets using the data collected in the previous steps. First, we created a clean version of each labeled dataset selected in the previous step by 1) unification, 2) removing duplicates, and 3) handling the missing data.

- Unification: we unified the 1) project names by removing “apatch” word, converting to lower case and using “-” to split words projects consisting of multiple words, 2) project versions by removing project names or (e.g., camel1.2.4 to 1.2.4), 3) file path by using dot in the file path instead of slash/backslash and removing the file extension (i.e., .java), 4) class name by converting it to a lower case and extracting the class name in case it is written within a file path, 5) method name by converting it to a lower case and extracting the method name when the full method header is provided, and 6) labels by converting variations of true, false to 1, 0.
- Removing duplicates: In the case of a class or method with different labels, we retain both. However, if we have two instances with exactly the same values, including the label, we retain only one version.
- Missing data: We removed all instances with empty values to match the labeled data with the source code, as we required all the information, including the project name, version, file path, class/method name, and labels.
- The final output is a clean version of the labeled datasets, where each instance has the following attributes: project name, project version, file path, class name, method

name (for method-level smells), smelly (1 for smelly, 0 for non-smelly), and source (used in previous studies to extract the label).

Match labels with source code. It produces a mapping for each class/method for each code smell, whether it is smelly or not. We used regex to map the unlabeled data with their labels. The final output is a set of labeled instances, where each instance has the following attributes: project name, project version, file path, class name, method name (for method-level smells), smelly (1 for smelly, 0 for non-smelly), source (which previous studies used to extract the label), and source code (which contains the class body for class-level smells and method body for method-level smells).

We used labeled datasets only in the fine-tuning phase. The final labeled datasets consisted of 1) the source code in its original format as input, and 2) each class/method associated with a label indicating whether it was smelly or non-smelly. The constructed fine-tuning datasets are presented in Table 25; 97 projects were used. The total number of smelly and non-smelly instances per project for each code smell is listed in the table. It is worth mentioning that some datasets with fewer than six positive instances were not used because SMOTE is based on the K-nearest neighbor algorithm, which generates synthetic instances based on the k-nearest instances. Therefore, we eliminated the datasets with very few labeled instances.

Table 25 Fine-tuning datasets description

	Data class			God class			Feature envy			Long method		
	Smell y	Non- smelly	Total	Smell y	Non- smelly	Total	Smell y	Non- smelly	Total	Smell y	Non- smelly	Total
Total instances	3317	68095	71412	4438	67075	71513	2016	31350	33366	1581	32765	34346

4.3 Approach

This section introduces the implementation details of CoRT. First, a brief overview of the proposed approach is presented, and each step is discussed in a separate subsection.

4.3.1 Overview

Self-supervised learning is a framework designed to address unsupervised learning problems as a supervised one. The motivation behind self-supervised learning is to learn useful representations of data from unlabeled data for supervised tasks (e.g., classification and regression). Supervised learning is used to approximate a target function using labeled data, whereas self-supervised learning does not require labeled data. However, self-supervised learning addresses supervised problems (classification, detection, etc.), similar to supervised learning. This approach can be viewed as an independent form of supervised learning. Unsupervised learning is the process of discovering hidden patterns and associations among data without using human-annotated labels. Both learning paradigms can work with unlabeled datasets. However, these studies address different types of problems. Unsupervised learning is concerned with clustering, ranking, and dimensionality reduction, whereas self-supervised learning is used for classification and detection. The general framework of self-supervised learning involves two steps: learning proxy tasks and applying them to downstream tasks.

1. ***Learning proxy tasks:*** In this phase, a set of tasks (pretext tasks) is designed and formulated based on the problem under investigation. Then, the model learns generalizable representations from unlabeled data by assigning model predesigned

task(s) that are specifically designed to capture implicit information in the source code. In this study, we refer to this phase as the pre-training phase.

2. ***Apply on downstream tasks***: representations will be used to build a supervised model for solving supervised learning tasks, for example, classification, detection, and regression. The model was fine-tuned using a small subset of labeled data, which we refer to as the fine-tuning phase in this study.

Our framework consists of four main steps, as shown in Figure 19: data extraction, data preprocessing, model pre-training, and model fine-tuning. For data extraction, we parsed the source code to extract the classes and methods in each repository and the features that will be used for training. In step two, we applied a set of preprocessing techniques, including tokenization, filtering, truncation, and vectorization, to convert the textual features into a proper format. In the pre-training step, we defined a proxy task and learned a transformer model using the unlabeled data. Finally, in the fine-tuning phase, the pretrained model is applied to multiple downstream tasks, where each downstream task has its own fine-tuned model created by a single pretrained model.

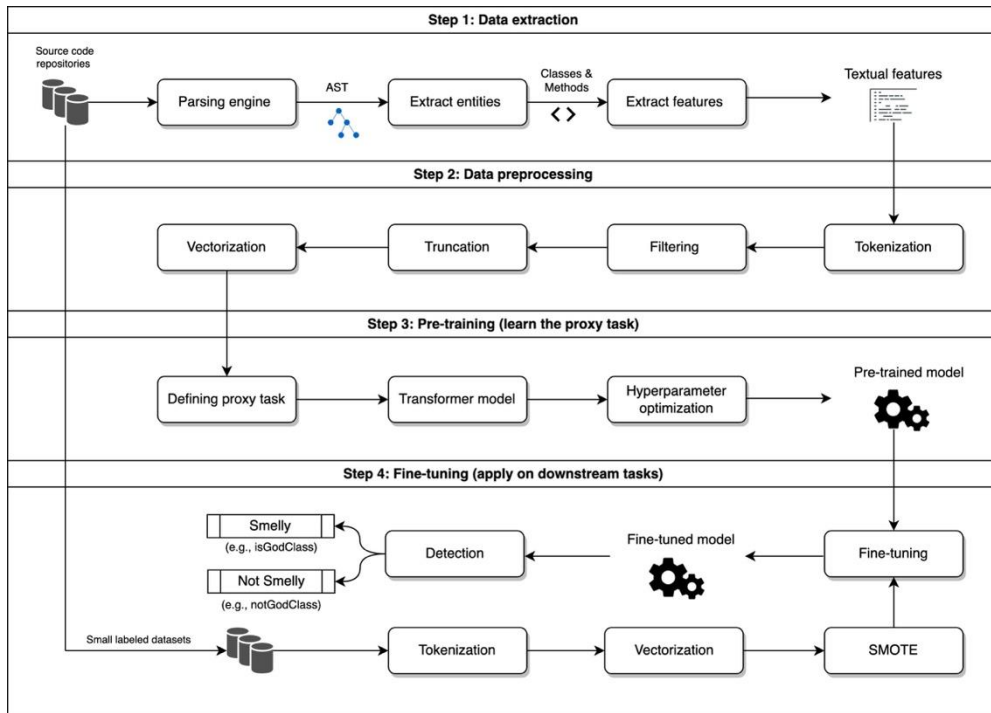


Figure 19 Overview of CoRT

4.3.2 Data extraction

We used ANTLR4 [163] to parse the source code into the AST and then traverse the AST to extract raw classes and methods (i.e., comments were discarded). The following information was extracted: the class name, class body, method names, and method body. Many features can be extracted and used as input, such as software metrics, control flow graphs, or binary code. However, we opted to use textual information (i.e., the code of extracted classes and methods) for two main reasons: it reduces transformation overhead and produces good results, as shown in previous studies [142].

4.3.3 Data preprocessing

For each entity (i.e., class or method), the following preprocessing steps were performed:

- Tokenization: The process of splitting a sentence into words. In this study, the entities were split into lexemes (i.e., tokens).
- Filtering: we removed any entity that has less than 50 tokens [164].
- Truncation: We define a maximum length of 256 tokens; however, some entities may contain thousands of tokens. Therefore, instead of selecting the first 256 and discarding the rest, we truncate a single entity into multiple instances of length 256 [153].
- Text vectorization: A vocabulary of size 30,000 was created in which each distinct token was represented by an integer [119]. Subsequently, all instances were converted from text to numerical values.
- Data balancing: We utilized a Synthetic Minority Oversampling Technique (SMOTE) to balance the labeled datasets.

4.3.4 Pre-training

In this section, we discuss the pre-training step, starting with the proxy task used to train the transformer model, model architecture, and pre-training procedure.

A. Proxy task

The main objective of this study was to detect code smells using unlabeled data. To achieve this goal, we propose CoRT that learns code representations from transformers using unlabeled data. CoRT is trained to predict a set of masked words applied to a part of the program code that is given as an input. Therefore, we must define a proxy task (i.e., the task used for pre-training) that facilitates the learning of intermediate data representations. The ultimate goal of the proxy task is to generate pseudo-labels (i.e., artificial labels). Then, a model is trained to predict these labels to implicitly capture structural and semantic features (i.e., representations) from the source code. There were two steps: 1) pseudo-label creation and 2) pseudo-label training. In this subsection, we focus on generating pseudo-labels, whereas the following subsections discuss the training stages.

The pseudocode in Table 26 summarizes the process of creating the pseudo-labels. The algorithm takes as input 1) an unlabeled dataset $\mathcal{D}$ that consists of m instances, where a single instance i is associated with x value, which is a feature vector of length n tokens. Each instance i represents an entire class/method or part of a class/method. 2) List $\mathcal{R}$ of reserved words in the target programming language. 3) a unique token $MASK_{\mathcal{D}}$. It returns a new dataset $\mathcal{D}^*$ that has m instances, where a single instance i is associated with a tuple (x^*, y) , where x is a new feature vector of length n tokens and y is the pseudo label of length n tokens. The four main steps are as follows:

- Step 1 Get reserved words positions: assume that we have $\mathcal{R}$ reserved words, initialize a new position list $\mathbf{P}$ to track the index of the reserved words; for each instance $\mathbf{i} \in \mathbf{m}$ find the index of token $\mathbf{t} \in \mathbf{n}$ and add it to $\mathbf{P}$ if and only if $\mathbf{t}$ is a reserved word.
- Step 2 Mask reserved words randomly: for each instance $\mathbf{i} \in \mathbf{m}$, create a random True/False array $Mask_{p_i}$ using position list $\mathbf{P}$ created in step 1 and with probability of 25% is True, then for each token $\mathbf{t} \in \mathbf{n}$ replace $\mathbf{t}$ with the $MASK_{ID}$ if and only if it is a reserved word and selected by $Mask_{p_i}$, otherwise keep the original token $\mathbf{t}$.
- Step 3 Mask random tokens: for each instance $\mathbf{i} \in \mathbf{m}$ create a random True/False array $Mask_{Rand_i}$ with probability of 10% is True, then for each token $\mathbf{t} \in \mathbf{n}$ replace $\mathbf{t}$ with the $MASK_{ID}$ if and only if it is selected by $Mask_{Rand_i}$, otherwise keep the original token $\mathbf{t}$.
- Step 4: Construct a new dataset using pseudo-labels: The outputs of steps 1 and 2 are a new feature vector $\mathbf{x}'$. For each instance $\mathbf{i}$ in $\mathcal{D}$, create a new instance i' ($\mathbf{x}^*$, y) such that (1) $\mathbf{x}^* = \mathbf{x}'$ where $\mathbf{x}'$ is the new feature vector created in the previous steps, and (2) pseudo-label $y = \mathbf{x}$ where $\mathbf{x}$ is the original feature vector.

Forcing the learning features: The fundamental reasoning behind masking reserved words rather than only random tokens relates to the simple fact that reserved words play a role in the syntactic structure of programming language grammar. The main objective of the pre-training phase is to have a model with a good representation of the features; by masking reserved words, we force the model to learn to localise/recognise reserved words based on

the context and then eventually learn the structural and semantic features of a programming language being targeted, which will result in a good representation.

Clarity: Furthermore, although program variables, such as identifier name, method name, and class name, play a role in the syntax of a programming language, program variables tend to have various naming conventions; thus, masking random tokens makes the task ambiguous. In contrast, there is usually no ambiguity in predicting masked reserved words.

Simplicity: An additional advantage of the proposed proxy task is that it can easily capture structural and semantic features without many code transformations, such as conversion to binary code, AST, CFG, or using additional tools for extracting software metrics.

Table 26 Create pseudo labels

Inputs: Training data $\mathcal{D} = \{x_i i = 1 \dots m, x_i \in \mathbb{R}^n\}$	
Reserved words list $\mathcal{R} = \{r_i i = 1 \dots L\}$	
Mask token id = $MASK_{ID}$	
1:	Step 1: Get reserved words positions
2:	$P = \{\}$ // empty list
3:	for $i \leftarrow 1$ to M do
4:	for $t \leftarrow 1$ to N do
5:	$P_i = \{\text{index}(x_{it}) \text{ if } x_{it} \in \mathcal{R}\}$
6:	$P = P \cup P_i$
7:	end for
8:	end for
9:	Step 2: Mask reserved words randomly
10:	for $i \leftarrow 1$ to M do
11:	$Mask_{p_i} = \begin{cases} \text{True} & \text{if } \text{random}(1, \text{len}(P_i)) < 0.25 \\ \text{False} & \text{if } \text{random}(1, \text{len}(P_i)) \geq 0.25 \end{cases}$
12:	for $t \leftarrow 1$ to x_i tokens do
13:	$x'_{it} = \begin{cases} x_{it} & \text{if } \text{index}(x_{it}) \notin P_i \\ MASK_{ID} & \text{if } \text{index}(x_{it}) \in P_i \text{ and } Mask_{p_i} = \text{True} \end{cases}$
14:	end for
15:	end for
16:	Step 3: Mask random tokens
17:	for $i \leftarrow 1$ to M do

```

18:    $Mask_{Rand_i} = \begin{cases} True & \text{if } random(1, len(x_i)) < 0.10 \\ False & \text{if } random(1, len(x_i)) \geq 0.10 \end{cases}$ 
19:   for  $t \leftarrow 1$  to  $N$  do
20:      $\hat{x}_{it} = \begin{cases} x_{it} & \text{if } Mask_{Rand_{it}} = False \\ MASK_{ID} & \text{if } Mask_{Rand_{it}} = True \end{cases}$ 
21:   end for
22: end for
23:   Step 4: Construct a new data set using with pseudo-labels
24:    $\mathcal{D}^* = \{x_i^*, y_i \mid i = 1 \dots m, x_i^* = \hat{x}_i \text{ and } y_i = x_i\}$ 

```

Output: Training data $\mathcal{D}^*$

B. Training procedure

The model architecture of the CoRT is the multi-layer transformer proposed by Vaswani et al. [165]. In addition to the transformer, we considered the CNN, LSTM, and ANN models for comparison. These models were selected because they have been frequently utilized and have shown promising results [24]. Although CNNs are typically applied for feature extraction from numerical data and LSTMs are commonly used with sequential data, their selection is primarily driven by their prevalence in the existing literature. The transformer has several advantages over CNN and LSTM. It is similar to CNN, as its computation can be performed in parallel, and similar to LSTM, which can process sequential data, such as text and source code.

Hyperparameter optimization. Hyperparameter optimization was performed to select the number of layers (L), embedding dimension (E), number of units of hidden layers (H), and number of self-attention heads (A). We used the Optuna library to perform hyperparameter optimization [166]. We employed the tree-structured Parzen estimator (TPE) as the parameter-sampling algorithm [167]. We opted to use this method because other methods, such as grid search, are computationally expensive in the DL context. Moreover, random search is a well-known algorithm that has been shown to produce results almost identical

to those of grid search but much faster [122]. TPE is extremely computationally efficient compared to a random search.

Table 27 presents the search spaces used in this study. Hyperparameter optimization was performed on all the four models (Transformer, CNN, LSTM, ANN). Note that the number of heads is solely a hyperparameter of the transformer model and is not considered in other models. The number of layers in a transformer represents the number of transformer blocks. In addition, we searched for the best training hyperparameters, including batch size (B), learning rate (L), and dropout rate (DR). We ran 100 trials (25 trials for each model) for approximately five weeks of running.

Table 27 Hyperparameter optimization search space

Hyperparameter	Values
Model architecture	
Number of layers (L)	3, 6, 12
Embedding dimension (E)	128, 256, 512
Number of units of hidden layers (H)	128, 256, 512, 1024, 2048
Number of self-attention heads (A)	8, 16
Model training	
Batch size (B)	32, 64, 128, 256
Learning rate (L.R)	0.0001, 0.001, 0.01
Dropout rate (DR)	0.1, 0.2, 0.3

Pre-trained model architecture. Figure 20 shows the architecture of the CoRT model after hyperparameter optimization. The base model has 12 L, 256 E, 512 H, 12 A. We also report the results on two model sizes for comparison purposes: a small model with 12 L, 256 E, 512 H, 12 A, and a large model with 12 L, 256 E, 512 H, 12 A. For other models, the best hyperparameters for the model architecture were as follows: CNN model with 12 L, 256

E, 512 H. LSTM model with 12 L, 256 E, 512 H, and ANN model with 12 L, 256 E, 512 H.

Multi-head self-attention. The first main layer of the Encoder is a multi-head self-attention layer that receives multiple versions of the queries, keys, and values, each to produce self-attention outputs simultaneously, which are then used to produce the final results using the following equation:

$$MultiHead(Q, K, V) = concat(head_1, head_2, \dots, head_h)W^o$$

Where Q, K, and V are queries, keys, and values, respectively; W^o is the weight matrix; $i = 1 \dots h$ is the number of heads, where each head is a self-attention representation:

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^W)$$

where QW_i^Q, KW_i^K, VW_i^W are the weight matrices that create multiple subspace representations of the query, key, and value matrices, respectively, and are calculated as follows:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

where d_k is the scaling factor used to mitigate the vanishing gradient problem, which is equal to the number of embedding dimensions divided by the number of heads. In this study, the number of heads is denoted as A, and d_k is set to 256 because E=512 and A=8. This first multi-head self-attention layer on the decoder side is similar to that implemented in the encoder; however, the second multi-head self-attention layer on the decoder obtains

the keys and values from the encoder's output and the queries from the preceding embedding layer.

Feed-forward network. The second main layer is a fully connected feedforward neural network comprising two dense layers with Rectified Linear Unit (ReLU) activation.

$$FFN(x) = ReLU(W_1x + b_1)W_2 + b_2$$

Normalization [168]. Each layer is followed by a normalization layer, *layernorm(.)* which normalizes the sum by re-centering and rescaling the layer input x and the output of the layer itself, that is, $layer(x)$:

$$layernorm(x + layer(x))$$

The computed sum is normalized using the following equation:

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu^{(k)}}{\sqrt{(\sigma^{(k)})^2 + \epsilon}}$$

Where $k = 1 \dots d$ (i.e., embedding dimension), $i = 1 \dots m$ (instances in training dataset), μ is the mean, σ is the standard deviation, and epsilon ϵ is an arbitrarily small constant set to $1e-6$ in this work.

Residual connections [169]. Finally, the layers were connected via residual connections (blue connections). The main advantages of these connections are 1) to remind the contextual representations of the original input tokens as deep neural networks might "forget" some features, and 2) to mitigate the vanishing gradient problem by summing up the values of the input to the output result of the layer in the following way:

$$layer(x) + x = [W_2\alpha(W_1x + b_1) + b_2] + x$$

Where x is the input, $layer(x)$ is the output generated by the previous layer, W_2 and b_2 are the weight and bias for the current layer, W_1 and b_1 are the weight and bias for the previous layer, and α is the activation function.

Input/output representations. The input is a sequence of codes that are either a class, a method, or a subsequence of a class/method. The representation of each token in a sequence is the summation of the token and positional embeddings. We utilized the word embedding provided by TensorFlow⁸ a vocabulary size of 30,000 tokens and a set of special tokens ([unk] to represent out of vocabulary tokens, and [mask] to represent masked tokens). In token embeddings, each token (represented as a positive integer) is mapped to a dense vector of fixed size (i.e., embedding dimension). Positional embedding is the unique representation of the position of each token within a sequence. Two mathematical functions, sine and cosine, are used to construct positional embedding vectors, which have the same dimensions as the token embeddings, using the following equations:

$$PE_{(p,2i)} = \sin(\frac{p}{n^{2i/d}})$$

$$PE_{(p,2i+1)} = \cos(\frac{p}{n^{2i/d}})$$

Where p is the position of the token, i is the column index $0 \leq i < d/2$, d is the embedding dimension, and n is a user-defined scalar, set to 10,000 as recommended in the original

⁸ <https://www.tensorflow.org/>

paper [165]. The sine function was used to encode the even position ($2i$), whereas $\cos$ encodes the odd positions ($2i + 1$).

The output of the model is the prediction of the masked token value, which is a probability matrix of size (sequence length $\times$ vocabulary size), calculated using the *softmax* function:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^k e^{x_j}}$$

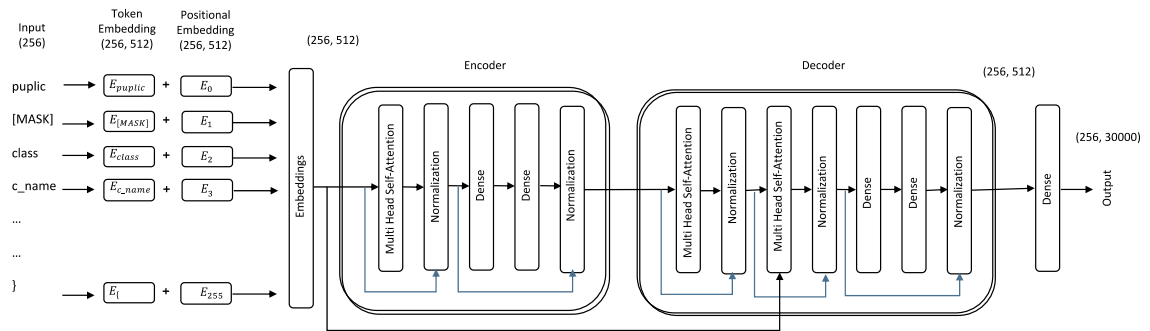


Figure 20 CoRT model architecture

Model training. We trained our model with a batch size of 64 instances; each instance had a maximum length of 256 for five epochs (55,000 steps). We used Adam as the optimization algorithm with a learning rate of 0.001, which is the most common learning rate utilized in the literature [24]. Also, we performed hyperparameter optimization and the best results were achieved with a 0.001 learning rate. A Xavier uniform initializer was used to initialize the weights. Finally, a dropout rate of 0.1 was used on all layers. The basic objective of the network training process is to reduce losses, either in terms of errors or cost. In our model, we use a categorical cross-entropy function with sample weights defined as follows:

$$L(\theta) = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^m w_{ij} (y_{ij} \log(\hat{y}_{ij}))$$

Where n is the number of masked tokens across instances, m is the total number of classes, w is the sample weight, y is the true pseudo label, and $\hat{y}$ is the predicted pseudo label. In this study, the classes were all vocabulary tokens (i.e., 30000 classes). The weight is created as follows: a sequence of code with length 256 is given; a token is given a weight of 1 if it is a mask token, and 0 otherwise. The validation method used for model evaluation is discussed in Section 4.4.3.

4.3.5 Fine-tuning

The pretrained parameters are utilized to initialize the fine-tuned model; subsequently, the parameters are tuned using the labeled datasets. The pre-trained model is fine-tuned for each downstream task using its own labeled datasets; therefore, it will have its own fine-tuned model. Although each downstream task is fine-tuned individually, they are all

initialized with the same parameters, which has two main advantages: 1) fine-tuning does not require a large amount of data (i.e., it can be fine-tuned on small, labeled datasets), and 2) it speeds up the training as the fine-tuning process is usually very fast compared to the pre-training that has been trained once and then used multiple times on various downstream tasks.

Downstream tasks. The proposed approach detects smells at both the method and class levels. We selected two class-level code smells—God Class and Data Class—and two method-level code smells—Feature Envy and Long Method—as downstream tasks. The selection was based on the fact that these code smells are highly important (i.e., they have a major impact on software quality) and are more prevalent in software programs [117].

Fine-tuned model architecture. We implemented an ANN, which is a simple feedforward neural network comprising of two layers.

Input/Output Representations. The input was either a class for class-level code smells or a method for a method-level code smell. The output is 0 or 1, indicating whether a class/method is smelly, as calculated using the *sigmoid* function

$$\text{sigmoid}(x_i) = \frac{1}{1 + e^{-x_i}}$$

Fine-tuning procedure. We fine-tuned our model with a batch size of 16 instances, where each instance had a maximum length of 256 for five epochs (55,000 steps). We used Adam as the optimization algorithm with a learning rate of 0.001. We utilized a pretrained model to initialize the weights. Finally, a dropout rate of 0.1 was used on all layers. For the fine-tuned models, we used the binary cross-entropy function as follows:

$$L(\theta) = -\frac{1}{n} \sum_{i=1}^n y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)$$

Where n is the number of instances, y is the true label, and $\hat{y}$ is the predicted label.

Model built for comparison. 1) Supervised learning: We compared the proposed approach to a supervised learning approach. We trained a baseline model in which the model architecture was a simple ANN model with the same hyperparameters used for fine-tuning; that is, we used Adam as an optimization algorithm with a learning rate of 0.001. We utilized the Xavier uniform initializer to initialize the weights and a dropout rate of 0.1. labeled datasets were used to detect the four code smells. The input/output representations, as well as the training procedure used to build the model, are similar to those used in fine-tuning. 2) Feature-based: We also compared our approach with the feature-based approach, where we used the extracted features from CoRT as input to the ANN model without any fine-tuning and then used labeled datasets to make predictions. The ANN model was trained using the same input/output representations, training procedures, and hyperparameters as those used for the fine-tuning and baseline models. In our study, both feature-based and baseline approaches were developed using an Artificial Neural Network (ANN) model. This choice was made to ensure a fair and consistent comparison, because we utilized an ANN as the underlying model for fine-tuning. By employing the same model architecture, we aimed to isolate the impact of different input data sources rather than variations in the model types.

4.4 Experiments and evaluation

This section presents the details of the experiments conducted to evaluate the proposed approach for various code-smell detection tasks. The experiment planning outline for evaluating the CoRT is as follows:

- 1) Defining the goal and RQs: We defined a set of RQs in section 4.4.1.
- 2) Datasets Preparation: Details of the used datasets are in section 4.2. The independent variables are the source code, while the independent variable is smelly or non-smelly. We preprocessed the data as described in section 4.3.3.
- 3) Performance Metrics: We selected a set of metrics listed in section 4.4.2.
- 4) Validation Method: We used a train-test-split method with an 80/20 ratio for pre-training and a 70/30 ratio for fine-tuning more details are in section 4.4.3.
- 5) Baseline Comparisons: We compared our approach with two approaches supervised and feature-based approaches (section 4.3.5). Also, we compared our pre-trained model with existing pre-trained models for code representation (section 4.1.2).
- 6) Hardware and Software: We used a PC with the following specifications: two Quadro M6000 (each one has 3072 CUDA cores with 24 GB GPU memory). We implemented our approach using Python. We used the TensorFlow framework for developing and training the deep learning models.
- 7) Data Analysis: Finally, we analyzed the collected results using various methods such as tables, boxplots, and heatmaps.

4.4.1 Research questions

The main goal of the experiment was to evaluate the CoRT for detecting God Class, Data Class, Feature Envy, and Long Method code smells. We formulate the following research questions:

- RQ1: How does the models' architecture influence the pre-trained model performance of CoRT?

The main architecture of CoRT is a Transformer; however, other architectures that have been widely used in the literature, such as CNN, LSTM, and ANN, have produced good results [24]. We used various architectures for the pretrained model, and the detection performances of these models were investigated and compared in this RQ.

- RQ2: How does hyperparameter optimization affect the performance of CoRT?

In this RQ, we investigated how the model size influences the detection performance and computational time. Moreover, we show how training hyperparameters influence the detection performance and computational time.

- RQ3: What is the detection performance of CoRT for identifying class-level and method-level code smells?

This RQ investigated the performance of the proposed approach in detecting two class-level code smells—God Class and Data Class, in terms of accuracy, precision, recall, F1-score, and AUC. The detection performance was compared with that of state-of-the-art DL-based approaches. We investigated the performance of the proposed approach in detecting two method-level code smells: Feature Envy and

Long Method. The detection performance was compared with that of state-of-the-art DL-based approaches.

- RQ4: How efficient is the proposed approach?

Model complexity is another significant factor to consider when choosing a model, in addition to model performance [123], [170], because additional resource consumption (e.g., storage and computational time costs) may be necessary for complex models. DL models are complex; hence, time complexity is an important metric to consider when evaluating DL models. One of the main advantages of self-supervised learning is its capability to train one complex DL model (which takes a very long time to train) that can be used multiple times in other downstream tasks by fine-tuning the pretrained model (which is fast to train). Therefore, this RQ quantitatively reveals the difference between the pre-training and fine-tuning times of the proposed approach.

- RQ5: To what extent does CoRT enhance the detection performance compared to the feature-based approach?

Many previous studies have generated features using unsupervised learning and then utilized these features as inputs to a supervised classifier. In this RQ, we compared this approach (i.e., feature-based approach) to our approach (i.e., fine-tuning).

- RQ6: How well does CoRT generalize on unseen projects?

To evaluate the robustness and generalizability of CoRT, we tested the proposed approach for unseen projects. In particular, we considered models that have been

trained on 100 Java projects, and then tested the detection performance of the model on new Java projects that have not been used in either pre-training or fine-tuning.

The first two RQs (RQ1 and RQ2) are related to the first part of the proposed method, which is pre-training, whereas RQ3 is related to fine-tuning. In RQ4, we contrast the time complexity of the two parts, that is, pre-training and fine-tuning. In RQ5 and RQ6, we performed two ablation studies.

4.4.2 Evaluation metrics

To evaluate the detection performance of our models, we calculated metrics frequently used in DL-based code smell detection. In code smell detection, the capability of the model to detect smelly instances is crucial; hence, using only the accuracy is not sufficient. Therefore, we used five metrics along with accuracy. However, for RQ1 and RQ2, we solely used accuracy and loss because these RQs are related to the pre-training phase, where we evaluated the overall performance of the representation, which is the result of training the model on the proxy task. Therefore, we utilized the accuracy (defined below) and loss (defined in Section B) to demonstrate the overall performance of the pre-training phase. For RQ3, RQ5, and RQ6, the evaluation metrics used to measure the detection performance are as follows:

- Accuracy: The percentage of instances correctly detected was calculated as follows:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \times 100$$

- Precision (also known as the positive predictive value) is the percentage of retrieved instances that are relevant.

$$Precision = \frac{TP}{(FP + TP)}$$

- Recall: (also called sensitivity) is the percentage of retrieved relevant instances and is defined as follows:

$$Recall = \frac{TP}{(FN + TP)}$$

- F1-score: is the harmonic mean of both precision and recall, calculated as:

$$F - measure = 2 \times \frac{precision \times recall}{precision + recall} \times 100$$

- Matthews Correlation Coefficient (MCC) computes the correlation coefficient (i.e., measures the strength of a correlation) between the true and predicted classes. The MCC can range from 1 to -1 where a value of 1 denotes the best classifier, -1 denotes complete inconsistency, and 0 denotes a random prediction. MCC is defined as:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

- Area Under Curve (AUC) is a measure of how much of a region is below the receiver operating characteristic (ROC) curve. The AUC ranged from 0 to 1, with a value of 0 denoting the worst classifier and 1 denoting the best classifier. A classifier with an AUC value of 0.5 resembles a classifier that makes guesses at random.

Where TP is the number of smelly instances correctly detected, TN is the number of non-smelly instances predicted as non-smelly, FN is the number of non-smelly instances

incorrectly predicted as smelly, and FP is the number of non-smelly instances incorrectly detected as smelly.

4.4.3 Validation method

We used the *train-test-split* method to validate our models, in which the data were randomly split into two sets: one used for training and the other for testing. We used a ratio of 80/20, where 80% of the data were used for training and 20% for testing. This method was used to evaluate the pretrained models. For fine-tuning, we used a ratio of 70/30, where 70% of the data were used for training and 30% for testing. For pre-training, the unlabeled dataset was constructed using 165 K Java classes. Therefore, the training set contains approximately 132 K classes while the testing contains 33 K classes. For fine-tuning (i.e., labeled datasets), the training and testing sets differed based on the number of instances for each project; for instance, the Accumulo project in the Data Class smell has 2250. Therefore, for this fine-tuning dataset, the training set had 1575 instances, whereas the testing set had 675 instances.

The selection of the ratio was based on two facts: 1) the two ratios are the top two validation methods according to a recent review [24], and 2) we considered the number of instances in the datasets, so for pre-training, 20% for validation was sufficient; however, in fine-tuning, the datasets were considered relatively small; therefore, we selected 30% of the data for validation.

4.5 Results and Discussion

In this section, we discuss the results obtained using the proposed approach with respect to the RQs defined in Section 4.4.1.

4.5.1 RQ1 How does the models' architecture influence the performance of CoRT?

In this RQ, we evaluated the performance of CoRT compared with three architectures: CNN, LSTM, and ANN. These models were trained on the proxy task defined in Section A and using the following hyperparameters: 128 E, 32 B, 0.001 LR, 0.1 DR, 128 H, 3 L, 8 A. Starting by analyzing the accuracy and loss for the pre-trained models, shown in Figure 21 and Figure 22; we observe that the best results were obtained by CoRT with an accuracy of 0.75 and loss of 0.2, followed by the CNN with an accuracy of 0.71 and loss of 0.33, on the validation set. The worst results were obtained by the ANN, with an accuracy of 0.27 and a loss of 0.83. The second worst algorithm was LSTM, with an accuracy of 0.62 and a loss of 0.38.

RQ1 Answer. CoRT outperformed the other architectures, demonstrating its effectiveness in achieving higher accuracy and lower loss for the given task. CNN also exhibited promising performance, whereas LSTM and ANN exhibited comparatively lower accuracy and higher loss.

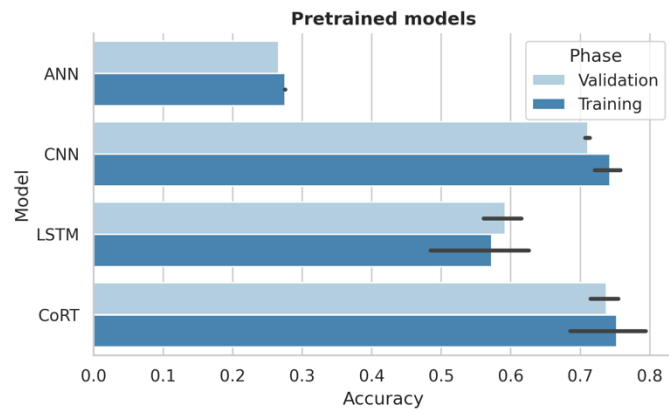


Figure 21 Pre-trained models accuracy

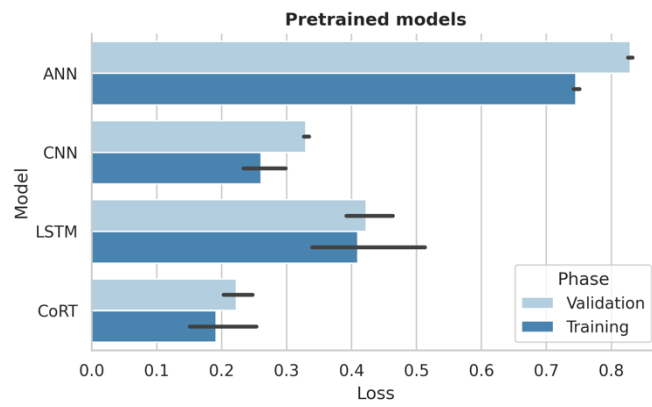


Figure 22 Pre-trained models loss

4.5.2 RQ2 How does hyperparameter optimization affect the performance of CoRT?

The results of hyperparameter optimization are presented in this section. For each pretrained model (i.e., CoRT, CNN, LSTM, and ANN), we performed hyperparameter optimization using Optuna. First, we discuss the effect of training hyperparameters on the loss (Figure 23), and then we present the effect of model size on the loss (Figure 24). Finally, we discuss the optimal hyperparameters for each model.

Figure 23 provides a comprehensive overview of the effects of the different training hyperparameters on the performance of the model. Figure 23 A presents a parallel coordinate plot that allows us to analyze the relationship between the performance of the ANN model (i.e., loss) and the three training hyperparameters (dropout rate, learning rate, and batch size). Each hyperparameter has a range of values, as specified in Table 27. The parallel coordinate plot consisted of three vertical axes representing each hyperparameter, and the fourth and last axes represented the loss values. The plot includes multiple lines, each representing a different combination of hyperparameter values and their corresponding loss values. By examining the lines in the plot, we can observe the impact of different hyperparameter settings on model performance. Lines closer to the bottom of the last axis indicate lower loss values, indicating a better model performance. In contrast, lines closer to the top of the last axis represent higher loss values, indicating poorer model performance. We observed that the ANN model was the least affected by hyperparameter optimization, whereas the CoRT was sensitive to hyperparameter optimization, as shown

in Figure 23 and Figure 24. CNN and LSTM were less sensitive to hyperparameter optimization than CoRT.

We plotted the training hyperparameters (i.e., learning rate, batch size, and dropout rate) against the loss in Figure 23 to show the effect of these hyperparameters on the model loss. We observed that a small learning rate with a small batch size produced good results, as shown in Figure 23 B, C, and D. Figure 24 shows the model loss against the model size (i.e., the embedding dimension, number of layers, number of hidden units, and number of attention heads). As shown in Figure 24, deep models ($L > 3$) performed well with smaller embedding and layer dimensions, whereas shallow models ($L = 3$) required larger embedding and layer dimensions.

Table 28 presents the best hyperparameters for each model along with the loss before optimization. Although all models achieved better performance with a decrease in model loss, CoRT maintained the lowest loss value of 0.148. Similarly, CNN was the second-best model, with a loss of 0.163; ANN achieved the worst results, with a loss of 0.584; and LSTM was the second-worst model, with a loss of 0.192. The best CoRT hyperparameters were as follows: 0.3 DR, 0.001 LR, 128 B, 6 L, 128 E, 2048 H, and 8 A, which were selected as the pretrained models to perform fine-tuning on the labeled datasets.

RQ2 Answer. CoRT showed notable sensitivity to hyperparameter adjustments, whereas ANN had the least impact. Smaller learning rates and batch sizes resulted in better performance across the models. Deep models with smaller embedding and layer dimensions performed well, whereas shallow models required larger dimensions. CoRT achieved the lowest loss of 0.148 after hyperparameter optimization, followed by CNN

with a loss of 0.163. Conversely, ANN performed the poorest, with a loss of 0.584. The optimized hyperparameters for CoRT included a dropout rate of 0.3, learning rate of 0.001, batch size of 128, six layers, 128 embedding dimensions, 2048 hidden units, and eight attention heads.

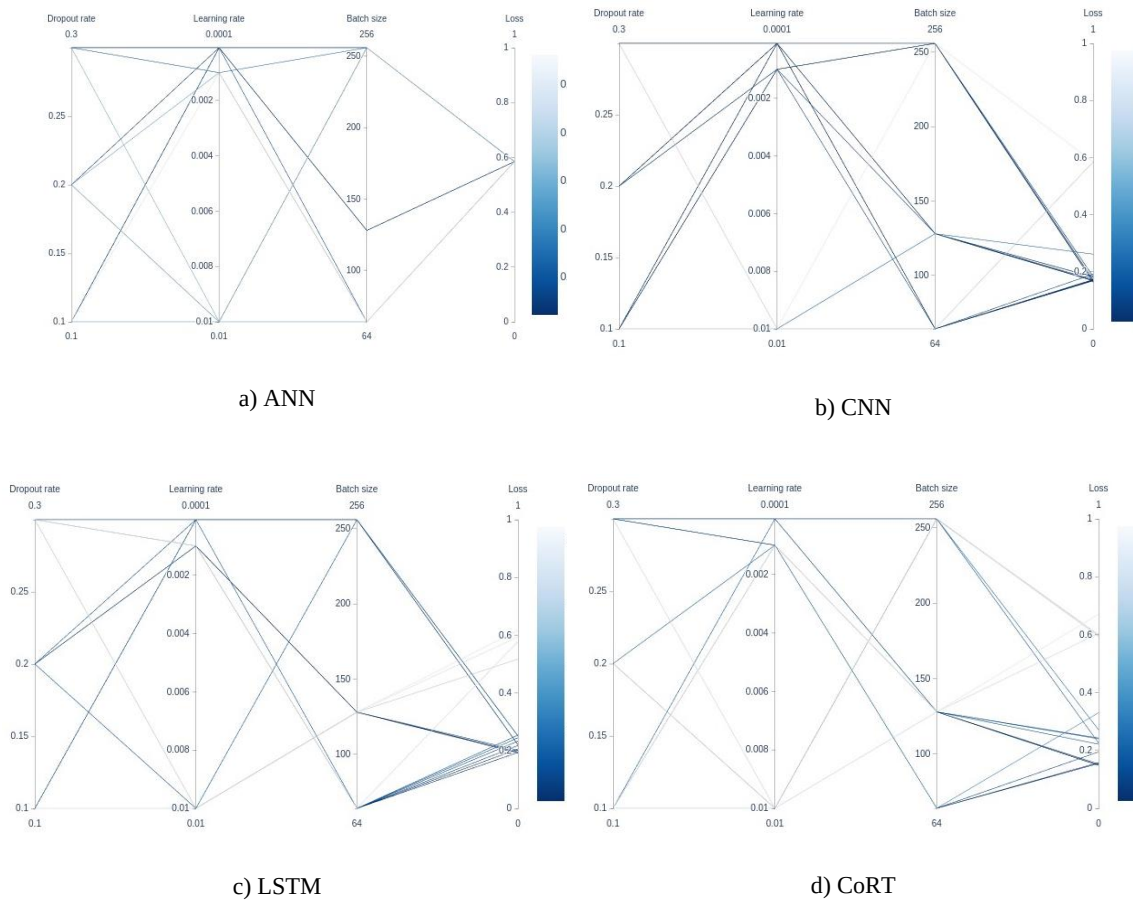
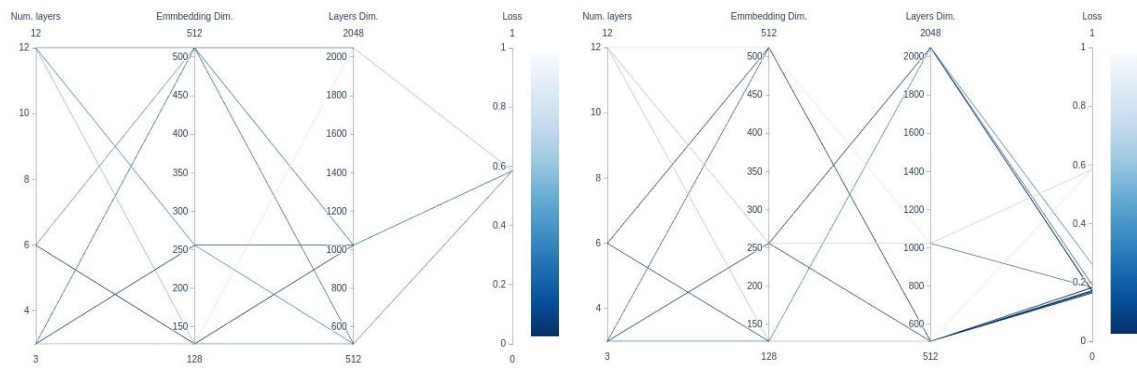
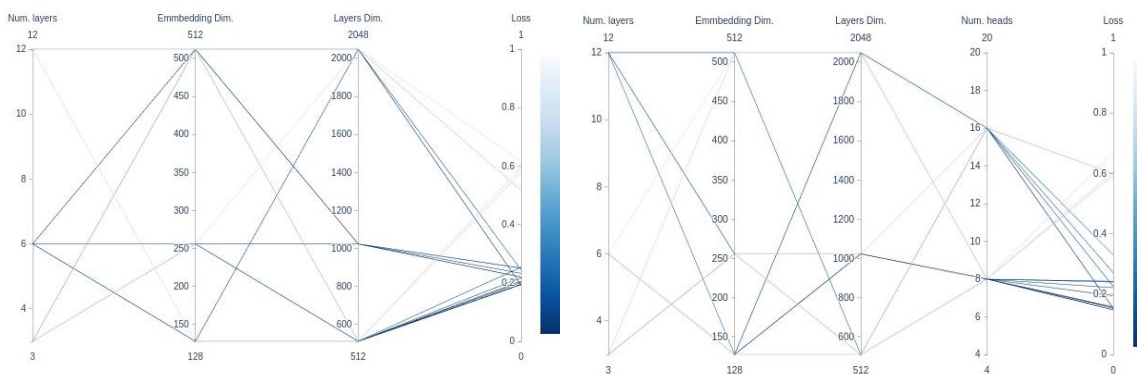


Figure 23 Effect of training hyperparameters on model loss



a) ANN

b) CNN



c) LSTM

d) CoRT

Figure 24 Effect of model size on model loss

Table 28 Pre-trained model's best hyperparameters

Pre-trained model	Dropout rate	Learning rate	Batch size	Num. layers	Embedding Dim.	Layers Dim.	Num. heads	Loss	Loss (before)
ANN	0.2	0.0001	256	6	128	1024	-	0.584	0.834
CNN	0.1	0.001	256	6	512	512	-	0.163	0.329
LSTM	0.3	0.001	128	3	512	2048	-	0.192	0.382
CoRT	0.3	0.001	128	6	128	2048	8	0.148	0.199

4.5.3 RQ3 What is the detection performance of CoRT for identifying class-level and method-level code smells?

In this section, we discuss the detection performance after fine-tuning CoRT for four downstream tasks. We decided to use the F1, AUC, and MCC as our main metrics; however, ACC, P, and R are also presented in the tables. The detection performances of the two class-level code smells (i.e., Data Class and God Class) are presented in Table 29 and Table 30.

The detection performance of CoRT on Data Class achieved a score of F1 between 88.08-99.4, AUC between 89.62-99.88, and MCC between 75.28-98.8, while on God Class it achieved a value of F1 ranges from 86.32-99.03, AUC of 92.1-99.85, and MCC of 76.15-98.09. We observed that CoRT performed well on some datasets (among the top 10 datasets) in the two class-level smells, Jena and Knox, while exhibiting the worst performance on other datasets (among the bottom 10 datasets) across both class-level smells, Archiva and Struts. However, there was a contradiction in the performance in a few datasets; for instance, CoRT performed well on detecting Data Classes in the Atlas dataset but failed to maintain compatible performance in detecting God Class smell for the same dataset. In contrast, it achieved good performance for detecting the God Class in Fob and

Tomcat, but performed poorly for detecting the Data Class for the same datasets. A possible explanation is the project size; we noticed that both Jena and Knox are large projects with more than 100 K LOC, whereas Archiva and Struts are both small projects with LOC less than 100 K. Furthermore, we observe from the table that the detection performance is not sensitive to the number of positive (smelly) instances, whereas in the Atlas dataset, the model performed well in detecting the Data Class, although it had only 18 smelly instances, whereas it performed poorly in detecting the God Class, which had almost three smelly instances (i.e., 52).

Table 31 and Table 32 present the detection performance of the two-method-level code smells, Feature Envy, and Long Method. The Feature Envy detection performance obtained by CoRT achieved an F1 between 40.15-99.31 (Avg. 88.68), AUC between 67.35-99.97 (Avg. 92.4), and MCC between 14.91-98.64 (Avg. 78.34). For the Long Method, the model obtained an F1 between 28.57-99.87 (Avg. 88.87), AUC between 25-100 (Avg. 90.2), and MCC between -25.82-99.75 (Avg. 77.16). Similar to class-level smells, CoRT exhibited good detection performance across the two method-level smells on some datasets, such as Activemq and Nifi, but the worst detection performance on other datasets, such as Accumulo, Airavata, Jasml, Storm, and Zeppelin. We observe that CoRT performed well on larger projects, where Activemq and Nifi have 300 K, and 550 K LOC, respectively, whereas low detection performance scores were obtained for smaller projects, such as Jasml, Storm, and Zeppelin, with LOCs of 6k, 200k, and 60k, respectively. An exception to this is the Accumulo and Airavata datasets, both of which have 400 K LOC; however, the model achieved low performance. On one dataset, Cocoon, the model performed well on one code smell (Long Method), while achieving low performance on the other smell

(Feature Envy). Although the Cocoon dataset had the same number of smelly instances (12) for both method-level smells, the model performed well on one smell and failed to maintain a similar performance on the other, which supports the claim that the detection performance of the CoRT model is not sensitive to the number of smelly instances, as observed in the class-level smells.

After examining the detailed results, we observe that both AUC and MCC demonstrate high values across most datasets, suggesting a level of consistency in their evaluation of the model's performance. For instance, datasets like 'accumulo', 'activemq', and 'jena' exhibit high scores in both metrics for detecting Data Class smell, indicating not only an excellent capability of the model to distinguish between classes (as shown by AUC) but also a strong correlation between predictions and actual classifications, even in balanced scenarios (as indicated by MCC). However, in a few datasets, such as 'roller-weblogger', there is a noticeable divergence between AUC and MCC values. In these cases, despite high AUC scores, the MCC values are relatively lower. This discrepancy suggests that while AUC and MCC are generally consistent in assessing model performance, their correlation is not absolute and may vary depending on specific dataset characteristics. Therefore, it's important to consider both metrics together for a more comprehensive evaluation of the model's effectiveness.

Table 29 Data Class code smell detection performance

Data Class													
Dataset	ACC	P	R	F1	AUC	MCC	Dataset	ACC	P	R	F1	AUC	MCC
accumulo	99.4	99.7	99.1	99.4	99.74	98.8	karaf	90.8	99.43	82.08	89.93	91.82	82.88
activemq	98.4	99.7	97.08	98.37	99.5	96.83	kerby	96.37	93.89	99.19	96.47	99.11	92.89
airavata	97.67	96.77	98.63	97.69	99.88	95.36	knox	98.91	98.91	98.91	98.91	99.69	97.83
archiva	95.35	100	90.7	95.12	96.74	91.09	kylin	95.63	97.72	93.45	95.54	98.69	91.35
atlas-sources	98.94	100	97.89	98.93	99.13	97.91	manifoldcf	94.82	100	89.64	94.54	94.68	90.13
beam	97.14	99.83	94.44	97.06	99.51	94.41	maven	92.73	92.73	92.73	92.73	97.05	85.45
calcite	97.62	99.55	95.67	97.57	99.11	95.31	nifi	97.78	98.25	97.28	97.76	99.16	95.55
camel	98.51	99.62	97.4	98.5	99.67	97.05	ofbiz	94.15	92.78	95.74	94.24	98.4	88.34
carbondata	95.64	96.45	94.77	95.6	98.49	91.29	olingo-odata	95.72	97.77	93.58	95.63	98.11	91.53
cayenne	97.55	98.18	96.9	97.54	98.97	95.11	openjpa	96.03	96.19	95.85	96.02	98.39	92.06
cocoon	97.43	97.43	97.43	97.43	99.22	94.86	openmeetings	91.3	86.26	98.26	91.87	97.98	83.42
commons-math3	98.12	97.31	98.97	98.13	99.54	96.25	openwebbeans	97.3	98.17	96.4	97.28	97.53	94.61
continuum	91.49	89.8	93.62	91.67	98.52	83.05	pulsar	96.06	98.17	93.88	95.98	98.13	92.22
cxfr	96.2	96.63	95.74	96.18	98.55	92.41	qpidd-jms	94.37	94.37	94.37	94.37	98.83	88.73
directory-server	97.94	97.76	98.13	97.94	99.49	95.88	ranger	91.62	93.37	89.6	91.45	98.26	83.31
directory-studio	93.77	96.24	91.1	93.6	95.5	87.67	roller-weblogger	87.36	83.33	93.41	88.08	95.79	75.28
drill	97.74	99.02	96.43	97.71	99.4	95.51	samza	95.66	98.77	92.49	95.53	98.72	91.51
fop	92.63	89.9	96.05	92.87	97.9	85.46	shiro	94.23	93.67	94.87	94.27	98.49	88.47
giraph	90.36	100	80.72	89.33	89.62	82.26	storm	98.6	99.52	97.66	98.58	99.37	97.21
hadoop	97.68	98.28	97.06	97.67	99.26	95.37	struts	92.58	92.25	92.97	92.61	97.29	85.16
hbase	98.24	98.24	98.24	98.24	99.16	96.49	syncope	98.42	98.59	98.24	98.41	99.7	96.83
helix	94.58	98.2	90.83	94.37	99.06	89.42	tajo	98.14	99.05	97.21	98.12	99.35	96.3
hive	97.71	97.6	97.83	97.71	99.57	95.42	tez	95.43	99.45	91.37	95.24	99.14	91.16
isis	97.58	96.13	99.15	97.62	98.85	95.2	tika	97.4	98.03	96.75	97.39	98.78	94.81
jackrabbit	95.67	95.81	95.52	95.66	98.55	91.34	tomcat	93.44	93.44	93.44	93.44	97.74	86.89
jackrabbit-oak	96.74	98.31	95.11	96.68	98.44	93.53	uima-ruta	98.05	98.52	97.56	98.04	99.5	96.1
jena	98.99	99.79	98.18	98.98	99.52	97.99	wicket	98.05	99.2	96.88	98.03	99.38	96.13

Table 30 God Class code smell detection performance

God Class													
Dataset	ACC	P	R	F1	AUC	MCC	Dataset	ACC	P	R	F1	AUC	MCC
activemq	96.97	99.52	94.4	96.89	98.95	94.07	johnzon	94.79	97.78	91.67	94.63	98.18	89.76
archiva	89.32	90.35	88.03	89.17	94.5	78.66	kafka	96.45	98.97	93.87	96.35	98.7	93.02
atlas-sources	92.37	98.26	86.26	91.87	98.29	85.37	karaf	95.63	98.45	92.72	95.5	97.96	91.42
avro	98.55	100	97.1	98.53	99.14	97.14	kerby	93.03	95.65	90.16	92.82	98.55	86.21
beam	97.61	99.15	96.04	97.57	98.63	95.26	knox	96	97.63	94.29	95.93	99.25	92.05
bookkeeper	96.05	97.3	94.74	96	97.24	92.14	kylin	97.6	99.55	95.63	97.55	98.59	95.27
calcite	99.04	100	98.07	99.03	99.61	98.09	lucene	97.41	98.25	96.55	97.39	98.67	94.84
camel	94.79	94.98	94.58	94.78	99.11	89.58	maven	95.95	99.04	92.79	95.81	98.74	92.08
carbodata	92.55	95.97	88.82	92.26	97.42	85.33	nifi	93.83	96.65	90.81	93.64	97.69	87.82
cayenne	92.57	90.26	95.44	92.78	98.18	85.28	nutch	92.44	95.06	89.53	92.21	97.79	85.03
cocoon	96.43	98.51	94.29	96.35	98.85	92.94	ofbiz	95.13	96.3	93.86	95.06	98.7	90.28
commons-beanutils	95.12	100	90.24	94.87	96.1	90.68	olingo-odata	88.2	86.17	91.01	88.52	96.35	76.53
commons-collections4	95.76	100	91.53	95.58	97.33	91.86	openjpa	98.28	98.94	97.61	98.27	99.32	96.58
commons-compress	92.55	97.62	87.23	92.13	97.69	85.59	openmeetings	93.75	97.12	90.18	93.52	97.61	87.72
commons-dbc2	89.58	88	91.67	89.8	98.44	79.24	opennlp	97.81	97.39	98.25	97.82	99.85	95.62
commons-lang3	94.12	95.92	92.16	94	98.37	88.3	openwebbeans	94.71	96.97	92.31	94.58	98.59	89.53
commons-math3	94.12	95.45	92.65	94.03	97.41	88.27	parquet	96.12	98.97	93.2	96	97.94	92.39
commons-net	91.25	92.31	90	91.14	95.78	82.53	phoenix	90.79	94.29	86.84	90.41	95.29	81.83
commons-validator	90.48	86.96	95.24	90.91	93.76	81.32	pulsar	98.44	98.3	98.58	98.44	99.8	96.88
commons-vfs	93.85	98.31	89.23	93.55	98.17	88.07	qpids-jms	91.91	93.85	89.71	91.73	96.15	83.91
continuum	95.45	96.88	93.94	95.39	98.13	90.95	ranger	94.97	97.08	92.74	94.86	98.84	90.03
crunch	94.44	97.06	91.67	94.29	97.28	89.03	reef	94.91	97.47	92.22	94.77	98.88	89.95
curator	87.5	95.35	78.85	86.32	92.1	76.15	roller-weblogger	94.27	96.7	91.67	94.12	98.24	88.66
cx4	95.5	97.61	93.29	95.4	98.53	91.1	samza	95.34	95.63	95.03	95.33	98.74	90.68

directory-server	94.53	95.45	93.52	94.48	98.24	89.09	shiro	93.28	95.31	91.04	93.13	97.57	86.65
directory-studio	93.33	95.85	90.59	93.15	97.56	86.8	struts	90.77	90.77	90.77	90.77	97.85	81.54
flink	95.83	97.83	93.75	95.75	98.57	91.75	syncope	92.88	94.34	91.24	92.76	98.31	85.81
fop	96.8	98.67	94.88	96.74	99.25	93.68	tajo	95.92	97.31	94.44	95.85	98.37	91.87
giraph	94.81	100	89.62	94.53	98.73	90.11	tez	97.99	99.48	96.48	97.96	99.41	96.02
hadoop	94.92	99.39	90.39	94.68	98.7	90.2	tika	92.36	93.57	90.97	92.25	97.26	84.75
helix	96.15	100	92.31	96	98.38	92.58	tinkerpop	95.98	98.13	93.75	95.89	99.23	92.06
isis	94.79	95.81	93.69	94.74	99.16	89.61	tomcat	96.42	97.34	95.44	96.38	99.54	92.85
jackrabbit	94.84	96.74	92.81	94.73	98.34	89.76	uima-ruta	94.56	97.25	91.71	94.4	97.86	89.26
jackrabbit-oak	94.29	95.32	93.14	94.22	98.11	88.59	wicket	95.03	96.31	93.65	94.96	98.21	90.09
jena	97.65	98.41	96.87	97.63	99.19	95.32	-	-	-	-	-	-	-

Table 31 Feature Envy code smell detection performance

Dataset	Feature Envy												
	ACC	P	R	F1	AUC	MCC	Dataset	ACC	P	R	F1	AUC	MCC
accumulo	65.24	61.32	82.55	70.37	72.87	32.48	jena	94.17	97.93	90.24	93.93	98.66	88.61
activemq	98.25	98.82	97.67	98.24	99.6	96.51	kafka	91.8	96.36	86.89	91.38	94.28	84.01
airavata	66.15	60.48	93.17	73.35	71.54	38.39	manifoldcf	92.45	100	84.91	91.84	99.81	85.89
beam	94.34	97.2	91.32	94.17	98.28	88.85	nifi	97.77	97.58	97.98	97.78	99.6	95.55
calcite	97.14	98.53	95.71	97.1	99.44	94.32	ofbiz	93.5	93.07	94	93.53	97.71	87
cocoon	61.85	93.1	25.59	40.15	78.39	34.41	olingo-odata	96.71	93.83	100	96.82	98.65	93.62
commons-math3	96.25	94.05	98.75	96.34	98.23	92.62	openjpa	90.05	99.66	80.38	88.99	96	81.65
cxfl	98.56	98.21	98.92	98.56	99.53	97.12	ranger	94.81	94.81	94.81	94.81	98.15	89.61
directory-studio	99.32	100	98.63	99.31	99.97	98.64	storm	72.36	65.05	96.63	77.76	73.35	51.15
drill	97.49	96.59	98.46	97.52	98.92	95	syncope	94.96	98.33	91.47	94.78	99.35	90.14
fop	97.56	97.56	97.56	97.56	99.77	95.12	tajo	95.87	98.26	93.39	95.76	99.13	91.85

giraph	87.98	97.12	78.29	86.69	93.89	77.44	tez	85.23	87.8	81.82	84.7	92.99	70.62
hadoop	92.42	98.28	86.36	91.94	97.27	85.48	tinkerpop	94.03	90.41	98.51	94.29	99.03	88.41
hbase	79.97	71.82	98.66	83.13	82.15	64.63	tomcat	96.31	96	96.64	96.32	97.21	92.62
hive	82.64	76.78	93.6	84.36	87.35	66.91	uima-ruta	97.75	97.75	97.75	97.75	99.66	95.51
jasml	57.14	55.56	71.43	62.5	67.35	14.91	zeppelin	68.64	61.46	100	76.13	68.64	47.87

Table 32 Long Method code smell detection performance

Long Method													
Dataset	ACC	P	R	F1	AUC	MCC	Dataset	ACC	P	R	F1	AUC	MCC
accumulo	59.27	55.18	98.78	70.81	70.93	30.26	jasml	37.5	33.33	25	28.57	25	-25.82
activemq	99.88	99.75	100	99.87	100	99.75	jena	98.38	99.34	97.4	98.36	99.84	96.77
airavata	65.24	59.96	91.78	72.53	70.22	35.96	manifolcd f	97.68	100	95.36	97.62	98.68	95.47
calcite	98.92	99.13	98.71	98.92	99.73	97.85	nifi	99.27	99.27	99.27	99.27	99.85	98.53
cocoon	97.58	96.83	98.39	97.6	99.85	95.17	ofbiz	96.74	98.86	94.57	96.67	98.64	93.57
cx	96.34	98.9	93.73	96.25	98.41	92.81	olingo- odata	96.74	96.74	96.74	96.74	98.84	93.48
drill	98.6	98.8	98.4	98.6	98.7	97.2	pulsar	99.18	99.18	99.18	99.18	99.96	98.36
fop	95.28	99.39	91.11	95.07	98.72	90.87	storm	77.54	70.2	95.72	81	78.83	59.13
hadoop	98.34	98.88	97.79	98.33	99.07	96.69	tajo	95.31	96.77	93.75	95.24	97.96	90.67
hbase	86.05	79.12	97.93	87.53	87.28	74.22	tomcat	91.4	97.1	85.35	90.85	98.74	83.42
hive	82.47	76.51	93.71	84.24	87.72	66.65	zeppelin	67.27	63.01	83.64	71.87	77.54	36.56

Compared to previous studies, as summarized in Table 33, Barbez et al. [144] proposed a DL-based approach for detecting God Class using CNN, which achieved 68% f-score for detecting God Class in Tomcat, 38% in Qpid, 23% in Struts, and 21% in Wicket dataset, whereas our model achieved 96%, 98%, 90%, and 94% f-scores in Tomcat, Qpid, Struts, and Wicket datasets, respectively. In another study by Hadj-Kacem and Bouassida [147], where they utilized variational autoencoder to extract features from ASTs, their model

achieved high f-scores for detecting Blob in Lucene (80%) and Qpid (80%) datasets, yet our approach has higher scores with 97% f-score in Lucene and 98% in Qpid datasets. Compared to the previous approach, the Hadj-Kacem and Bouassida [147] approach achieved 84% and 85% f-scores for detecting Feature Envy and Long Method, respectively, in the Hive dataset, whereas our approach achieved similar results (i.e., 84% f-score for detecting both smells).

Table 33 Comparison with previous studies

Dataset	Tomcat	Qpid	Struts	Wicket	Lucene	Hive
God Class						
Barbez et al.	68%	38%	23%	21%	-	-
Hadj-Kacem and Bouassida	-	80%	-	-	80%	-
CoRT	96%	98%	90%	94%	97%	-
Feature Envy						
Hadj-Kacem and Bouassida	-	-	-	-	-	84%
CoRT	-	-	-	-	-	84%
Long Method						
Hadj-Kacem and Bouassida	-	-	-	-	-	85%
CoRT	-	-	-	-	-	84%

In our study, we benchmarked CoRT against a comprehensive set of state-of-the-art pretrained models for code smell detection. We considered a diverse range of encoder-only (i.e., CodeBERT [153] and GraphodeBERT [154]), decoder-only (i.e., CodeGPT [158]), and encoder-decoder (i.e., CodeT5 [156] and Unixcoder [171]) models to ensure a thorough evaluation.

Table 34 presents the results of a comprehensive evaluation of the pretrained models for code smell detection. For each code smell, we averaged the results across all datasets. We observed variations in the performance across different code smells and models. When it comes to detecting Data Class and God Class code smells, CoRT emerged as the best

performer, achieving 95.93% accuracy and an F1-score of 95.89% and 94.59% accuracy and an F1-score of 94.48%, respectively. Moreover, it exhibits remarkable precision and recall, making it a promising choice for identifying class-level code smells. However, for other models, such as CodeBERT, CodeT5, GraphCodeBERT, and Unixcoder, although their performance was commendable, they showed varying degrees of trade-off between precision and recall.

The detection of Feature Envy and Long Method code smells presents additional challenges for the evaluated models. CoRT continued to excel in Feature Envy detection, achieving an accuracy of 88.40% and an F1-score of 88.68%. The results for the long method detection follow a similar pattern to the Feature Envy detection, with CoRT outperforming the other models, achieving an accuracy of 87.95% and an F1-score of 88.87%. Nevertheless, other models have faced difficulties in achieving a balanced trade-off between precision and recall in these categories, with CodeGPT struggling to achieve satisfactory performance across all code smells.

Table 34 Comparison with state-of-the-art pre-trained models

Data class							God class					
Model	ACC	P	R	F1	AUC	MCC	ACC	P	R	F1	AUC	MCC
CodeBERT	63.51	44.86	59.13	49.30	63.51	27.17	69.87	58.59	67.24	60.40	69.87	40.27
GraphCodeBERT	65.92	49.38	62.17	53.21	65.92	32.01	70.50	59.40	68.95	61.76	70.50	41.43
CodeGPT	50.88	23.04	40.75	28.17	50.88	2.03	50.83	26.91	46.18	31.66	50.83	2.16
CodeT5	53.99	39.06	69.76	49.31	53.99	8.33	54.74	33.04	50.13	37.31	54.74	10.10
Unixcoder	64.77	52.34	64.02	54.21	64.77	30.87	64.49	54.62	61.80	52.69	64.49	30.39
CoRT	95.93	96.82	95.12	95.89	98.33	91.99	94.59	96.54	92.58	94.48	98.05	89.33
Feature envy							Long method					
Model	ACC	P	R	F1	AUC	MCC	ACC	P	R	F1	AUC	MCC

CodeBERT	59.78	46.07	69.89	53.71	59.78	20.48	60.74	47.57	71.60	55.39	60.74	21.87
GraphCodeBERT	65.20	53.60	64.21	54.90	65.20	32.33	62.23	46.15	67.26	53.21	62.23	24.63
RT												
CodeGPT	50.00	21.88	43.75	29.17	50.00	0.00	50.00	29.55	59.09	39.40	50.00	0.00
CodeT5	57.43	38.85	60.26	45.73	57.43	15.59	56.27	39.26	59.49	45.04	56.27	13.84
Unixcoder	69.00	54.43	53.12	50.48	69.00	39.67	65.28	45.59	56.47	48.90	65.28	30.95
CoRT	88.40	89.49	90.38	88.68	92.40	78.34	87.95	87.10	92.11	88.87	90.21	77.16

RQ3 Answer. The detection performance of CoRT was evaluated for four downstream tasks, focusing on class-level (Data Class and God Class) and method-level (Feature Envoy and Long Method) code smells. CoRT exhibited strong detection performance for Data Class and God Class, achieving F1 scores between 88.08-99.4 for Data Class and, 86.32-99.03 for God Class. However, the performance varied across different datasets, with some datasets showing excellent results and others displaying lower performances. Project size appeared to influence CoRT performance, with larger projects demonstrating better results. For method-level smells, CoRT achieved F1 scores between 40.15-99.31 for Feature Envoy and between 28.57-99.87 for the Long Method. Similar to class-level smells, the CoRT performance varies across different datasets, with larger projects generally showing better results. Notably, the CoRT detection performance did not appear to be solely dependent on the number of smelly instances present in a dataset.

4.5.4 RQ4 How efficient is the proposed approach?

The time complexities of pre-training and fine-tuning are presented in Table 35 and Table 36, respectively. For each pretrained model, we calculated the average, minimum, and maximum training times across all models trained during hyperparameter optimization. For fine-tuning, we calculated the average, minimum, and maximum tuning times across all labeled datasets for each code smell. The tables clearly show that pre-training is time-

consuming compared with the fine-tuning process, which is one of the main advantages of self-supervised learning, where a single complex model is trained and then efficiently used on multiple downstream tasks. Among the pre-trained models, CoRT and ANN had the shortest average training time of 6 h per training, whereas LSTM had the longest training time, with an average of 16 h. The CNN comes after the LSTM with an average of 10 h. Although CoRT and ANN had the shortest training times, CoRT had the best performance in terms of training accuracy and loss, whereas ANN had the worst performance (RQ1). For fine-tuning, the average tuning time across the code smells was between 1.3-2.02 minutes.

RQ4 Answer. Pre-training is more time-consuming than the fine-tuning process. Among the pretrained models, CoRT and ANN had the shortest average training times (6 h), whereas LSTM had the longest (16 h). Despite the shorter training time, CoRT demonstrated the best performance in terms of accuracy and loss (RQ1), whereas the ANN exhibited the worst performance. For fine-tuning, the average tuning time across code smells ranges from 1.3 to 2.02 minutes.

Table 35 Pre-training time

Model	Avg (min.)	Min (min.)	Max (min.)
ANN	350.33	214	550
CNN	583.95	231	1714
LSTM	954.65	349	2980
CoRT	355.71	251	664

Table 36 Fine-tuning time

Code smell	Avg (min.)	Min (min.)	Max (min.)
Data Class	2.02	0.55	5.73

God Class	1.45	0.23	5.52
Feature Envy	1.30	0.22	2.87
Long Method	1.45	0.23	2.77

4.5.5 RQ5 To what extent does CoRT enhance the detection performance compared to the feature-based approach?

In this study, we compared the proposed approach with supervised learning and feature-based approaches. The results are presented in Figure 25, in which the CoRT ACC, P, R, F1, AUC, and MCC boxplots are plotted against the baseline and feature-based models for each code smell. The feature-based model performed competitively with CoRT (fine-tuned) in detecting class-level smells, where both models had smaller boxes (i.e., lower dispersion) with shorter whiskers. However, CoRT outperformed the feature-based model in detecting method-level smells because it had smaller boxes with shorter whiskers. Moreover, we observed that the baseline model struggled to obtain a high detection performance for all four code smells. The feature-based model demonstrated competitive performance with CoRT because both models were pretrained using unlabeled data. However, CoRT consistently outperformed the feature-based model in detecting method-level smells. This suggests that the fine-tuning process with CoRT enhances detection performance, particularly for code smell detection at the method level.

RQ5 Answer. A comparison between the proposed CoRT approach and the supervised learning baseline and feature-based approaches revealed that the feature-based model performed competitively with CoRT in detecting class-level smells but was outperformed by CoRT in detecting method-level smells. The baseline model struggled to achieve high

detection performance for all code smells. These findings highlight the enhanced detection performance of CoRT compared with feature-based and supervised learning approaches, particularly for method-level smells.

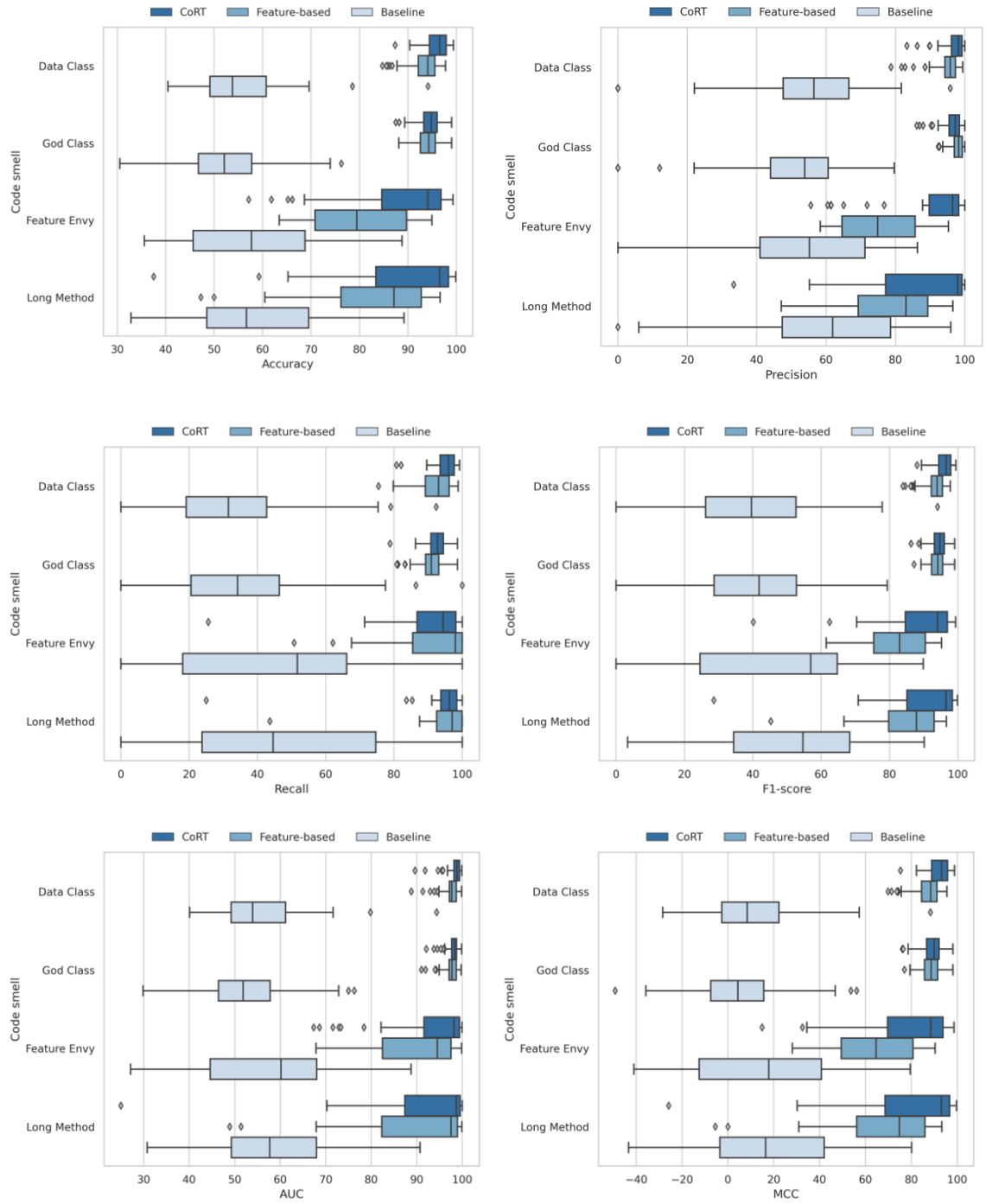


Figure 25 Detection performance boxplots of CoRT compared to baseline and feature-based

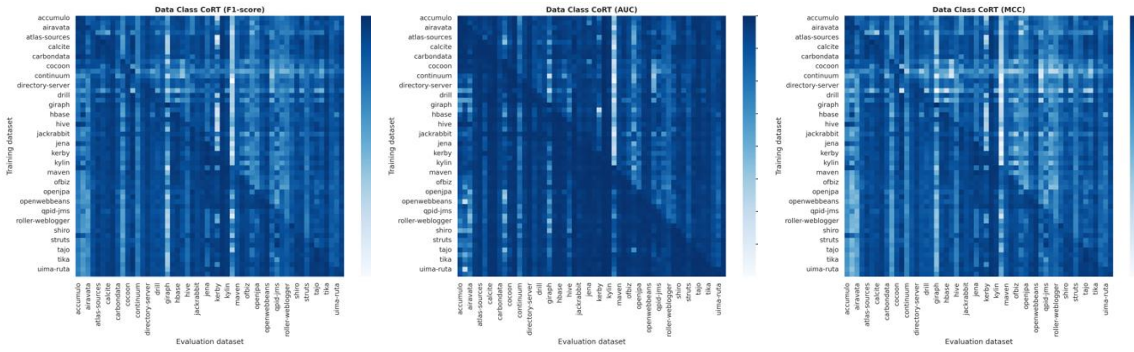
4.5.6 RQ6 How well does CoRT generalize on unseen projects?

In this section, we perform a cross-project evaluation. The main goal of this experiment is to evaluate the extent to which a fine-tuned model (on one dataset) can be used to detect smells in other (unseen) datasets. We performed a cross-project evaluation of CoRT and the feature-based and baseline models. In this section, we consider only the F1, AUC, and MCC metrics. The results of the other metrics are provided in Appendix E. Figure 26, Figure 27, Figure 28, and Figure 29 illustrate the detection performance of the cross-project evaluation for the Data Class, God Class. Feature envy and long methods. We used heatmap plots, which can be interpreted as follows: 1) Vertical Axis (Fine-tuning Datasets): the datasets on the vertical axis represent the datasets used for fine-tuning CoRT, feature-based model, and baseline model. Each row corresponds to the specific dataset used to train the model. 2) Horizontal Axis (Evaluation Datasets): The datasets on the horizontal axis represent those used for evaluation. Each column corresponds to the specific dataset on which the trained models were evaluated. 3) Color Representation: The Color shading in the heatmap represents the detection performance. Darker colors indicate higher performance, whereas white colors indicate lower performance. This color gradient allows for a visual comparison of the detection performance across different combinations of fine-tuning and detection datasets.

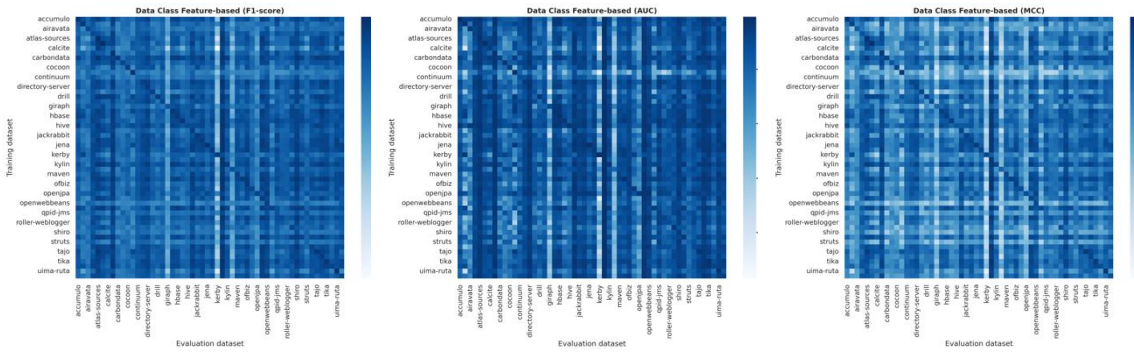
In most cases, we observed that the diagonal axis was darker than the rest because the same datasets used for fine-tuning were evaluated. Although the diagonal axis represents the evaluation of the same dataset used for fine-tuning, we do not necessarily expect perfect performance along this line. The reason behind This is because detection performance can

be influenced by various factors, including differences in the codebase, project characteristics, and specific code smells present in each dataset. For class-level smells, CoRT has a better detection performance when used on unseen datasets. Some datasets, such as Kerby, were difficult to detect using the feature-based model; however, Kerby performed better than CoRT. For method-level smells, we observed that both feature-based and CoRT had similar detection performances, where both models struggled to detect almost the same datasets. These insights contribute to the understanding of the generalization ability of CoRT compared to feature-based and baseline models.

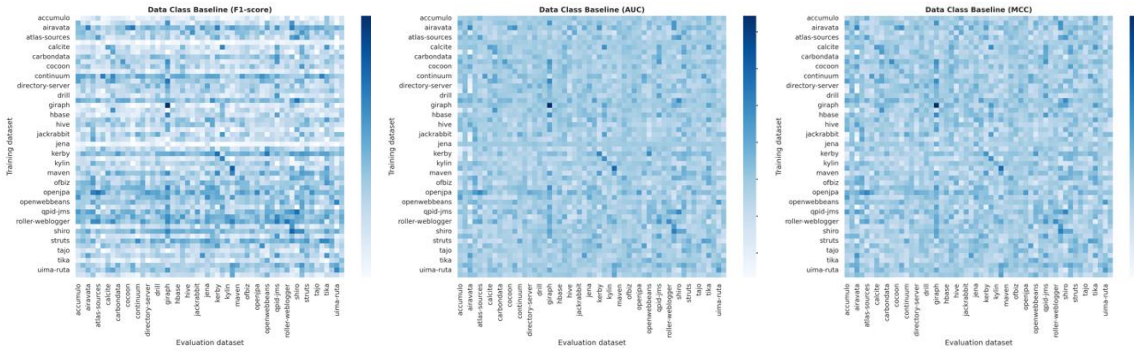
RQ6 Answer. This cross-project evaluation aimed to assess the generalization capabilities of the CoRT, feature-based, and baseline models in detecting code smells on unseen datasets. Heatmap visualizations showed that CoRT generally outperformed the feature-based model in detecting class-level smells on unseen datasets, whereas both models struggled to detect certain method-level smells. The baseline model struggled to achieve high detection performance for all code smells. These results highlighted the advantage of CoRT over feature-based and baseline models in terms of its generalization capability for detecting code smells across different projects.



a) CoRT

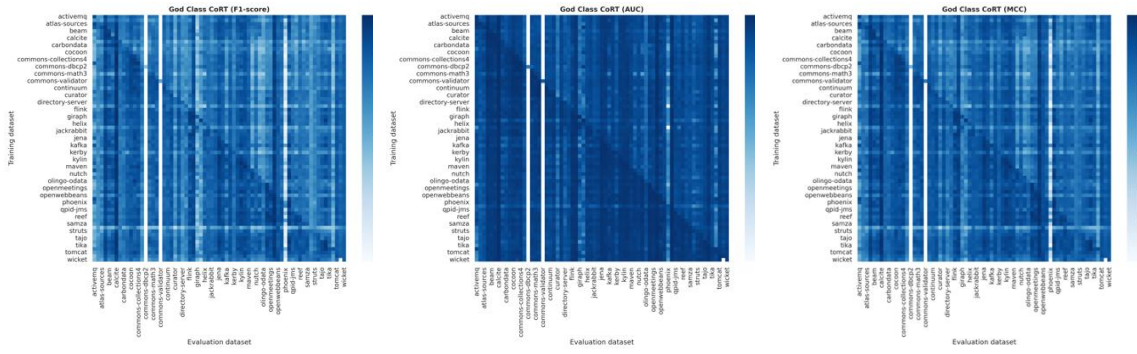


b) Feature-based

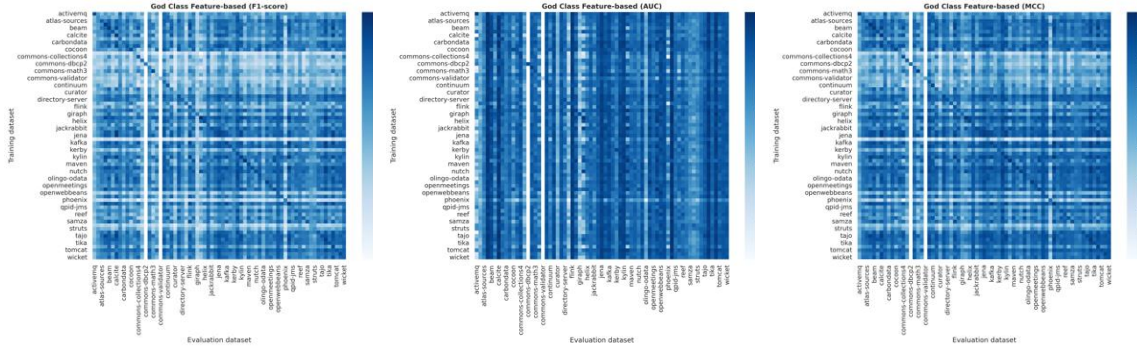


c) Baseline

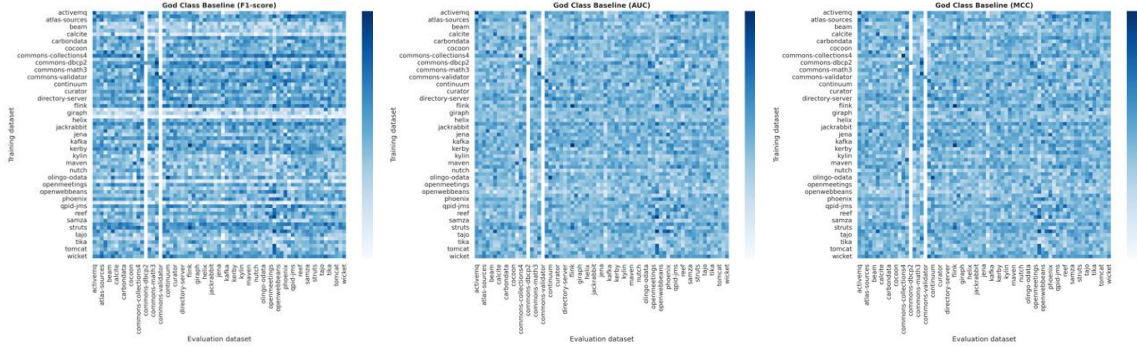
Figure 26 Cross-project performance heatmap for Data Class



a) CoRT

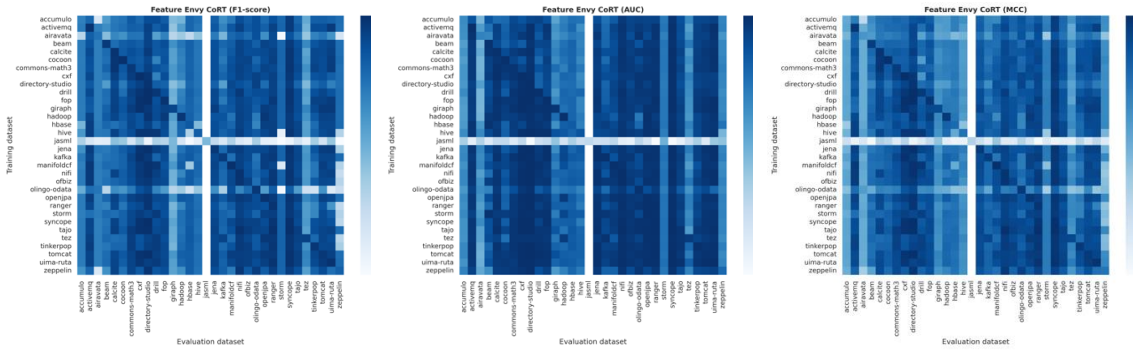


b) Feature-based

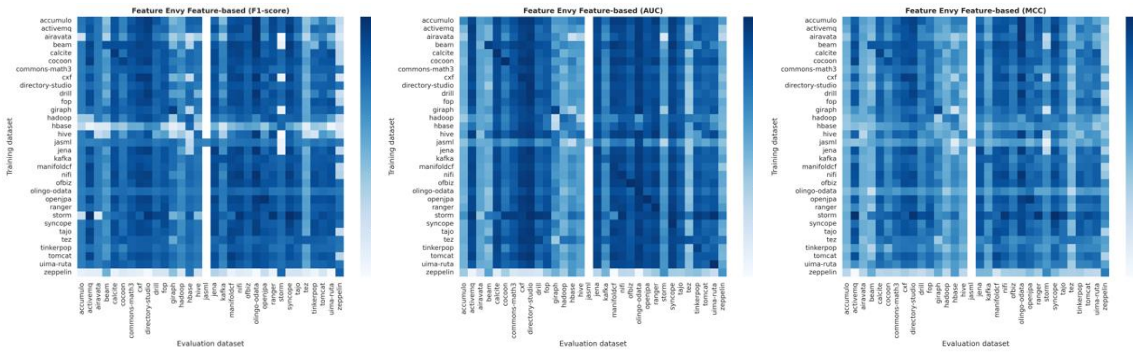


c) Baseline

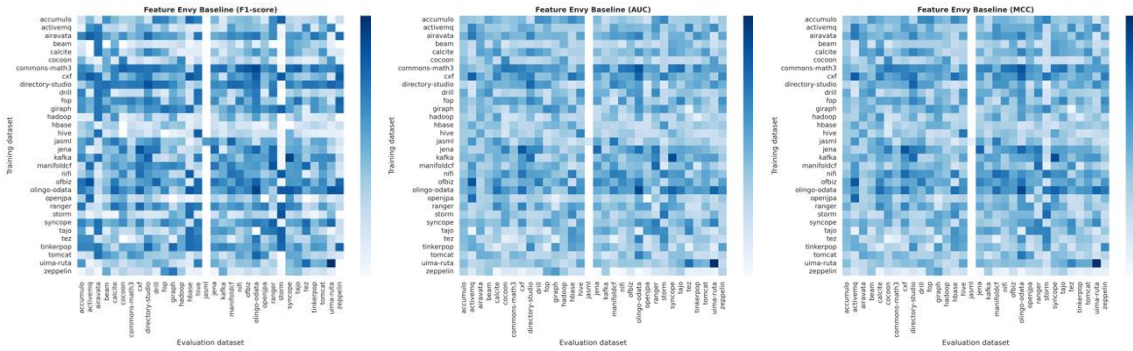
Figure 27 Cross-project performance heatmap for God Class



a) CoRT

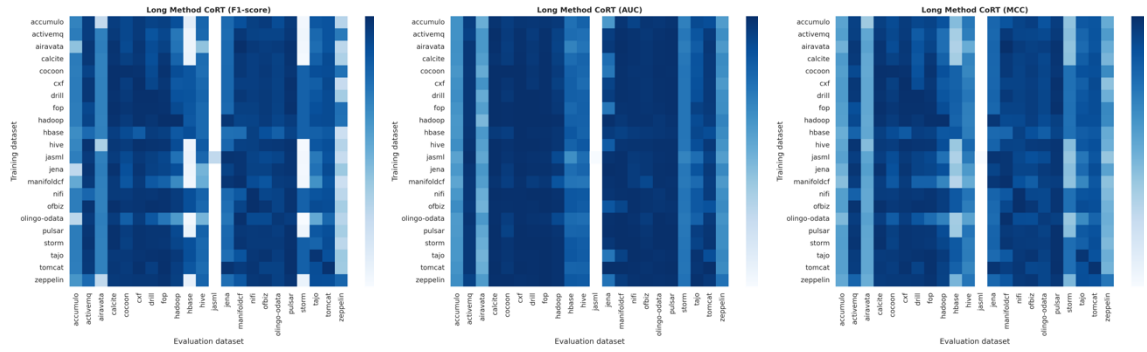


b) Feature-based

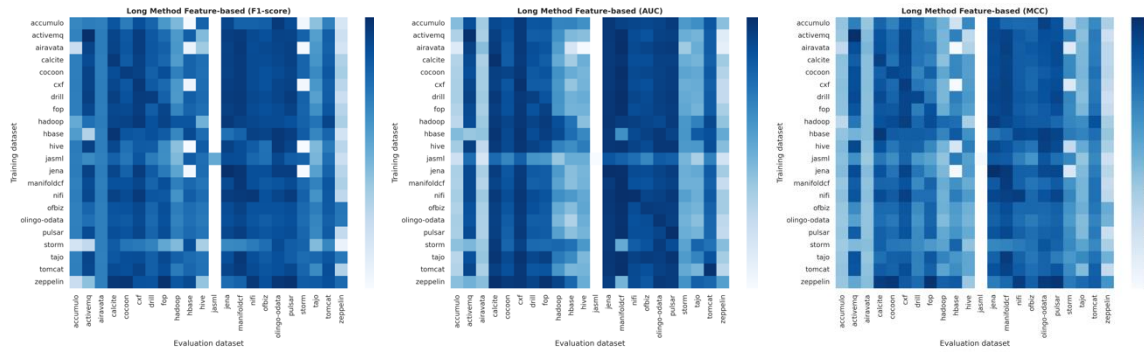


c) Baseline

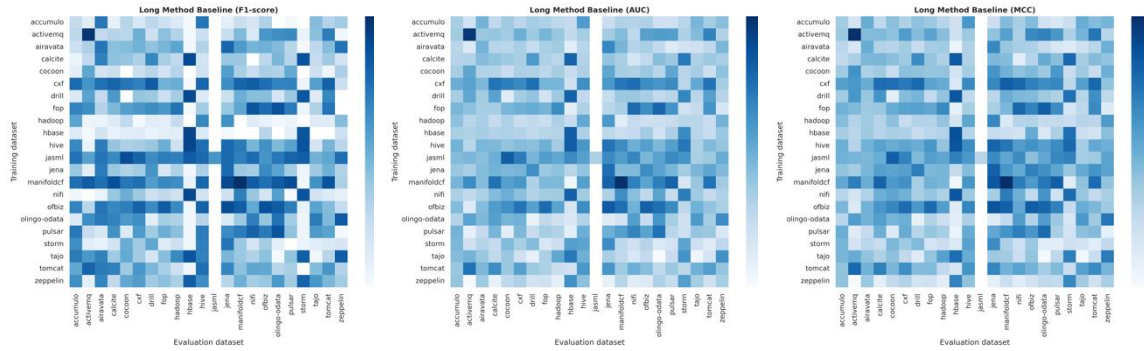
Figure 28 Cross-project performance heatmap for Feature Envy



a) CoRT



b) Feature-based



c) Baseline

Figure 29 Cross-project performance heatmap for Long Method

4.6 Implications and Limitations

CoRT is a DL-based self-supervised model trained with a proxy task using a transformer. CoRT was trained using 16 million lines of code collected from 103 open-source projects and was used to detect code smells in the source code. In this section, the implications and limitations of the proposed approach are presented.

4.6.1 Proxy Tasks

The selection of a proxy task plays a vital role in providing a good representation of structural and semantic features. For instance, if we choose to use source code without punctuation, the original structure of the statements will be lost, and this proxy task will do more harm than good. The findings of this study suggest that the defined proxy task was well formulated, as it produced good results in both the pre-training and fine-tuning phases. Although CoRT performs well on the proxy task, it requires hyperparameter optimization to determine the optimal settings, which is an expensive process. Thus, it is necessary to develop a model with good representation without an extensive hyperparameter search. Strategies to enhance CoRT may involve formulating a new proxy task, such as masking punctuation instead of the reserved word, or a combination of the two methods. Although the premise of self-supervised learning is not to use labeled data, pseudo-labels must be created from the proxy task. One disadvantage of this approach is that it requires enormous volumes of data to generate accurate pseudo-labels without compromising the accuracy.

4.6.2 Detection Performance

Our results suggest that the proposed method has better detection performance in detecting class-level code smells than method-level code smells. There are several possible

explanations for these results. First, the pretrained model was trained using all the extracted classes, assuming that the methods were included within the classes. Second, previous studies that used an ML-based approach reached a similar conclusion, where ML models struggled to detect method-level smells compared to class-level smells [172]. A possible area for future research would be to investigate why method-level code smells are difficult to detect. A reasonable approach to tackle this issue is to train the pre-trained model using methods instead of classes, and then use the model to detect method-level code smells.

To further enhance detection performance, the proposed method can be extended to enhance its generalizability by including other features as additional inputs alongside textual features, such as structural information. Incorporating structural information such as class hierarchies and dependencies can provide a more comprehensive understanding of code smells. By analyzing both the textual and structural aspects, we can detect a wider range of code smells and improve the accuracy of our approach. Although integrating structural information poses challenges such as extracting and integrating relevant features, addressing them would enhance the effectiveness of our method in detecting code smells. Moreover, this study can be extended to include a variety of code-related tasks, such as code understanding and code generation, by incorporating a natural language and/or other programming languages as additional inputs to the model. This allows the model to be applied to tasks, such as natural language code searches, code documentation, code translation, and code refinement.

4.6.3 Computational Efficiency

The proposed method has two main stages: pre-training and fine-tuning. There are significant computational advantages to computing an expensive representation of the data

in advance (i.e., the pre-training step) and then applying the model to multiple tasks using cheaper models in addition to the representations (i.e., fine-tuning). However, compared to supervised learning, the pre-training phase is computationally expensive because it requires a large amount of data and consists of two steps: 1) defining pseudo-labels and 2) training the model on pseudo-labels. Therefore, the trade-offs between the two approaches must be considered in future studies.

4.6.4 Comparative evaluation

The findings from the evaluation show that CoRT is effective for both fine-tuning and feature-based approaches. However, we recommend fine-tuning the model because it requires minutes of training and will obtain better results. In our study, we constrained the comparison of the proposed approach to the baseline and feature-based approaches. We did not compare the performance with other DL-based approaches because our study's scope and research goal only targeted the implementation of a self-supervised learning approach for code smell detection. Empirical studies are required to demonstrate the superiority of the proposed approach over other DL-based approaches.

4.6.5 Advantages for researchers and developers

The proposed method has several benefits for researchers and developers because semantics and structural features are automatically extracted without the need for transformations and/or defining rules and thresholds for the detection of each code smell. This will facilitate the adaptation of the method by developers, which implies the development of plugins and/or software bots for detecting code smells, and consequently, using them in reality. Moreover, the proposed approach is not suitable for a specific programming language or task, which might open doors for researchers to investigate the

capability of this approach in detecting code smells in other programming languages or applying the model to other downstream tasks such as code generation and defect predictions.

4.7 Threats to Validity

We made some assumptions that could compromise the accuracy of our findings. In this section, we present potential threats and how to mitigate them.

4.7.1 Conclusion validity

This threat is related to the association between the treatment and outcomes. The evaluation methodology used to evaluate the proposed approach is considered a threat, as the use of other evaluation methods might affect the outcomes of this study; however, we adopted 80/20 and 70/30 validation methods, which are the two most commonly used methods for evaluating DL-based models. Regarding evaluation metrics, adopting one metric exposes the risk of bias, leading to misleading outcomes. Therefore, we used multiple metrics that are commonly used in the literature, such as precision, recall, accuracy, F-score, MCC, and AUC. In addition, some metrics, such as precision and recall, are threshold-dependent, and finding the proper threshold is challenging. Therefore, we adopted both threshold-dependent and threshold-independent evaluation metrics, such as AUC and loss.

4.7.2 Construct validity

Regarding the potential issues related to the relationship between theory and observation, we identified potential problems with the adopted methodology. The implementation of the experimental prediction models exposes a threat; we utilized the implementation

provided by TensorFlow, which is considered a reliable framework for implementing DL models. In this study, we used tokenized source code as an independent variable. The tool used for data extraction might be a threat because we assumed its reliability; however, we selected ANTLR4, which has been used in previous studies [173] [164] [174]. Another threat to the validity stems from the labeled datasets used to construct the fine-tuned dataset. To mitigate this threat, we selected labeled datasets that were verified by the original authors. Moreover, we automated the process of mapping the extracted code to labels to minimize the risk of mislabeling because manual mapping is error-prone.

Overfitting is a threat, particularly when DL techniques are employed. Overfitting can occur when a model is overly tailored to specific characteristics of the training dataset, thereby compromising its ability to generalize unseen data. To mitigate this risk, we employed a regularization technique, which is a dropout in model training, to control the complexity of the model and prevent it from fitting noise or irrelevant features in the dataset. We opted for dropout because other techniques, such as early stopping, have their limitations. A study [175] suggests that continuing the training process, even if no significant change in validation loss is observed, can lead to better generalization. This contrasts with the common practice of early stopping, where training is typically halted once the validation loss ceases to decrease, to prevent overfitting. Additionally, we considered potential bias in the dataset as another threat to construct validity. Dataset bias can arise due to various factors, including the data collection process, sampling methods, and characteristics of the selected projects. Several measures were taken to minimize bias. First, we employed a representative sampling strategy that included projects from diverse industries, maturity levels, and sizes. This helped mitigate the risk of bias towards a specific

project type or context. Second, we thoroughly documented the dataset collection process, including the sources from which the data were collected.

In our study, we utilized pseudo-labels generated through a specific process to pre-train the model. While this approach has demonstrated its utility in detecting code smells, it is important to acknowledge the potential limitations associated with pseudo-labels, which might threaten construct validity. The effectiveness of pseudo-labels may be influenced by the choice of pre-training task and pre-trained model. Variations in the pre-training objectives or model architectures may yield different results when pseudo-labels are incorporated into the training pipeline. Hence, the choice of pre-training tasks should be carefully considered and examined. We mitigated this threat by evaluating the impact of different model architectures, including ANN, CNN, and LSTM, on the performance of pseudo-labels. Moreover, we evaluated the proposed pre-training task against five pre-trained models trained using various pre-training tasks. Nevertheless, although our study demonstrates the potential benefits of the pre-training task, further validation and consideration are essential to support its robustness and suitability for detecting other types of code smells.

4.7.3 Internal validity

Internal threats were associated with the accuracy of the experimental outcomes. Code smell definitions are subjective, and code smells with almost identical definitions may have two names. To mitigate this problem, we referred to the definition used by the authors of the original papers to label the datasets; in cases where no definition was stated, we referred to the original definitions proposed by Fowler [2].

4.7.4 External validity

This threat is related to the generalization of the conclusions of the study. With respect to generalizability, we evaluated our approach on four code smells. Although we selected four of the most common code smells, two at the class level and two at the method level, the results cannot be generalized to other types. Although the proposed approach was effective in detecting four specific code smells, it is important to acknowledge that the applicability of this approach to the detection of other types of code smells may vary. This variation stems from the distinctive characteristics inherent to each code smell, which may require different sets of metrics for accurate identification and analysis. In terms of programming languages, we believe that our method is language-agnostic and can be adapted for any programming language; however, we evaluated our approach using Java projects; hence, we cannot claim that this method works well for programming languages other than Java. Finally, the results cannot be generalized to all software systems. To mitigate this, we have tried to use a sizable and representative dataset of 103 Java projects. Moreover, we performed several cross-project experiments to assess the generalizability of the proposed method.

4.8 Chapter Summary

In this chapter, we introduced a novel proxy task for self-supervised learning that trains a transformer model to recognize code representations. We showed that the proposed method successfully forces the trained model (CoRT) to learn structural and semantic features that are useful for multiple downstream tasks, such as the detection of Data Class, God Class, Feature Envy, and Long Method code smells. We built our model by comparing multiple

DL model architectures and applying hyperparameter optimization. We exhaustively evaluated the proposed method by applying CoRT to two class-level, and two method-level code smells. In addition, we evaluate the time complexity of the proposed approach. Moreover, we thoroughly assessed our method by comparing it with supervised and feature-based approaches and achieved high detection performance in all code smells. Specifically, we evaluated the generalizability of our model compared to supervised and feature-based methods by applying a cross-project evaluation. We obtained high detection scores in terms of accuracy, precision, recall, F1-score, AUC, and MCC, significantly surpassing other approaches.

Overall, the findings of **RQ1** highlight the influence of different model architectures on the performance of the pretrained model, and CoRT demonstrated superior results in this study. **RQ2** results highlighted the significant influence of hyperparameter optimization on the performance of CoRT and other models. This demonstrates that careful selection and tuning of the hyperparameters can lead to improved model performance and reduced loss. In particular, CoRT exhibited notable sensitivity to hyperparameter adjustments and achieved the lowest loss after optimization.

The results for **RQ3** indicate that CoRT demonstrates strong detection performance for both class- and method-level code smells. However, the performance varied across the datasets, suggesting the influence of project size and characteristics. Larger projects tend to yield better detection results, whereas smaller projects may pose challenges for accurate code smell detection. **RQ4** results indicate that pre-training is a time-consuming process, with CoRT and ANN having the shortest average training times of 6 h. In contrast, the fine-

tuning process is relatively faster, requiring an average tuning time of 1.3 to 2.02 minutes across various code smells.

The results from **RQ5** demonstrate that the proposed CoRT approach outperforms the feature-based model in detecting method-level smells, whereas both models perform competitively for class-level smells. In comparison, the supervised learning baseline model struggled to achieve high detection performance across all code smells. These findings highlight the effectiveness of CoRT in enhancing the detection performance compared with feature-based and supervised learning approaches. **RQ6** results, focusing on cross-project evaluation, demonstrate the generalization capability of CoRT compared to feature-based and baseline models in detecting code smells on unseen datasets. These findings highlight the effectiveness of the CoRT in detecting code smells in new projects.

CHAPTER 5

Automated Detection of Class Diagram Smells Using Self-Supervised Learning

This chapter introduces MoRT (Model Representations with Transformers) as a design smell detection approach that utilizes the self-supervised paradigm. In software engineering, identifying design smells in an early stage of the development process, i.e., before coding or implementation begins, plays an important role in ensuring the quality and efficiency of developing software systems [23]. In Model-Driven Development (MDD), the class diagrams serve as a foundational component of the modeling process. Therefore, addressing design smells at the class diagram level is essential because it will decrease development costs and increase agility [23].

The main advantage of this approach is decreasing the time of constructing manually annotated datasets, thus increasing efficiency, whereas software engineers can devote their time to more important tasks. Moreover, the proposed approach is expected to increase the generalization of the detection model that will be trained utilizing unlabeled UML class diagrams. We present the following contributions:

- Proposing a self-supervised learning approach, named MoRT, to detect class diagram smells using deep learning.
- Evaluating MoRT for detecting four class diagram smells, including Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife.

- Conducting a comparative analysis of MoRT against feature-based and supervised learning approaches.

The rest of this chapter is structured as follows: A motivating example is provided in Section 5.1. Section 5.2 provides a summary of the related literature. The data collection process is presented in Section 5.3. In Section 5.4, we provide a comprehensive description of our self-supervised methodology. Section 5.5 focuses on the empirical evaluation of our method. The detection performance results for our proposed approach are discussed in detail in Section 5.6. Section 5.7 discusses the implications and limitations of the study, while Section 5.8 delves into the identified threats to validity. Finally, we present the conclusion in Section 5.9.

5.1 Motivating Example

To illustrate the importance of design smell detection at this architectural level and differentiate it from traditional code smell detection, consider the following scenario. Suppose a software development team is designing a software system for an e-commerce platform, and they have a class diagram representing various components of the system. In the class diagram, as shown in Figure 30, they identify the following classes:

- **OrderProcessing**: A class responsible for processing customer orders, handling payments, and managing order status.
- **Shipping**: A class responsible for managing shipping and delivery logistics.
- **CustomerManagement**: A class responsible for customer registration, login, and profile management.

- **ProductCatalog:** A class responsible for managing the catalog of products available for purchase.
- **EmailNotification:** A class responsible for sending email notifications to customers regarding order status and updates.

In a well-designed class diagram, each class should have a specific and well-defined responsibility. However, in this example, we notice that the `OrderProcessing` class is overloaded with multiple responsibilities, including order processing, payment handling, and order status management. This is a classic example of the Functional Decomposition (FD) design smell. The FD design smell occurs when a class takes on too many functions or responsibilities. In this scenario, we should decompose the `OrderProcessing` class into separate classes, each responsible for a specific functionality. While detecting design smells at the code level improves the overall software quality, identifying these design issues at an early stage, at the class diagram level, not only enhances software quality but also saves the developer's time and effort and facilitates the development process. This example shows the importance of detecting design smells in class diagrams, particularly within the framework of Model-Driven Development (MDD), motivating our research on leveraging self-supervised learning for effective and efficient design smell detection.

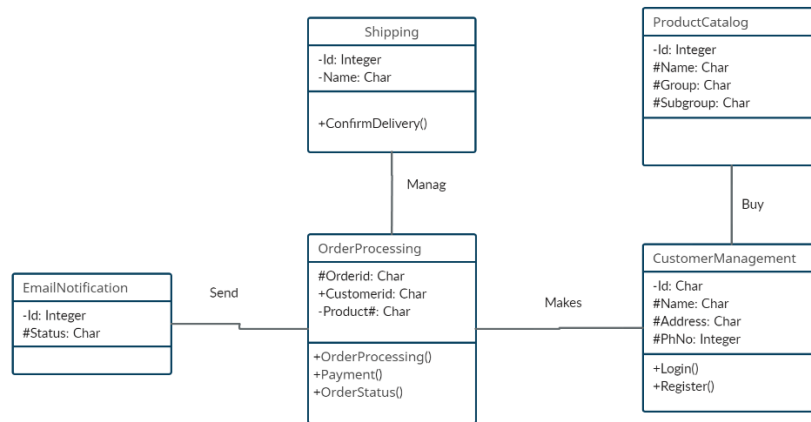


Figure 30 Online shopping class diagram

5.2 Related Work

The detection of bad smells can be done at the code level [15], [140], [172], [176] for purposes of software re-engineering and at the model level [25] [177] for improving design quality. In this section, we review the most relevant studies on bad smell detection at the model level. A few studies have used UML models to detect bad smells using various approaches, including metrics-based, rule-based, text-based, tree-based, and machine learning-based, which will be discussed in this section.

Fourati et al. [178] presented a metric-based approach to identify anti-patterns through the class and sequence diagrams. They extracted a set of quality metrics to detect five anti-patterns Blob, Lava Flow, Functional Decomposition, Poltergeists, and Swiss Army Knife. Besides the structural and behavioral information extracted from the two UML diagrams, they extracted semantic features using WordNet.

Moha et al. [17] proposed a rule-based approach to detect both code and design smells using a domain-specific language (DSL). This language allows the experts and engineers to define the rules using software metrics to detect bad smells. They used the proposed approach to detect four smells Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife. Ten Java projects were used to evaluate the detection performance. Similarly, Maddeh and Ayouni [179] presented a rule-based approach to detect 16 design smells from class and sequence diagrams by using gradual rules. However, instead of defining thresholds for 32 object-oriented metrics, they used GRITE algorithm to define the thresholds.

Alalfi et al. [180] proposed a text-based approach to detect clones in sequence diagrams. They transformed the diagram into an XMI text representation and leveraged NICAD to process the detection in the reverse-engineered sequence diagram. They evaluated the detection performance in terms of precision and recall on seven sequence diagrams. Misbhauddin and Alshayeb [181] also proposed a text-based approach to detect various design smells at the model level using XMI representation. They proposed a multi-view integrated approach to model-driven refactoring. They suggested an integrated meta-model based on one model chosen from each UML view (i.e., class diagram, sequence diagram, and use-case diagram). They evaluated their approach using eight software design examples and in terms of two performance metrics precision and recall.

Rattan et al. [182] proposed a tree-based approach for clone detection at the model level. They transformed the UML class diagram to XML tree representations and then applied tree comparison to find duplicates in two class diagrams. Ghannem et al. [183] proposed a search-based approach to detect design smells from class diagrams. Their approach uses genetic algorithms to measure the similarity between the studied system and a set of defective examples. The approach was evaluated using four Java projects to detect the three design defects, namely blob, functional decomposition, and data class. Similarly, Alshayeb et al. [184] proposed an approach for bad smell detection based on a genetic algorithm to improve sequence diagram security. The approach was evaluated using five sequence diagrams from five different systems to detect the three design defects, namely Missing Modularization, Broken Modularization, and Unutilized Abstraction.

Maneerat and Muenchaisri [185] proposed a machine learning-based for detecting bad smells from UML class diagrams. They applied seven distinct machine learning classifiers

to identify seven bad smells. They used 27 software metrics as independent variables. They evaluated their methodology on seven UML class diagrams. Sidhua et al. [186] also utilized machine learning to detect bad smells from UML class diagrams. They used Artificial Neural Networks (ANN) and software metrics to detect Functional Decomposition. They evaluated their approach using 500 class diagrams. Madden et al. [187] proposed the use of a decision tree classifier to detect bad smells. They have considered 15 Object-Oriented metrics to detect Blob, Data class, Lazy class, and Feature Envy from UML class diagrams. They evaluated their method using 7 Java systems. Barriga et al. [188] presented a reinforcement learning approach to detect bad smells from Ecore class diagrams. They utilized the Q-learning algorithm to detect Missing Modularization, Broken Modularization, and Unutilized Abstraction bad smells. They evaluated their approach to 555 Ecore models.

All these approaches significantly contributed to bad smell detection. However, most of the deep learning-based studies have detected bad smells at the code level [24]; only a few studies detected bad smells from UML models, yet they used the supervised learning paradigm. Compared to the implementation phase, it is more difficult to detect bad smells at the design stage because of the lack of information in the models compared to the code [25].

Table 37 summarizes the previous studies compared to the proposed work. Previous approaches utilized pre-defined rules, which require setting a threshold for each metric. Recent approaches utilized genetic algorithms and ML algorithms to automate the detection. However, these approaches follow a supervised learning paradigm that depends on labeled datasets and hence suffers from generalization issues [22]. Therefore, in this

study, we utilize self-supervised learning to automatically learn deep representations of the features using DL models. The benefit of representation learning is to capture the underlying structure and semantics of the features. Self-supervised learning is a new learning approach that utilizes unlabeled data. This approach eliminates the need for manual annotations, improving detection efficiency and reducing costs. Leveraging the Transformer's capabilities for sequential data processing and parallel computation enhances model training and evaluation.

Table 37 Summary of previous approaches for design smell detection

Year	Bad smells	Design Model	Approach	Technique	Input
Fourati et al., 2011 [178]	Blob, LF, FD, PG, SAK	UML class diagrams, UML sequence diagrams	Metric-based	NA	Software metrics, Semantic features (WordNet)
Moha et al., 2010 [17]	Blob, FD, SC, SAK	PADL meta-model (Pattern and Abstract-level Description Language)	Rule-based	NA	Software metrics
Maddeh and Ayouni, 2015 [179]	Blob, SAK, LF, PG, FD, GP, GC, LPL, DCL, DCH, SS, LC, FE, CMT, DC, RB	UML class diagram, UML sequence diagram	Rule-based	Gradual rules	32 OO metrics
Alalfi et al., 2016 [180]	Clone	UML sequence diagrams	Text-based	NA	XMI
Misbhaudhin and Alshayeb, 2017 [181]	FD, DU, SC, RB, LC, DC, IM, UF, SP, CF, MP, EA, SW	UML class diagram, UML sequence diagram, UML use case diagram	Text-based	NA	XMI, software metrics, pre-defined rules
Rattan et al., 2012 [182]	Clone	UML class diagrams	Tree-based	NA	XML Tree
Ghannem et al., 2015 [183]	Blob, DF, DC	UML class diagrams	Search-based	GA	Software metrics
Alshayeb et al., 2020 [184]	MMO, BM, UA	UML sequence diagrams	Search-based	GA	Software metrics
Maneerat and Muenchaisri, 2011 [185]	LC, FE, MM, MC, LM, LPL, SWS	UML class diagrams	ML-based	RF, NB, LR, IBI, IBk, VFI, J48	27 software metrics
Sidhua et al., 2020 [186]	FD	UML class diagrams	ML-based	ANN	Software metrics
Maddeh et al., 2021 [187]	Blob, DC, LC, FE	UML class diagrams	ML-based	DT	15 OO metrics
Barriga et al., 2021 [188]	CAC, DFH, DFC, DEC, RCR, ASCS, ACC, CBH	Ecore class diagrams	RL-based	Q-learning algorithm	Ecore model
This work	Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife	UML class diagrams	DL-based	Transformer, CNN, LSTM, ANN	XMI

Abstract concrete class (ACC), Broken Modularization (BM), Concrete abstract class (CAC), Creeping Featurism (CF), Comments (CMT), Data Class (DC), Data Clumps (DCL), Dead class (DEC), Divergent Change (DCH), Duplication (DU), Duplicated features in classes (DFC), Duplicated features in hierarchy (DFH), Excessive Alternation (EA), Feature Envy (FE), Functional Decomposition (FD), God Classes (GC), God Package (GP), Lazy Class (LC), Long Method (LM), Long Parameter List (LPL), Lava Flow (LF), Message Chains (MC), Middle-Man (MM), Missing Modularization (MMO), Multiple Personality (MP), Poltergeists (PG), Redundant Container Relation (RCR), Refused Bequest (RB), Shotgun Surgery (SS), Spider's Web (SW), Specters (SP), Swiss Army Knife (SAK), Switch Statement (SWS), Undue Familiarity (UF), Unutilized Abstraction (UA)

5.3 Dataset Preparation

In this section, we discuss the construction of the model smells datasets that are used in pre-training and fine-tuning. Unlabeled data (Section 5.3.1) were used in the pre-training phase, whereas labeled data (Section 5.3.2) were used for fine-tuning. The constructed datasets are available in our online appendix⁹.

5.3.1 Unlabeled data collection

Our pre-training dataset was collected using more than 100k UML models in XMI representation. We utilized two publicly available software model datasets described below. We focus in this study on detecting design smells from class diagrams, so we only extracted the XMI representation of UML class diagrams from the following datasets.

*The Lindholmen Dataset*¹⁰. Hebig et al. [189] collected UML models from GitHub projects using a semi-automated approach. They provided a list of GitHub URLs of more than 24k projects that contain at least one UML model as an image, XMI, or UML file. We automated the collection process by cloning the project and then extracted solely XMI files using Python. Many UML models were provided in image format, which was excluded from our dataset. In addition, we were not able to access all projects, as some projects were deleted; however, we extracted a total of 4276 XMI files.

*ModelSet*¹¹. López et al. [190] collected more than 500k software models of different types, involving UML, Ecore, Archimate, etc. We used only the UML models dataset, which was

⁹ <https://github.com/amalazba/Automated-Detection-of-Class-Diagram-Smells-Using-Self-Supervised-Learning>

¹⁰ <http://www.models-db.com/>

¹¹ <https://models-lab.github.io/blog/2021/modelset/>

extracted from the GenMyModel website, and excluded other models, such as Ecore models, which is a modeling framework and part of the Eclipse Modeling Framework (EMF) that has its own modeling language. It is worth mentioning that some files failed to process due to encoding errors. We extracted a total of 96,468 UML diagrams in XMI representation. All data are available in the online appendix.

5.3.2 Labeled datasets construction

The overall process for creating the labeled datasets is shown in Figure 31. First, we collected publicly available datasets. Second, we reverse-engineered open-source systems to extract the UML class diagram. Third, we parsed the XMI files to extract entities (i.e., classes). Finally, we mapped the extracted entities with their labels.

Labeled datasets. We collected the labeled datasets (i.e., ground truth) using publicly available datasets; the datasets were chosen based on 1) the presence of one of the four design smells that are investigated in this study, and 2) the datasets should have a full file path and the class name in order to match the labels with the extracted XMI. We found two datasets that meet our conditions listed below:

- Khomh et al. [191] manually annotated three design smells (Blob, Functional Decomposition, and Spaghetti Code) by undergraduate and graduate students¹².
- Moha et al. [17] identified the occurrence of four design smells using DETEX; they identified various types of smells, including Blob, Functional Decomposition,

¹²https://www.ptidej.net/tools/designsmells/materials/Occurrences/080920%20-%20BDTEX%20-%20QSIC09%20Paper/plfng_view

Spaghetti Code, and Swiss Army Knife. Then, they manually validated the results by independent engineers¹³.

Reverse engineering. A set of Java projects was identified during the collection of labeled datasets. For each project, 1) we downloaded the source code, then 2) we reverse-engineered the source code to get the corresponding UML class diagram, and finally, 3) we exported the diagram as an XMI representation.

XMI parsing. We generated the XMI files for each project in the previous step. In this step, we parsed the XMI files to traverse over all classes within the class diagram, and for each class, we extracted 1) the class name, 2) the package name (i.e., the path to the class), and 3) XMI representation of the class. The final output is a CSV file for each project, where each instance represents a class that has the following attributes: project name, project version, package name, and class name.

Match labels with XMI. We used the labeled datasets and mapped each entity to its XMI representations extracted using reverse engineering. The final output is the fine-tuning datasets where each dataset has labeled instances, and every instance has the following attributes: project name, project version, package name, class name, label (1 for smelly, 0 for non-smelly), and XMI representation. It is worth mentioning that we removed the duplicates when two instances are identical and have the same label. However, in case we have two identical instances but with different labels, we kept them both.

A description of the 11 fine-tuned datasets is shown in Table 38. The table displays the overall number of smelly and non-smelly instances for each dataset. Due to SMOTE's

¹³https://www.ptidej.net/tools/designsmells/materials/Occurrences/100720%20-%20DETEX%20-%20TSE10%20Article/plfng_view

reliance on the K-Nearest Neighbor algorithm, which generates synthetic instances from the k-nearest examples, datasets with few labeled occurrences (less than six positive instances) were excluded.

Table 38 Fine-tuning datasets description

Project	Blob			FD			SC			SAK		
	Smelly	Non-smelly	Total	Smelly	Non-smelly	Total	Smelly	Non-smelly	Total	Smelly	Non-smelly	Total
ArgoUML	129	1077	1206	92	1065	1157	70	1067	1137	246	999	1245
Azureus	99	1408	1507	107	1395	1502	196	1297	1493	220	1304	1524
GanttProject	113	357	470	124	339	463	51	350	401	24	180	204
JHotDraw	18	158	176	176	0	176	18	158	176	-	-	-
log4j	4	169	173	13	161	174	4	170	174	52	132	184
Lucene	10	234	244	2	236	238	15	230	245	27	228	255
Nutch	18	338	356	43	338	381	27	676	703	64	338	402
PMD	9	423	432	33	423	456	10	423	433	26	423	449
QuickUML	-	-	-	10	146	156	4	146	150	12	146	158
Xerces	29	179	208	8	185	193	48	165	213	30	177	207
Xerces2	341	471	812	51	488	539	271	869	1140	115	472	587
Grand Total	770	4814	5584	659	4776	5435	714	5551	6265	816	4399	5215

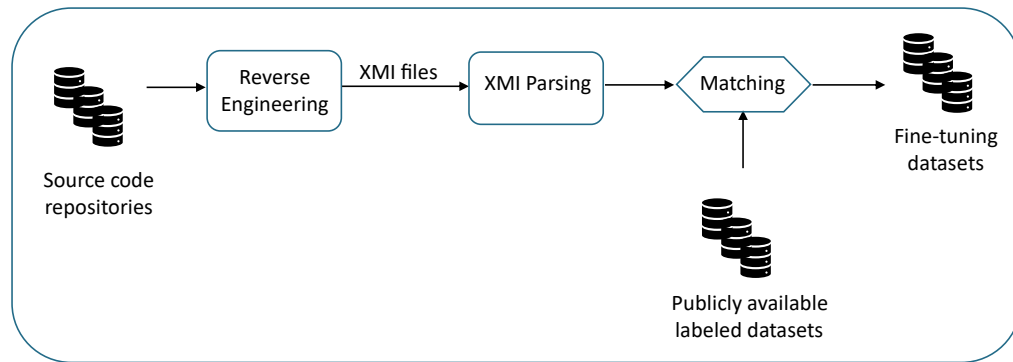


Figure 31 Fine-tuning datasets construction

5.4 MoRT

This section introduces a novel self-supervised learning approach, called MoRT, for detecting design smells in class diagrams. MoRT differs from traditional supervised approaches as it does not rely on manually labeled data, making it more scalable and cost-effective. MoRT leverages unsupervised semantic feature learning to automatically learn meaningful representations of class diagrams without the need for manual annotations. This is a departure from conventional supervised methods that depend on labeled data. We present the implementation details of MoRT in this section. An overview of our framework is shown in Figure 32. As illustrated in the figure, there are two main steps: 1) pre-training, where we utilize the unlabeled datasets to train various DL models on pre-defined tasks, and 2) fine-tuning, in which several tasks are considered to fine-tune the pre-trained model. We describe each step in the following subsections.

5.4.1 Pre-training MoRT

In the pre-training phase, we collected unlabeled data that constitute more than 100k UML models in XMI representation. Then, we preprocessed the data to transform textual features into numerical features using four techniques: tokenization, filtering, truncation, and vectorization. Next, we trained the model on a proxy task where we created pseudo labels from the unlabeled data; the model was trained to predict the pseudo labels. Finally, we performed hyperparameter optimization using four different DL architectures and multiple hyperparameters. In the following subsections, we first present the defined task used for training; then, we discuss each step.

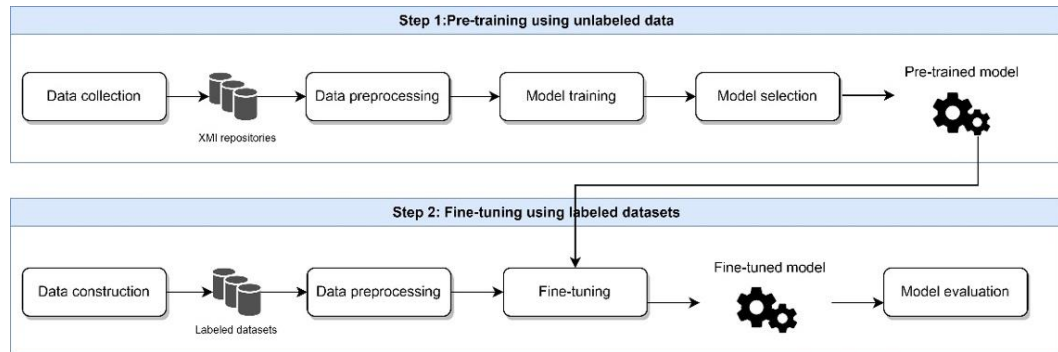


Figure 32 Overview of MoRT

Task overview

To detect design smells using unlabeled data, we need to create pseudo labels (i.e., artificial labels) to enable the DL model to learn the structural and semantic features of the UML Class Diagram model. MoRT is trained to predict a set of masked words applied to a part of the XMI representation of the UML Class Diagram that is given to it as input. Therefore, we need to define a proxy task (i.e., the task used for pre-training) that would facilitate learning intermediate data representations.

Assuming the unlabeled dataset D consists of m instances where a single instance i is associated with x value, which is a feature vector of length n tokens, and each instance i represents a whole class/method or part of a class/method. We created the pseudo labels by randomly masking a set of keywords, i.e., for each instance $i \in m$, for each token $t \in n$ replace 25% of n with the $MASK_{ID}$ if and only if it is a keyword; otherwise, keep the original token t . The keywords consist of the tags in the XMI files as well as some attributes. A complete list of the keywords is in the online appendix ¹⁴. To minimize the bias, we also randomly masked tokens that do not belong to the keywords list. To construct a new data set with the masked instances as a new instance with the original value as labels, i.e., a new feature vector x' is created as follows: for each instance, i in D , create a new instance i' (x^*, y) such that (1) $x^* = x'$ where x' is the new feature vector created in previous steps (masked), and (2) $y = x$ where x is the original feature vector, we call y as pseudo labels.

¹⁴<https://github.com/amalazba/Automated-Detection-of-Class-Diagram-Smells-Using-Self-Supervised-Learning>

The proposed method of masking keywords instead of random tokens in the pre-training phase offers several advantages. Firstly, this approach ensures that the model learns to recognize the syntactic structure of UML class diagrams by focusing on keywords, which are essential in defining the diagram's structure. By localizing and understanding these keywords in context, the model can effectively learn the structural and semantic features of the targeted UML model, resulting in a representation with good fidelity. Additionally, using masked keywords reduces ambiguity in prediction compared to masking random tokens, as other variables like identifier names, class names, and package names in UML models often follow various naming conventions, making the task less clear. Lastly, the proposed proxy task provides a simple yet effective way to capture the structural and semantic features without the need for complex transformations or additional software metric extraction tools, thereby streamlining the pre-training process. These advantages collectively enhance the effectiveness and efficiency of the proposed method.

Preprocessing

During the preprocessing phase, each instance (i.e., class) undergoes the following steps: first, tokenization, where the instance is divided into individual words or tokens. Next, filtering, where instances containing less than 50 tokens are removed from the dataset. Then, truncation is applied to set the maximum length of an instance to 256 tokens. For instances exceeding this limit, instead of discarding the extra tokens, they are utilized to create multiple instances, each with a length of 256 tokens. The total number of instances deleted was 103 instances, and the total number after filtering equals to 100K instances. Subsequently, text vectorization is performed with a vocabulary size of 30,000, assigning a distinct numerical representation to each unique token. This transformation allows all

instances to be converted from text-based to numerical representations, facilitating the modeling processes.

Model training

Four architectures were used for comparison purposes, which are Transformer, CNN, LSTM, and ANN. The last three models were chosen because they have been tested frequently and have produced positive outcomes in detecting bad smells [24], [141], [151]. However, we considered a multi-layer Transformer introduced by Vaswani et al. [165] because it has several advantages. When comparing the Transformer to LSTM, the Transformer offers several advantages. Unlike LSTM, which computes its parameters sequentially, Transformer computes them in parallel, making it more computationally efficient. Both Transformer and LSTM are capable of processing sequential data, such as text, effectively. However, the Transformer is considered more efficient than LSTM due to its parallelized computation. On the other hand, Transformer shares a similarity with CNN in that they both utilize parallel computations. However, it's important to note that while CNN is typically employed for processing image data efficiently, it may not perform as effectively when applied to sequential data. We used a batch size of 64, a maximum sequence of 256 tokens, and five epochs. For the optimization algorithm, we utilized Adam with various learning rates. We used the Xavier uniform initializer to initialize the weights. Finally, we had a dropout rate between 0.1 and 0.3 across all layers.

The input is a sequence of the XMI representation that is a class in the UML class diagram. Each sequence consists of a set of tokens, and we used token embedding and positional embedding to represent each token in a sequence. For token embedding, we employed the word embedding provided by TensorFlow [192] with 30,000 vocabulary size and two

special tokens ([unk] to represent out of vocabulary tokens, and [mask] to represent masked tokens). In token embeddings, each token (represented as a positive integer) is mapped to a dense vector of a fixed size (i.e., embedding dimension). Every token in a sequence has a different location, which is represented by positional embedding. The output of the model is a prediction of the masked tokens value that is a probability matrix of size (sequence length x vocabulary size) calculated using the *softmax* function:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^k e^{x_j}}$$

Model selection

The selection of the best model was achieved using the hyperparameter optimization technique. We considered the four DL architectures along with the following hyperparameters: the number of layers (expressed as L) including 3, 6, and 12 layers, embedding dimension (E) with 128, 256, and 512, number of units in hidden layers (H) including 128, 256, 512 and 1024, 2048 units, number of self-attention heads (A) with two values 8 and 16, learning rate (LR) with three values 0.0001, 0.001 and 0.01, and the dropout rate (DR) with 0.1, 0.2 and 0.3 values. It is important to note that A (i.e., self-attention heads) is only a hyperparameter of the Transformer model. In addition, the number of Transformer blocks is represented by L.

The tree-structured Parzen estimator (TPE) was used as a sampling algorithm [167] that is implemented in the Optuna library [166]. We selected this approach since alternative approaches, such as grid search, require significant processing resources when used in DL. Additionally, a well-known technique called random search has been demonstrated to provide outcomes almost comparable to those of grid search but considerably more quickly

[122]. When compared to random search, TPE is considered computationally efficient [122]. We optimized the hyperparameters for all four models (Transformer, CNN, LSTM, and ANN). For about five weeks, we ran a total of one hundred trials (25 trials for each model). The computer used to train all models was equipped with an NVIDIA GeForce RTX 4090 GPU, 64 GB of RAM, and an Intel Core i9 processor. The best model was obtained using the Transformer with the following hyperparameters 6 L, 128 E, 1024 H, and 8 A.

5.4.2 Fine-tuning MoRT

We fine-tuned the pre-trained model by 1) initializing the fine-tuned model using the parameters of the pre-trained model, then 2) starting the fine-tuning process utilizing the labeled datasets. Each task has a different model that has been fine-tuned employing its labeled datasets; however, all fine-tuned models have the same initialization parameters (i.e., the pre-trained model parameter). This has two primary benefits: 1) we can leverage the large unlabeled data that is available to be used in training, and 2) the fine-tuning procedure would not need huge datasets (i.e., it only requires small labeled datasets); hence, it will speed up training since the pre-training, which is taught once and then utilized repeatedly on different tasks, often takes a lot longer.

Tasks overview

The tasks in this phase are straightforward; they include the detection of four Class Diagram smells, which are Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife. We selected these design smells because they can be detected from the UML class diagram and are among the most investigated design smells [23]. In addition, these

smells were used in the most well-known tools like DÉCOR [17], providing a clear and standardized basis for identification and analysis. Moreover, our selection was guided by the availability of publicly accessible labeled datasets for these specific design smells.

Preprocessing

In addition to the preprocessing techniques discussed in the previous section, we used the Synthetic Minority Oversampling Technique (SMOTE) to mitigate the bias towards non-smelly instances in the dataset and ensure a more balanced representation of both classes. SMOTE is a widely used method for increasing the number of minority class instances by generating synthetic samples that are similar to the existing minority class instances; this technique is implemented in the Scikit-learn library.

Fine-tuning

The fine-tuning model is a simple deep learning model, which is a feed-forward neural network with two layers. We used a batch size of 16, a maximum sequence of 256 tokens, and five epochs. For the optimization algorithm, we utilized Adam with a 0.001 learning rate. We used the pre-trained model to initialize the weights. Finally, we set the dropout rate to 0.1 across all layers. To calculate the loss, we utilized binary cross entropy.

The input to the fine-tuning model is a set of instances where each instance is the XMI representation of a class in the UML class diagram. The output of the fine-tuned model is 0 or 1, where 0 means this class is not smelly and 1 means smelly. The output is calculated utilizing the *sigmoid* function as follows:

$$\text{sigmoid}(x_i) = \frac{1}{1 + e^{-x_i}}$$

5.4.3 Comparison models

In this study, we compared MoRT with the feature-based and baseline approaches, which were built using an Artificial Neural Network (ANN) model. This choice was made to ensure a fair and consistent comparison since we utilized ANN as the underlying model for fine-tuning. By employing the same model architecture, we aimed to isolate the impact of the different input data sources rather than the variations in model types.

Supervised learning. We compared the proposed approach with the supervised learning approach. We trained a baseline model where the model architecture is a simple ANN model with the same hyperparameters used for fine-tuning, i.e., we used Adam as an optimization algorithm with a learning rate of 0.001. We utilized Xavier uniform initializer to initialize the weights and a dropout rate of 0.1. The labeled datasets were used to detect the four code smells. The input/output representations, as well as the training procedure used to build the model, are exactly like fine-tuning.

Feature-based. We conducted a comparison between our approach and a feature-based method. In the feature-based approach, we utilized the features extracted from MoRT during the pre-training step without further fine-tuning using labeled datasets and fed them as input directly into an ANN model. Subsequently, we employed labeled datasets to evaluate this approach. ANN model was trained using the same input/output representations, training procedure, and hyperparameters used for fine-tuning and baseline model.

5.5 Experimentation

The experiments conducted to assess the approach are explained in this section. The experiment planning outline for evaluating the MoRT for detecting various model smells is as follows:

- 1) Defining the goal and RQs: We defined a set of RQs in section 5.5.1.
- 2) Datasets Preparation: Details of the used datasets are in section 5.3. The independent variables are the source code, while the independent variable is smelly or non-smelly. We preprocessed the data as described in section 5.4.
- 3) Performance Metrics: We selected a set of metrics listed in section 5.5.2.
- 4) Validation Method: We used a train-test-split method with an 80/20 ratio for pre-training and a 70/30 ratio for fine-tuning more details are in section 5.5.3.
- 5) Baseline Comparisons: We compared our approach with two approaches supervised and feature-based approaches (section 5.4.3).
- 6) Hardware and Software: We used a PC with the following specifications: NVIDIA GeForce RTX 4090 GPU, 64 GB of RAM, and an Intel Core i9 processor. We implemented our approach using Python. We used the TensorFlow framework for developing and training the deep learning models.
- 7) Data Analysis: Finally, we analyzed the collected results using various methods such as tables and boxplots.

5.5.1 Research questions

The objective of our experiment is to assess MoRT’s performance in identifying four design smells: Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife.

We articulated three research questions:

- RQ1: What is the best architecture and hyperparameters to learn representations from class diagrams?
- RQ2: What is the detection performance of MoRT for detecting class diagram smells?
- RQ3: To what extent does MoRT outperform state-of-the-art approaches?

To answer RQ1, we utilized four DL architectures, including Transformer, CNN, LSTM, and ANN. We selected architectures that were used in previous studies and have produced positive results, such as CNN, LSTM, and ANN [24], [141], [151], as well as architectures that have not been investigated before in design smell detection, like Transformer. In addition, we considered the diversity when selecting the models as each model belongs to different categories of deep learning models, and each category has its strengths and characteristics. LSTM belongs to recurrent networks, which is known for its ability to model sequential data effectively, while CNN belongs to convolutional networks, which is known for its ability to process spatial data. ANN feed-forward networks, which is the basic model for all other models. While other deep learning models are certainly valuable and may be considered in future research, our selection aimed to strike a balance between diversity, frequency, and the proven efficacy of these architectures in bad smell detection tasks. The detection performance of the pre-training models is examined and evaluated based on the different architectures. Moreover, we explored the impact of the model size

and training hyperparameters on the detection performance using the hyperparameter optimization method.

For RQ2, we examined how well the proposed model performs in terms of multiple performance metrics (e.g., precision, recall, ... etc.) to identify four design smells Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife. A comparison of the detection performance of our approach against state-of-the-art methods is presented in this RQ. In this RQ, we selected the best model obtained by RQ1, i.e., the Transformer, as the primary model. Along with model performance, model complexity is an important consideration when selecting a model [123], [170], as complex models may require additional resource consumption (such as storage and computational time costs). Since DL models are complex models, it is crucial to take their time complexity into account when assessing their performance. One of the key benefits of self-supervised learning is the ability to train a single complicated DL model that can be utilized again on several multiple tasks, employing less complex fine-tuning models that can be trained quickly. As a result, this RQ would quantitatively show how long the MoRT pre-training and fine-tuning phases take.

To answer RQ3, we compared MoRT against two state-of-the-art approaches, feature-based and supervised learning approaches. Moreover, we expanded the comparison to cross-project evaluation, where we assessed the three approaches on unseen projects to evaluate their resilience and generalizability.

5.5.2 Evaluation metrics

We selected a set of evaluation metrics to assess the detection performance of our approach. We selected the most frequently used metrics in DL-based bad smell detection, including Accuracy (Acc), Precision (P), Recall (R), F1-score, Matthews Correlation Coefficient (MCC), and Area Under Curve (AUC).

5.5.3 Validation technique

As a validation technique, we utilized the stratified *train-test-split* method, in which the data is randomly divided into two stratified subsets: one is used for training, and the other is used for testing. Although cross-validation is known for providing a more robust assessment, deep learning models can be computationally intensive, and implementing cross-validation may significantly increase the time required for evaluation. Also, in our study, we performed hyperparameter optimization, which is computationally intensive. Given these considerations, we opted for a train-test split approach to balance the need for a reliable evaluation and computational efficiency.

The pre-trained models were validated using an 80:20 split, in which 80% of the dataset is used to train the model, and 20% of the dataset is utilized for testing. In the fine-tuning step, we used 70:30 validation techniques, in which 70% of the dataset is used to train the model and 30% for model testing. The split ratio selection was attributed to two factors. First, consistent with a recent systematic literature review [24], these ratios are among the most used validation techniques. Second, the dataset size (i.e., the number of samples) was taken into consideration; given the large size of the pre-training datasets, 20% of the data is enough for validation; on the contrary, the dataset size of the fine-tuning is relatively small; hence 30% of the data is chosen for validation.

5.6 Experimental Results

This section discusses the experimental results to answer the research questions defined in section 5.5.1.

5.6.1 The effect of model architectures and hyperparameters

Model architecture. Table 39 shows the loss and accuracy of MoRT compared to ANN, CNN, and LSTM models. All models were trained on the proxy task discussed in Section 5.4.1. We selected the hyperparameters manually as follows: 128 E, 64 B, 0.001 LR, 0.1 DR, 128 H, 3 L, 8 A. We observe from the tables that MoRT achieved the best results with an accuracy of 0.96 and loss of 0.04, followed by the LSTM with an accuracy of 0.94 and loss of 0.06, on the validation set. The worst results were obtained by ANN with an accuracy of 0.46 and a loss of 0.62. CNN is doing well and close to LSTM with an accuracy of 0.93 and a loss of 0.06.

Table 39 Pre-trained models performance		
Model	Loss	Accuracy
ANN	0.62	0.46
CNN	0.06	0.93
LSTM	0.06	0.94
MoRT	0.04	0.96

Hyperparameter optimization. For all pre-trained models (i.e., MoRT, CNN, LSTM, ANN), we conducted hyperparameter optimization utilizing Optuna. MoRT was the most affected model by hyperparameter optimization, while ANN and LSTM models were not sensitive to hyperparameter optimization. CNN was less sensitive to hyperparameter optimization compared to MoRT. The best hyperparameters for each model are shown in

Table 40, along with loss before and after hyperparameter optimization. All models achieved higher performance after hyperparameter optimization with a decrease in the model loss; however, MoRT maintained the best performance (i.e., lowest loss value) with 0.03206 loss. CNN was the second-best model with a loss of 0.03222, ANN achieved the worst results with a loss of 0.62013, and LSTM was the second-worst model with a 0.04913 loss. The best hyperparameters of MoRT were 0.3 DR, 0.001 LR, 6 L, 128 E, 1024 H, and 8 A, which were selected as the pre-trained model to perform fine-tuning on the labeled datasets.

The results of all runs are in the online appendix. We notice from the results that there is no direct relationship between the dropout rate and the loss; however, in two models (i.e., MoRT and LSTM), a larger dropout rate produced the best results. Moreover, we observe from the table that lower learning rates produce better results (i.e., low loss value). Also, an inverse relationship was found between the embedding dimension and layers dimension (i.e., the number of hidden units) where low embedding dimension with high layers dimension archived the best results.

Table 40 Pre-trained model's best hyperparameters (MoRT, ANN, CNN, and LSTM)

Pre-trained model	DR	LR	L	E	H	A	Loss	Loss (B)
ANN	0.2	0.0001	3	128	1024	-	0.62013	0.62029
CNN	0.1	0.0001	12	256	2048	-	0.03222	0.06165
LSTM	0.3	0.001	3	256	2048	-	0.04913	0.05708
MoRT	0.3	0.001	6	128	1024	8	0.03206	0.03938

RQ1 Answer. MoRT outperformed the other models with an accuracy of 0.96 and a loss of 0.04 on the validation set. LSTM also performed well, with an accuracy of 0.94 and a

loss of 0.06. CNN demonstrated competitive performance, while ANN performed the worst among the models. MoRT was the most sensitive to hyperparameter optimization, achieving a loss of 0.03206 after optimization, the lowest among all models. There was no direct relationship between the dropout rate and loss, although larger dropout rates performed better in MoRT and LSTM. Lower learning rates tended to produce better results, resulting in lower loss values. An inverse relationship was observed between the embedding dimension and layer dimensions, where lower embedding dimensions combined with higher layer dimensions yielded the best results.

5.6.2 The detection performance of MoRT for detecting class diagram smells

Detection performance. We fine-tuned MoRT on four tasks (i.e., the detection of four design smells), which are the Blob, Functional Decomposition (FD), Spaghetti Code (SC), and Swiss Army Knife (SAK). Table 41 presents the detection performance of the four design smells in terms of ACC, P, R, F1, AUC, and MCC evaluation measures. We used F1, AUC, and MCC as our main metrics in the discussion. The detection performance of MoRT on Blob achieved a score of F1 ranging from 82.24 to 96.59, AUC from 90.39 to 99.45, and MCC from 64.49 to 93.14. For FD, our approach achieved F1 between 74.49-96.13, AUC between 82.66-99.1, and MCC between 51.14-92.17. The detection performance of SC attained by MoRT achieved a score of F1 from 77.2 to 95.86, AUC from 91.8 to 98.82, and MCC from 66.47 to 91.65. For SAK, it achieved a score of F1 between 79.81-94.03, AUC between 85.5-98.15, and MCC between 58.07-88.55. Although the model performs well on the detection of Blob and FD, we notice a slight degradation in the performance when detecting SC and SAK design smells due to the complexity of these smells, as they might need additional features. Different design smells may require

distinct sets of features to be effectively detected. Additional features extracted from sequence diagrams and/or use case diagrams might increase the detection performance.

We noticed that MoRT performed well (i.e., not among the bottom two datasets) on some datasets across all design smells, such as JHotDraw Azureus, PMD, and ArgoUML, while it had the worst performance on other datasets, such as Xerces2 across multiple design smells. Moreover, we observed that there is no direct relationship between the dataset and the type of design smell on the detection performance. For example, MoRT performed well for detecting SAK in the Lucene dataset but failed to maintain compatible performance in detecting SC for the same dataset. On the other hand, it scored poorly for detecting SAK in Nutch but performed well for detecting SC for the same datasets.

In the Nutch dataset, the model performed well on detecting SC despite having only 27 smelly instances, while it performed poorly on detecting SAK, which has almost double the number of smelly instances (i.e., 64). On the contrary, the model scored poorly for detecting SC in the Lucene dataset with 15 smelly instances, while it performed well for detecting SAK, which has almost double the number of instances (i.e., 27). This indicates that the detection performance is not sensitive to the number of positive (smelly) instances.

After examining the detailed results, we observe that the consistency between AUC and MCC is not uniform across all datasets and smell types. For example, Xerces2 in Blob smell detection shows a moderate AUC and MCC, suggesting reasonable but not exceptional model performance. In contrast, Lucene for SC smell detection shows a high AUC but a moderate MCC, indicating good class separation but potential issues in balanced prediction accuracy. The results suggest that the model, despite its ability to

separate classes, might not be equally accurate in all classification scenarios, possibly due to dataset-specific factors. These variations highlight the importance of using multiple metrics to gain a comprehensive understanding of a model's performance.

Table 41 Design smell detection performance

Dataset	Blob						FD					
	ACC	P	R	F1	AUC	MCC	ACC	P	R	F1	AUC	MCC
ArgoUML	85.76	82.91	90.09	86.35	95.39	71.79	88.44	86.39	91.25	88.75	93.12	77
Azureus	90.43	89.22	91.96	90.57	94.59	80.89	92.48	93.84	90.93	92.36	97.1	85.01
GanttProject	82.24	82.24	82.24	82.24	92.08	64.49	75.49	77.66	71.57	74.49	82.66	51.14
JHotDraw	94.79	93.88	95.83	94.84	99.31	89.6	-	-	-	-	-	-
log4j	-	-	-	-	-	-	89.8	86.79	93.88	90.2	97.67	79.86
Nutch	96.57	96.12	97.06	96.59	99.45	93.14	86.27	84.26	89.22	86.67	92.69	72.67
PMD	-	-	-	-	-	-	96.06	94.66	97.64	96.13	99.1	92.17
Xerces	90.74	86.67	96.3	91.23	97.99	81.99	-	-	-	-	-	-
Xerces2	81.69	75.86	92.96	83.54	90.39	65.05	82.31	86.82	76.19	81.16	90.72	65.12
Dataset	SC						SAK					
	ACC	P	R	F1	AUC	MCC	ACC	P	R	F1	AUC	MCC
ArgoUML	91.28	93.16	89.1	91.08	96.62	82.63	90.5	90.64	90.33	90.48	93.17	81
Azureus	88.3	84.49	93.83	88.92	93.96	77.08	89.29	89.09	89.54	89.31	95.98	78.57
GanttProject	94.81	92.79	97.17	94.93	98.6	89.72	85.45	86.79	83.64	85.19	90.18	70.96
JHotDraw	93.75	97.73	89.58	93.48	96.48	87.81	-	-	-	-	-	-
log4j	-	-	-	-	-	-	83.75	90.91	75	82.19	85.5	68.56
Lucene	81.16	97.78	63.77	77.2	98.82	66.47	94.2	96.92	91.3	94.03	98.15	88.55
Nutch	95.81	94.71	97.04	95.86	98.14	91.65	78.92	76.58	83.33	79.81	91.15	58.07
PMD	-	-	-	-	-	-	86.61	87.8	85.04	86.4	93.47	73.26
QuickUML	-	-	-	-	-	-	85.23	91.89	77.27	83.95	87.71	71.36
Xerces	91	85.96	98	91.59	94.36	82.82	87.96	97.67	77.78	86.6	95.88	77.55
Xerces2	84.29	82.78	86.59	84.64	91.8	68.65	88.38	81.14	100	89.59	98.12	78.92

To further examine the effectiveness of the proposed approach, we compare the detection performance of MoRT with DÉCOR, which was proposed by Moha et al. [17], where the description of the bad smells is used to generate a set of rules to be utilized for detecting these bad smells. We selected DÉCOR since, in their study, they examined identical projects and considered the Blob, FD, SC, and SAK design smells, which is the focus of our investigations. Also, we included DÉCOR in our comparison as it is a well-known and established method in the field. Moreover, DÉCOR continues to serve as a benchmark in recent studies published in prestigious journals [141], where it is frequently employed for comparative evaluations. This deliberate choice was made to ensure that our study maintains alignment with the current state of the art standards in the domain of bad smell detection. Table 42 shows the results of our approach compared to DÉCOR in terms of the precision metrics for each design smell and project. It is worth mentioning that some datasets have not been used in the experiments due to the number of smelly instances exceeding the minimum number required by MoRT. We observed from the table that, on average, for the projects under consideration, our method provides higher precision scores than DÉCOR. However, for the Blob design smell, the precision of DÉCOR is slightly higher than MoRT, with a gap reaching 13% for Xerces and Xerces2. Nevertheless, for other datasets, such as Nutch, MoRT outperformed DÉCOR with an increase of 29%. For SC, our approach performed slightly better than DÉCOR, with an improvement of the precision of 23% in the Lucene dataset. However, as compared to DÉCOR's technique, our approach precision for the FD and SAK is high. For instance, MoRT has outperformed DÉCOR for detecting FD in the GanttProject and PMD datasets with a difference that is more than three times (78 vs. 27% and 95 vs. 31%, respectively) and more than four times

for the Nutch dataset (84 vs. 20%). Similarly, the gap reaches double for Xerces and Xerces2 in detecting SAK, i.e., 98 vs. 42 % and 81 vs. 41%, respectively. These findings may be explained by the fact that the FD, SC, and SAK smells are particularly challenging to characterize in terms of rules and metrics, and a DL-based method is much more successful in this instance, whereas Blob is more easily identified. DÉCOR and MoRT are distinct approaches with different underlying mechanisms. DÉCOR relies on a rule-based system generated from textual descriptions of design smells, while MoRT is based on deep learning and learns patterns directly from the data. This could lead to variations in their performance on different types of design smells. SAK, is a complex design smell that may not be easily characterized by a rule-based system. It may require a deeper understanding of XMI structures and interactions, which MoRT, with its deep learning capabilities, might be better suited to capture. These reasons might explain the differences in performance between DÉCOR and MoRT.

Time complexity. We calculated the time required for pre-training and fine-tuning. To calculate the pre-training time, we considered the training time for all models trained during the hyperparameter optimization process. To calculate the fine-tuning time, we considered the time for all labeled datasets per design smell. The average (Avg), minimum, (Min), and maximum (Max) times for training the models and fine-tuning the pre-trained models are summarized in Table 43 and Table 44, respectively.

The tables demonstrate that one of the key benefits of self-supervised learning, where a single complicated model is learned and then effectively employed on several tasks, is that pre-training takes longer than the fine-tuning procedure. ANN has the shortest average training times (5 hours per training), whereas LSTM has the longest average training time

(19 hours). CNN has a slightly longer training time than MoRT, i.e., around 8.5 and 8 hours, respectively. Although MoRT and ANN require the least amount of training time, MoRT performs better in terms of training accuracy and loss, while ANN performs worse (RQ1).

For fine-tuning, the average tuning time across all design smells was between 1.3-2.02 minutes. Fine-tuning is considered much more efficient than pre-training, and the average fine-tuning time for all design smells ranged from 12 to 14 seconds.

Table 42 Comparison with previous work

Dataset	ArgoUML	Azureus	GanttProject	log4j	Lucene	Nutch	PMD	QuickUML	Xerces	Xerces2
Blob										
DECOR	86%	93%	90%	100%	67%	67%	100%	100%	100%	89%
MoRT	83%	89%	82%	-	-	96%	-	-	87%	76%
FD										
DECOR	60%	39%	27%	55%	0%	20%	31%	30%	100%	52%
MoRT	86%	94%	78%	87%	-	84%	95%	-	-	87%
SC										
DECOR	86%	82%	71%	67%	75%	85%	56%	0%	92%	61%
MoRT	93%	84%	93%	-	98%	95%	-	-	86%	83%
SAK										
DECOR	17%	23%	38%	65%	11%	39%	46%	17%	42%	41%
MoRT	91%	89%	87%	91%	97%	77%	88%	92%	98%	81%

Table 43 Pre-training time (MoRT, ANN, CNN, and LSTM)

Model	Avg (min.)	Min (min.)	Max (min.)
ANN	300.67	216	429
CNN	497.22	245	913
LSTM	1129.81	468	2103
MoRT	475.41	287	853

Table 44 Fine-tuning time (Blob, FD, SC, and SAK)

Design smell	Avg (sec.)	Min (sec.)	Max (sec.)
Blob	14	6	30
FD	14	5	28
SC	15	6	28
SAK	12	6	26

RQ2 Answer. MoRT exhibited strong detection performance across various design smells, achieving high F1 scores, AUC values, and MCC scores, indicating its effectiveness in identifying class diagram smells. Complex design smells like SC and SAK showed slightly lower performance due to their complexity and potential need for additional features. Compared to DÉCOR, on average, MoRT outperformed DÉCOR in terms of precision. DÉCOR had a slightly higher precision for Blob on some datasets, but MoRT outperformed DÉCOR significantly for other design smells like FD and SAK. This suggests that MoRT's deep learning-based approach is particularly effective for challenging design smells like FD and SAK. Pre-training is time-consuming compared to fine-tuning; it took an average of approximately 8 hours to pre-train MoRT. Fine-tuning, on the other hand, was much more efficient, with an average time between 1.3 to 2.02 minutes per design smell.

5.6.3 Comparison with state-of-the-art approaches

Detection performance. In this investigation, we contrasted our method with feature-based and supervised learning methods. Using the same hyperparameters employed for fine-tuning, we trained a baseline model for supervised learning, which is an ANN model utilizing Adam as an optimization algorithm and a learning rate of 0.001. The weights were initialized using the Xavier uniform initializer with a 0.1 dropout rate. Then, we trained

and evaluated the baseline model on four design smells utilizing the same labeled datasets used for fine-tuning. Additionally, we contrast our method with the feature-based approach, in which we feed the derived MoRT features directly into the ANN model without any further tuning before making predictions utilizing the labeled datasets. Because ANN was used to fine-tune MoRT on the tasks, we chose it as a baseline.

We plotted the result of the baseline and feature-based against MoRT for all design smells in boxplots in terms of six performance metrics, i.e., ACC, P, R, F1, AUC, and MCC, as shown in Figure 33. We observe from the figure that the feature-based model competes well with MoRT; however, MoRT outperforms the features-based model across all smells by having smaller boxes and shorter whiskers (i.e., lesser dispersion). Furthermore, we noticed that the baseline model found it challenging to achieve good detection performance for each of the four design smells. This clearly shows that the representation obtained by training the proxy task was good enough.

Generalizability evaluation. We conducted a cross-project evaluation with the main objective of determining the degree to which a fine-tuned model (on one dataset) may be utilized to identify design smells in other (unseen) datasets. We evaluated MoRT, feature-based, and baseline models across all projects. We followed this procedure: we initially fine-tune the model using the labeled dataset of a specific project (e.g., ArgoUML), then we subsequently evaluate that model performance on the remaining projects (i.e., Azureus, GanttProject, log4j, Lucene, Nutch, PMD, QuickUML, Xerces, Xerces2) without utilizing their own labeled datasets. We want to examine the generalizability of our approach, i.e., to which extent a model trained on a specific project can be used to detect bad smells in

new projects. Only the averages of ACC, P, R, F1, AUC, and MCC were considered; the detailed results of all projects are shown in the online appendix.

Table 45 shows the detection performance of cross-project evaluation for Blob, FD, SC, and SAK design smells. We notice that feature-based and MoRT have comparable performance; however, feature-based has better average performance on some smells Blob and SC. As anticipated, the feature-based approach demonstrates relatively better generalizability compared to MoRT in the cross-project evaluation. This result was expected since the feature-based approach only trains the last layers, allowing it to retain some level of transferable knowledge from the original pre-training because it uses its features without fine-tuning. On the other hand, MoRT fine-tunes all layers of the pre-trained model on the target dataset, which increases the detection performance of the target dataset, as discussed in the previous section, where MoRT performed better than the feature-based model. Compared to the baseline model, both feature-based and MoRT considerably outperformed the baseline model, where it has the lowest average performance.

RQ3 Answer. MoRT outperformed the feature-based model for all design smells, as evidenced by smaller boxplots and shorter whiskers, highlighting its superior detection performance. The baseline model struggled to achieve good detection performance across all four design smells, emphasizing the effectiveness of MoRT. In the cross-project evaluation, feature-based and MoRT significantly outperformed the baseline model, which had the lowest average performance, demonstrating the advantages of the self-supervised learning-based representations.

5.7 Discussion

In this section, we discuss the implications and limitations of the proposed method.

5.7.1 Pre-training tasks

A key factor in establishing accurate representations of structural and semantic properties is the choice of the proxy task. The results of this study indicate that the proxy task is well-defined since it achieved successful outcomes throughout both the pre-training and the fine-tuning phases. MoRT performed well on the proxy task; however, finding the best parameters involves a costly procedure, i.e., hyperparameter optimization. Therefore, it is essential to establish a model with good representation without performing a thorough search for hyperparameters. A new proxy task that predicts what UML snippets constitute (i.e., a class, method, or interface) or a mix of the two approaches would be one way to improve MoRT. This approach requires enormous volumes of unlabeled data to generate pseudo labels using the proxy task; however, one advantage is that it requires small labeled data for fine-tuning.

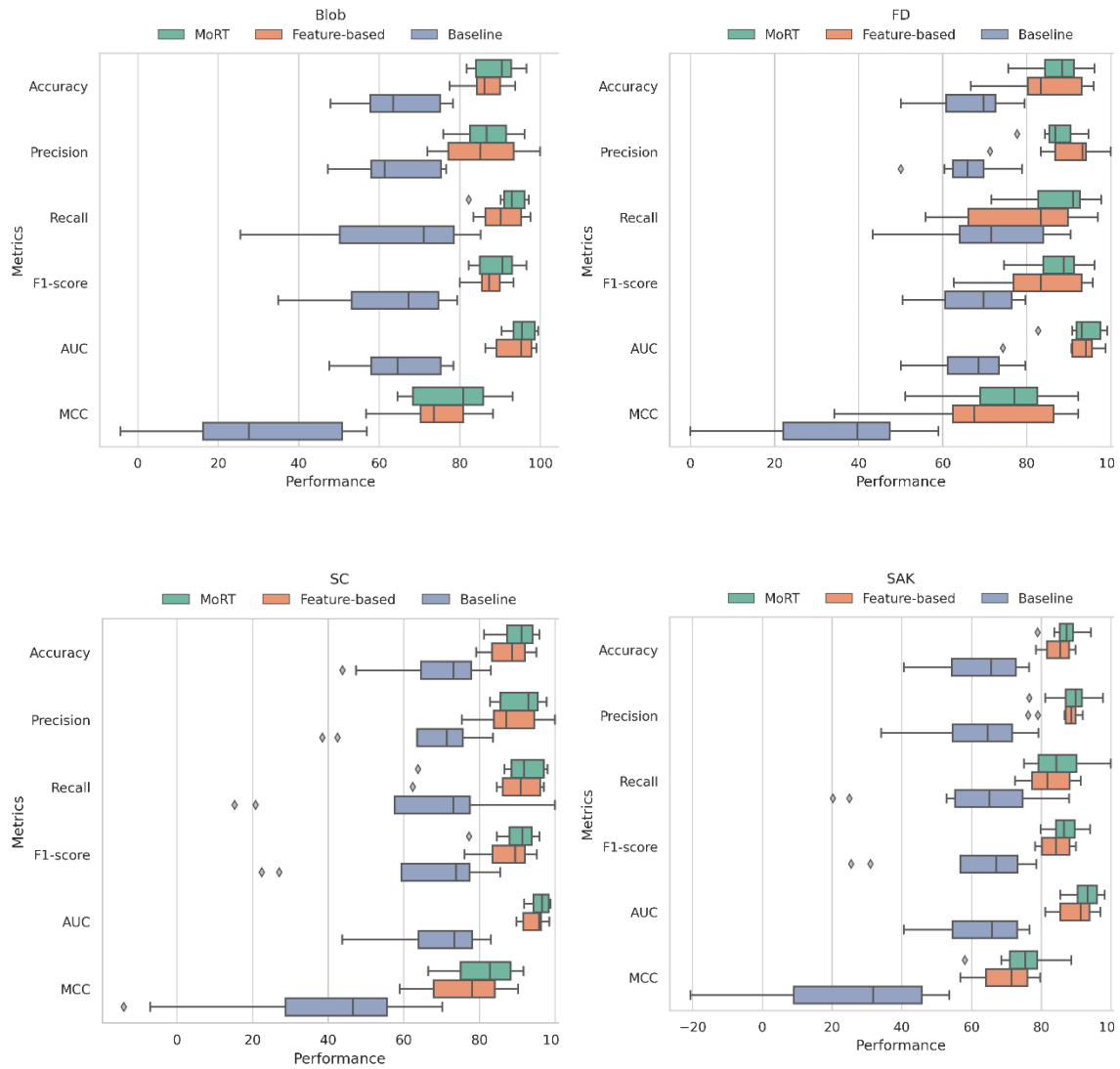


Figure 33 Detection performance boxplots of MoRT compared to baseline and feature-based

Table 45 Cross-project evaluation

Blob						
Model	ACC	P	R	F1	AUC	MCC
Baseline	56.54	56.08	45.79	48.49	56.74	13.01
Feature-based	73.80	86.01	58.14	64.23	86.38	51.41
MoRT	73.89	84.35	57.62	65.36	84.47	50.65
FD						
Model	ACC	P	R	F1	AUC	MCC
Baseline	53.55	52.99	42.11	45.81	53.62	7.02
Feature-based	68.85	86.97	44.33	55.73	73.20	43.39
MoRT	68.46	86.54	43.38	54.69	74.84	42.75
SC						
Model	ACC	P	R	F1	AUC	MCC
Baseline	58.13	59.17	45.18	50.00	58.22	16.57
Feature-based	76.12	87.88	62.41	69.41	86.57	55.90
MoRT	77.72	86.18	66.25	72.78	85.81	58.00
SAK						
Model	ACC	P	R	F1	AUC	MCC
Baseline	52.16	50.66	39.78	44.38	52.21	3.91
Feature-based	67.89	83.99	44.46	55.93	71.85	40.77
MoRT	67.94	82.18	46.01	56.73	71.52	40.28

5.7.2 Scalability and Adaptability

Our method is scalable, and it can handle large-scale datasets, making them suitable for analyzing complex and extensive UML class diagrams. This scalability allows for the detection of design smells in larger software systems, improving the overall quality and maintainability of the codebase. Moreover, our approach can learn from diverse datasets, enabling them to detect both known and previously unseen design smells. This adaptability is particularly valuable in an ever-evolving software landscape, where new design smells may emerge as programming practices and paradigms evolve. This makes our approach

general and can be used to detect other design smells, providing 1) the XMI representation and 2) labeled instances of the target design smells.

5.7.3 Advantages for researchers

The application of self-supervised learning to detect design smells in UML class diagrams opens new avenues for research and advancements in software engineering. Researchers can explore novel architectures, training methodologies, and feature representations to improve the accuracy and effectiveness of design smell detection. Also, researchers may investigate the application of this approach to model refactoring, where DL models are used to refactor UML diagrams.

5.7.4 Usage in industrial settings

The proposed method can be integrated into existing software development tools and workflows. For example, IDE plugins or code review tools can utilize these models to provide real-time feedback to developers, suggesting design smell resolutions and promoting cleaner code practices. Moreover, it can automate the detection process, reducing the need for manual inspection and analysis of UML class diagrams. This automation can significantly speed up the identification of design smells, enabling developers to focus on addressing the issues rather than spending time on manual detection.

5.8 Threats to validity

In conducting our investigation, we made a few assumptions that might have impacted the accuracy of our conclusions. We outline the potential risks and how we managed them in this section.

5.8.1 Conclusion validity

The selection of the validation method may affect the conclusion of this study. To minimize this threat, we chose the two most common approaches used for assessing DL-based models, which are the 80/20 and 70/30 split. Adopting one evaluation metric has a risk of bias that could produce false results. We employed several measures that are often used in the literature to mitigate the risk of bias, including accuracy, recall, f-score, precision, MCC, and AUC. We chose both threshold-dependent and threshold-independent assessment measures, such as AUC and loss, because some metrics, such as accuracy and recall, are threshold-dependent, and it might be difficult to determine the right threshold.

5.8.2 Construct validity

As for potential issues related to the relation between the theory and the observation, we identified potential problems in the adopted methodology. The implementation of the deep learning models is considered a threat; to mitigate this, we used TensorFlow's implementation, which is regarded as a trustworthy framework for creating DL models. Moreover, we used the XMI representation of the UML class diagram as independent variables; the transformation might be a threat since we cannot guarantee there are no structural changes; however, to minimize this threat, we used the Visual Paradigm tool, which has the highest success rate compared to other tools [193]. Also, the pre-training dataset was constructed from two different resources to provide diversity in our study. While we have ensured that these datasets do not directly overlap, as they were originally collected from two different resources, there is a possibility of unintentional overlap. We acknowledge that the potential for unintentional dataset overlap remains a concern in the context of construct validity. Our fine-tuning datasets were created using labeled datasets,

which pose further risk to their validity. We used labeled datasets that have been validated and used in previous studies to lessen this concern. Additionally, because manual mapping would be error-prone, we automated the mapping process to map the XMI representation of an entity to its true label to reduce the risk of mislabeling.

5.8.3 Internal validity

Internal threat is associated with the correctness of the experiment outcomes. One internal concern is that we concentrate on automatically detectable design smells that can be identified using structural information. This might restrict the approach's application because semantic-driven smells might not be detected by our method, which will be investigated in future studies.

5.8.4 External validity

This threat is concerned with the generalization of study conclusions. Although we chose four of the most popular smells, the results of our evaluation of the proposed approach on four design smells cannot be generalized to other smells. Finally, our findings cannot be generalized to all software systems, particularly industrial systems. However, to counteract this, we used a representative dataset that includes ten real Java projects and performed reverse engineering to extract the XMI representation. Additionally, we carried out several cross-project experiments to evaluate the generalizability of the suggested approach.

5.9 Chapter Summary

We introduced in this chapter a novel proxy task to generate pseudo labels, and a Transformer model was trained using XMI representation to recognize the structural and

semantic features. Although the suggested proxy task is straightforward, we demonstrated that it is effective in enforcing the trained model (MoRT) to acquire structural and semantic characteristics that are helpful for a variety of tasks, including the detection of Blob, FD, SC, and SAK design smells. By contrasting several DL model architectures and using hyperparameter optimization, we carefully constructed our model. We also assessed how time-consuming the suggested technique was. Additionally, we carefully evaluated our method by contrasting it with supervised and feature-based methods, and we achieved great detection performance for all design smells. By using cross-project assessment, we specifically assessed the generalizability of our model in comparison to supervised and feature-based techniques. We outperformed other techniques by a wide margin and achieved good detection scores in terms of accuracy, precision, recall, F1-score, AUC, and MCC.

CHAPTER 6

SmellyBot: Design and implementation of AI-powered

Software Bot for Code Smell Detection

Software bots can be leveraged to integrate DL-based code smell detection approaches into the development workflow. Bots were first introduced in 1950 by Alan Turing, who proposed that machines could think [64]. There is no universally agreed definition to describe bots; however, they have been defined by researchers and practitioners in terms of their particular context and usage [65]. Bots have been defined as software programs that automate repetitive tasks [26]. It has also been defined as a program that performs tasks, that can be executed by a human without any human interventions [66]. The recent advancement of AI and ML have facilitated the deployment of bots in various domains such as education, including students' engagement [67], self-learning [68], and tutoring [69]. They have also been used in other domains, such as in business and e-commerce [70] and in providing customer services [71].

In software engineering, bots are used to automate routine and repetitive tasks, allowing developers to focus on more complex tasks [34]. This enhances productivity and reduces the risk of human error [26] [28]. Bots can be integrated into collaborative platforms such as GitHub, GitLab, and Bitbucket to automate code testing, deployment, and monitoring processes. This ensures that code changes are continuously tested and deployed, leading to faster and more reliable software. Previous studies have developed bots to assist in code

reviews by automatically generating suggestions for fixing static analysis warnings [31], [194]–[196], fault fixing and localization [197], [198], and refactoring [28], [29]. This accelerates the review process and maintains code quality. Moreover, bots can assist in generating documentation [30], keeping them up to date, and ensuring consistency in code comments and documentation across the codebase, hence facilitating code maintainability.

In this study, we present the design and implementation of a software bot that allows seamless integration of DL models into developers’ workflows. We introduce SmellyBot, a code smell detection bot, based on pre-trained Transformer models trained on large-scale datasets [176]. It invokes the models that were deployed in a cloud platform whenever a developer submits a pull request or push submits a pull request or push, detects code smells, generates a report with all methods and classes that are smelly, and integrates the results directly by creating a new issue in the repository. Incorporating SmellyBot into software engineering processes can yield numerous advantages, including improved efficiency, code quality, security, and overall software reliability. Using SmellyBot reduces manual labor requirements, hence tend to minimize errors and optimize development and testing processes.

This work provides the following contributions:

- The design of a DL-based bot for code smell detection that can be integrated into developers’ workflow. The bot detects two class-level code smells (i.e., Data Class and God Class) and two method-level code smells (i.e., Feature Envy and Long Method).

- The implementation of the bot as a GitHub application facilitates the adaptation of the bot in a collaborative development platform; the implementation is publicly available on GitHub for replication.
- An empirical study to evaluate the bot in terms of its effectiveness and acceptance by developers. Furthermore, we provide insights and recommendations on the design and implementation of code smell detection bots.

The remainder of this chapter is organized as follows. In Section 6.1, we discuss related work to code smell detection and software bot. Section 6.2 describes the design and implementation of our bot and its main components. Section 6.3 presents the empirical study and the targeted research questions. The results are presented in Section 6.4 and the main implications are highlighted in Section 6.5. We discuss the threats of validity in Section 6.6 and summarize the chapter in Section 6.7.

6.1 Related Work

Recently, in the area of software engineering, the use of autonomous bots for automatic support of software activities has become more popular [26] [27]. For example, bots have been utilized for automatic code refactoring [28] [29], refactoring documentation [30], repair bugs [31], and code reviews [32] [33]. One main benefit of software bots is to reduce the workload with repetitive activities [34]. Wyrich and Bogner [28] have highlighted the value of refactoring bots and encouraged them to be adopted in practice as they are expected to increase efficiency and effectiveness. Moreover, software bots are also expected to enhance both the project quality and developer productivity, as discussed by Storey and Zagalsky's [26]. Nevertheless, there is a need for more studies that can confirm

these expectations. A recent study has shown that software bots are perceived as useful and generally accepted by developers [35].

Wyrich and Bogner [28] proposed a *Refactoring-Bot* prototype that supports a few refactoring operations, such as removing unused private fields and correcting the wrong order of modifiers. Their tool was integrated with a static analysis tool that detects a set of code smells, and then the *Refactoring-Bot* applies the refactoring techniques. Although the authors have thoroughly designed the prototype and discussed its importance, the bot has yet to be evaluated. Another refactoring bot named *RefBot* has been proposed by Alizadeh et al. [29]. *RefBot* constantly monitors the pull requests made in a version control system. Once it triggers a request, the bot analyzes the modifications made, recognizes refactoring opportunities, and then applies a series of refactoring operations to the modified code. They conducted an experiment to evaluate the effectiveness and performance of *RefBot*. Rebai et al. [30] introduced a refactoring documentation bot that suggests a message to document the refactoring operations applied in the pull request. The user can accept, reject, or modify the messages to be included in the pull request. Nakagawa et al. [199] proposed CLIONE bot, which tracks clone changes using NiCad and SourcererCC tools to detect code clones. The bot creates a pull request comment and then notifies the developer.

Uril et al. [31] proposed a testing bot where a *Repairnator* bot was developed to repair bugs. *Repairnator* constantly tracks the continuous integration test failures and then attempts to fix the bugs by giving recommendations from existing program repair libraries. The developer is responsible for manually verifying and fixing the bug according to the recommendation. Baudry et al. [200] proposed an ML-based approach to generate patches. They utilized SequenceR, which is a sequence-to-sequence model that takes buggy code

as input and outputs a patch suggestion. They developed a GitHub bot to repair bugs that can be fixed with a single-line code change. Balachandran proposed a *ReviewBot* [32] that analyzes the pull request using existing static analysis tools such as Checkstyle and FindBugs. Then, it publishes reviews automatically and suggests a human reviewer based on the change history of the code. *Fix-it* [33] is an extension of *ReviewBot*. It checks the auto-reviewer component of *ReviewBot* and then automatically fixes a variety of problems related to coding standard violations.

Some studies proposed bots for fixing static analysis warnings. Carvalho et al. [194] introduced C-3PR bot that suggests fixing the warning through pull requests. They integrated three linter violation repair tools ESLint, TSLint, and WalkMod Sonar plugin, to generate fix suggestions. Similarly, Serban et al. [195] developed a bot that generates suggestions for fixes and adds the suggestions to the pull request discussion. They used SonarQube to generate the suggestions. Etemadi et al. [196] proposed a bot that generates patch suggestions for static analysis warnings using SonarQube. They transformed the abstract syntax trees of the code utilizing metaprogramming to suggest patches. Then, the bot creates a new pull request with the changed code. Marginean et al. [198] introduced a bot for fault fixing using Getafix tool to generate patches. Their bot was implemented and used on Sapienz platform. Silva et al. [197] proposed a bot for fault localization using Spectrum-Based Fault Localization (SBFL) technique. The bot creates a new comment with suspicious lines on the pull request whenever there is a failure.

Besides refactoring and repairing bots, Abdellatif et al. [27] introduced a chatbot, MSRBot, that can answer developer queries by accessing information from software repositories. The knowledge base used to provide answers was generated from a compilation of

frequently asked questions concerning software repositories. They evaluated the bot on 12 participants, and 90% of participants reported the system's effectiveness.

Although software bots are promising, we can observe that most of the existing ML techniques have not been automated in a tool that can support the developers in detecting code smells. Therefore, there is a necessity for automating the detection process via the design and development of a software bot. One main benefit of software bots is to reduce the workload with repetitive activities [34]. They are expected to increase in efficiency and effectiveness in the practice [28]. Moreover, software bots are also expected to enhance both project quality and developer productivity [26]. Therefore, we aim to develop a bot that can automatically detect code smells.

Table 46 Related work summary on software bots

Study	Objective	Approach	Detection Tools	Development environment	Output
Wyrich and Bogner, 2019 [28]	Refactoring	Rule-based	SonarQube	Github	Creates a pull request in the original GitHub repository for each corrected finding.
Alizadeh et al., 2019 [29]	Refactoring	Search-based	Non-dominated sorting genetic algorithm (NSGA-II)	Github	Creates PRs containing refactoring suggestions.
Rebai et al., 2019 [30]	Refactoring documentation	Rule-based	RefactoringMiner	Github	Recommend a message to document the refactoring, their locations, and the quality improvement for that pull-request.
Nakagawa et al., 2020 [199]	Code clone detection	Text-based, token-based	NiCad, SourcererCC	Github	Makes a PR comment for each code fragment to be modified to notify the results to the developer.
Uril et al., 2018 [31]	Propose patches for bugs	NA	NPEFix, Nopol, and Astor	Github	When a patch is found, an email is sent to the Repairator patch analyst.

Baudry et al., 2021 [200]	Software Repair- repair a specific class of errors	ML-based	SequenceR: a neural network model.	Github	Submits a pull-request message describing the build failure and the patch,
Balachandran, 2013 [32] [33]	Review [32] and fix [33] coding standard violations	Rule-based	Checkstyle, PMD, FindBugs	Review Board	Publishes reviews, suggests a reviewer, and fixes a variety of problems
Carvalho et al., 2020 [194]	Fixing Static Analysis Violations	Event-based	ESLint, TSLint, and WalkMod Sonar plugin.	Github	Creates a PSs containing the corrections and issues description.
Serban et al., 2022 [195]	Fixes suggestions for Static Analysis Warnings	Rule-based	SonarQube	Github, Probot framework	Generates suggestions for fixes and integrates these suggestions directly into the pull request discussion.
Etemadi et al., 2022 [196]	Patch Suggestions for Static Analysis Violations	Rule-based	SonarQube	Github	Pull request with changes.
Marginean et al., 2019 [198]	Fault fixing	NA	Getafix	Sapienz	SAPFIX selects one of the published candidates and requests a reviewer for this candidate through the Phabricator code review system.
Silva et al., 2023 [197]	Fault Localization	Spectrum-Based Fault Localization (SBFL)	NA	Github	Creates a comment on the failure PR (a message reporting selected suspicious lines)
Abdellatif et al., 2020 [27]	Answer questions	ML-based (word2vec)	Google's Dialogflow engine	web-based	Answer
This work	Detect code smells	DL-based	CoRT	Github	Generates two reports, creates a new issue, and sends a notification email

6.2 Methodology

In this study, we propose an AI-powered bot for automating the process of code smell detection via the design and development of a software bot integrated within a software development workflow. The proposed method detects code smells using DL

implementation of SmellyBot. First, we give an overview of the architecture, and then we dive deep into the three main building blocks. Next, we present the main design decisions made during the development of SmellyBot and the implementations.

6.2.1 Overview

Figure 34 shows an overview of how SmellyBot operates. The main input of SmellyBot is the source code of the project repository. SmellyBot is triggered by push and pull requests made by developers. SmellyBot generates two outputs: (1) it automatically creates a new issue and posts it, and (2) it generates a log file with the detected smells. There are four types of smells that are supported by SmellyBot: (1) two class-level smells, including God Class and Data Class, and (2) two method-level smells, including Feature Envy and Long Method. The selection is based on the fact that these code smells are highly important (i.e., they have a major impact on software quality) and are more prevalent in software programs [117]. Also, our bot uses the CoRT [176] model, which supports only the four types of code smells. It consists of three building blocks: (1) GitHub actions and workflow, (2) DL model deployments, and (3) GitHub APP. Each component is discussed in the following subsections.

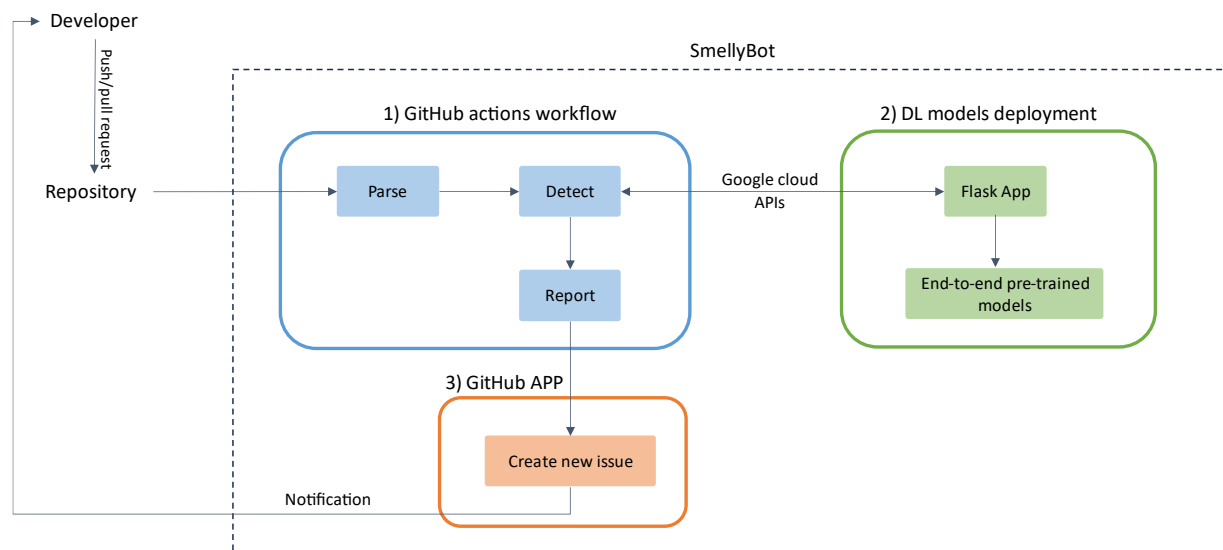


Figure 34 SmellyBot architecture

6.2.2 SmellyBot Building Blocks

SmellyBot operates on a GitHub repository, and the workflow is as follows. It continuously monitors all activities performed on the repository. Whenever the bot triggers a new pull request or push to the repository, it invokes the first block, i.e., the GitHub actions workflow, which uses the APIs of the DL models deployed on Google Cloud. Finally, it invokes the GitHub App responsible for preparing the two outputs: log files and new issues.

1 GitHub actions and workflow

We created three GitHub actions in the GitHub workflow: *Parsing*, *Detection*, and *Reporting*. In the Parsing component, we integrated source code parsing and preprocessing. This involves extracting relevant code from submitted pull requests or pushes, removing comments and white spaces, and ensuring consistent formatting for further analysis. The output of this entity is a list of all source code classes and methods, including the file path, the class/method name, and the body of the class/method.

The second component, the *Detection*, uses Google Cloud APIs to communicate with the deployed DL models to 1) detect the presence of Data Class or God Class smells for each class and 2) detect the presence of Feature Envy or Long Method smells for each method. The models return the detected code smells, and the results are processed further.

After code smell detection, the *Reporting* component generates two reports for each pull request: one for detected code smells at the class level and another for the method level. These reports provide detailed information about the identified issues, including the file path, the class/method name, and a binary value indicating whether this class/method is

smelly or not. The reports are automatically pushed to the repository as log files, making them easily accessible to developers.

2 DL Models deployment

To create models for code smell detection, we employed CoRT [176], which is a Transformer model trained on a large codebase. First, a labeled dataset of source code samples containing instances of various code smells was collected. We preprocessed the dataset to remove irrelevant information, such as comments and white spaces. We then fine-tuned CoRT for each code smell and created end-to-end models where each model takes a class or method of any length, performs vectorization and tokenization, and then returns a prediction. Then, we deployed the four models, i.e., one model for each code smell under investigation (i.e., Data Class, God Class, Feature Envy, and Long Method).

We developed a Flask application for interacting with the DL models. The application allows the bot to submit an entire file of code snippets for analysis and detection. The application loads the end-to-end model, reads the submitted file, and returns a JSON file that contains a list of 0/1 values for all code snippets (i.e., the prediction results). The Flask application serves as a gateway to our code smell detection models. Each code smell has a separate Flask application. To ensure the seamless deployment and scalability of Smellybot, we containerized the Flask application using Docker. This containerization simplifies deployment and facilitates compatibility across various cloud platforms. Finally, the containerized Flask application was deployed on the Google Cloud platform to make it accessible by SmellyBot via the Internet. Each one of the deployed models has an API that is used in the Detection component of the first block.

3 **GitHub App**

The last building block of SmellyBot creates a new issue in the repository when code smells are detected. This issue contains statistics about smelly and non-smelly classes and methods for each code smell type. Additionally, it includes a message that indicates the generated report files.

6.2.3 Design Decisions

In the development of SmellyBot, several key design decisions were made to ensure its effectiveness and flexibility. Each of these decisions was guided by a specific reason, resulting in a bot that is scalable, extensible, self-adaptive, and easily reachable by software developers.

Scalability. SmellyBot was designed to enable complicated and time-consuming code analysis. To achieve this, we opted for the Javalang library, which provides robust and efficient parsing capabilities for Java source code. Additionally, we implemented a data chunking mechanism to prevent API requests from resetting the connection during long and resource-intensive prediction tasks, especially when dealing with large volumes of data. This design decision ensures that SmellyBot can handle the parsing and detection of extensive codebases efficiently.

Extensibility. SmellyBot supports Java by default but is designed to be easily extensible to support other programming languages. To facilitate this, we divided the bot's components into modular units. The parsing component, for instance, can be extended to parse code in different languages. Furthermore, the detection component allows for the addition of new

features, and the reporting component allows the customization of the output format, making it adaptable to the evolving needs of software development teams.

Self-Adaptation. SmellyBot empowers its users by allowing them to adapt and fine-tune the DL models to their specific needs by deploying their model in any cloud platform and then updating the APIs. Users can adjust the models based on their own data, improving the bot's accuracy for their particular projects. Moreover, they can add new code smell detection capabilities by changing or adding new APIs. This level of self-adaptation ensures that SmellyBot remains a flexible and responsive tool for developers across various domains and projects.

Reachability. SmellyBot is designed to be integrated with the GitHub ecosystem, which is the largest open-source code collaborative platform. By seamlessly integrating into the GitHub environment, SmellyBot can reach a substantial community of developers, facilitating code quality assessment and ensuring that its capabilities are accessible to many users. This integration enhances the tool's reach and relevance within the software development community.

6.2.4 Implementation

SmellyBot is implemented in Python programming language. It uses the Tensorflow library to load the model and make predictions and the Javalang library to parse the source code. Moreover, the actions workflow is written in a Yaml file, which runs the components in a pre-defined sequence. The source code of SmellyBot and all pertinent experimental data

are openly accessible to support future research in this domain. You can access the repository online ¹⁵.

6.3 Evaluation

In this section, we present a comprehensive evaluation of SmellyBot, a code smell detection bot designed to enhance software development practices. In this section, we discuss the research questions (RQs), study subjects, and methods.

6.3.1 Research Questions

The main goal of our evaluation is to assess SmellyBot's usefulness, gather developer feedback, and measure its efficiency and accuracy in real-world software projects. We formulated the following RQs:

- RQ1. What is the developer's perception of using SmellyBot in their software development projects?
- RQ2. What are the developer recommendations and expectations for the design and functionality of an ideal code smell detection bot?
- RQ3. How efficient is SmellyBot in code smell detection across software projects of different sizes?
- RQ4. How accurate is SmellyBot in detecting code smells with real-world software projects?

¹⁵ <https://github.com/amalazba/smellybot/>

6.3.2 Methods

For RQ1 and RQ2, we opted for survey and open-ended questions to gain a deep understanding of developer perceptions, uncovering valuable insights into the strengths and weaknesses of SmellyBot. This allowed us to collect both qualitative and quantitative data, aiding in a comprehensive evaluation of the bot's effectiveness in real-world software development scenarios.

RQ1 (Developer Perception). We employed user testing as a means to facilitate open discussions about SmellyBot. In the preparation session, participants were provided with a concise overview of the study's goals and objectives. An introductory session was conducted to ensure that participants have a basic understanding of the concept of code smell detection. Participants were briefed on what code smells are, why they are important, and how they can impact software quality. This introduction aimed to establish a common understanding of the core concept being evaluated in the study. During the session, participants had the opportunity to try the bot by being directed to perform some tasks and express their thoughts and opinions on the bot. In the post-session, participants filled out a modified Technology Acceptance Model (mTAM) survey [201], which assessed their perception of SmellyBot's usefulness and ease of use.

RQ2 (Recommendations and Expectations). To capture developer recommendations and expectations, we utilized open-ended questions. Participants were encouraged to provide detailed feedback on their experience with SmellyBot, emphasizing features they liked and features they did not like, recommendations for new features, suggestions for improvements, and identification of challenges or difficulties encountered, along with proposed solutions.

For RQ3 and RQ4, we evaluated SmellyBot’s ability to operate efficiently and accurately in real-world software development scenarios. By encompassing various project sizes and comparing results with ground truth labels, we aimed to validate SmellyBot’s practical utility in the field of code smell detection. Details are below.

RQ3 (Efficiency). To assess the efficiency of SmellyBot in code smell detection across different project sizes, we conducted evaluations on real-world software projects. These projects encompassed a range of sizes, including small, medium, and large projects. This approach ensured that SmellyBot’s performance was tested under diverse scenarios, helping to understand its capabilities and limitations.

RQ4 (Accuracy). To evaluate the accuracy of SmellyBot in detecting code smells, we compared the bot’s predicted results with ground truth labels from real software projects. This comparison allowed us to quantify the accuracy (Acc), precision (P), recall (R), F1-score (F1), and Matthews Correlation Coefficient (MCC) of SmellyBot’s code smell detection capabilities.

6.3.3 Study Subjects

For RQ1 and RQ2, we engaged developers from diverse backgrounds, varying experience levels, and who worked on different types of projects. Participants were recruited through a recruitment form distributed among graduate and undergraduate students. Our study involved a total of 43 participants who were asked to share their screens while using the bot via Zoom or Microsoft Teams. Before the evaluation session, we collected background information, including their professional backgrounds, experience in software development using GitHub, and experience in code smell detection tools. Moreover, in

adherence with the ethical research practices, we obtained explicit consent from each participant before the evaluation sessions. Participants were provided with information about the study’s purpose, the data that would be collected, and how it would be used. They were given the opportunity to ask questions and provide their consent to participate voluntarily. Table 47 shows the background information of the recruited participants. The most common experience range among the respondents is “3-5 years”. This indicates that a significant proportion of the respondents are relatively experienced in software development. Most respondents (33 out of 43) have used GitHub in the past. Moreover, most respondents (32 out of 43) have not used other code smell detection tools or services in the past. Respondents who have utilized code smell detection tools listed the following tools ESLint (3), JDeodorant (3), Pylint (2), SonarCloud (2), iPlasma (1), inFusion (1), DÉCOR (1), PMD (1), and SonarLint (1).

For RQ3 and RQ4, we conducted our evaluations on a set of six open-source software systems. The selected projects, along with their version, and project size metrics number of classes (NOC), number of methods (NOM), and lines of code (LOC), are shown in Table 48.

Table 47 Participants background information

Background information	Distribution
Respondents' status	Student: 60.47% (Count: 26)
	Both: 32.56% (Count: 14)
	Employee: 6.98% (Count: 3)
Software development experience	3-5 years: 34.88% (Count: 15)
	1-2 years: 27.91% (Count: 12)
	6-10 years: 13.95% (Count: 6)
	More than 10 years: 11.63% (Count: 5)
	Less than 1 year: 11.63% (Count: 5)
Using bad smell detection tools	No: 81.4% (Count: 35)
	Yes: 18.6% (Count: 8)

Using GitHub in Project Development	Yes: 69.77% (Count: 30) No: 30.23% (Count: 13)
Familiarity with programming languages	Java: 27.21% (Count: 41) Python: 25.17% (Count: 37) C++: 16.33% (Count: 24) JavaScript: 15.65% (Count: 23) C#: 9.52% (Count: 14) VB: 0.68% (Count: 1) Php: 0.68% (Count: 1) Html: 0.68% (Count: 1) Rust: 0.68% (Count: 1) Dart: 0.68% (Count: 1) C: 0.68% (Count: 1) Swift: 0.68% (Count: 1) Flutter: 0.68% (Count: 1)
Types of software development projects worked on	Web Applications: 44.19% (Count: 19) Mobile Applications: 27.91% (Count: 12) Desktop Applications: 20.93% (Count: 9) Embedded Systems: 6.98% (Count: 3)

Table 48 Selected projects for evaluating SmellyBot

Name	Version	NOC	NOM	LOC
Cxf	2.7.0	5251	35479	468928
Nifi	1.9.2	5105	37262	557625
Calcite	1.21.0	2762	23918	321172
Activemq	5.8.0	3534	28867	295784
Karaf	4.2.7	1575	9075	122109
Olingo-odata	4.3.0	1751	12449	156166

6.4 Results and Discussion

In this section, we present the results and discuss our research questions. First, we will discuss the RQ1 and RQ2 in section 6.4.1 related to the developer perception evaluation. Then, we discuss the results of RQ3 and RQ4 related to the evaluation of real-world projects in section 6.4.2.

6.4.1 Developers' perception, recommendations, and expectations of using SmellyBot

We evaluated the developers' perception of SmellyBot usefulness and ease of use using mTAM. Table 49 presents the distribution of responses from participants on the mTAM survey, utilizing a seven-point Likert scale where 1 indicates strongly agree and 7 indicates strongly disagree. For perceived usefulness, most participants think that SmellyBot would accelerate tasks, enhance job performance, and increase productivity, with mean scores ranging from 1.74 to 2.19, suggesting a tendency towards "Agree" on the scale. The standard deviations were relatively low, suggesting a consensus among the participants.

For perceived ease of use, the majority of participants found SmellyBot straightforward to learn and operate, with clear and understandable interactions. The mean scores for these items were even lower, ranging from 1.42 to 1.70, showing a stronger agreement. Again, the standard deviations were low, indicating consensus in the perception of SmellyBot's ease of use.

Table 49 Modified Technology Acceptance Model (mTAM) scores

Statement	Strongly Agree (1)	Agree (2)	Slightly Agree (3)	Neutral (4)	Slightly Disagree (5)	Disagree (6)	Strongly Disagree (7)	Mean	Media	SD
Perceived usefulness (PU)										
1. Using SmellyBot in my job would enable me to accomplish tasks more quickly.	20	10	7	2	3	0	1	2.12	2	1.43
2. Using SmellyBot would improve my job performance.	24	10	5	0	3	0	1	1.88	1	1.38
3. Using SmellyBot in my job would increase my productivity.	18	14	3	4	2	1	1	2.19	2	1.50
4. Using SmellyBot would enhance my effectiveness on the job.	20	16	3	1	2	0	1	1.91	2	1.29

5. Using SmellyBot would make it easier to do my job.	22	9	7	4	0	0	1	1.95	1	1.29
6. I would find SmellyBot useful in my job.	25	11	4	1	1	0	1	1.74	1	1.24
Perceived ease of use (PEU)										
7. Learning to operate SmellyBot was easy for me.	35	6	0	0	0	0	2	1.42	1	1.30
8. I found it easy to get SmellyBot to do what I want it to do.	28	10	2	0	1	0	2	1.70	1	1.42
9. My interaction with SmellyBot has been clear and understandable.	34	5	1	0	1	0	2	1.53	1	1.42
10. I found SmellyBot to be flexible to interact with.	31	6	3	1	0	0	2	1.63	1	1.40
11. It was easy for me to become skillful at using SmellyBot.	30	8	3	0	0	0	2	1.60	1	1.35
12. I found SmellyBot easy to use.	37	2	2	0	0	0	2	1.42	1	1.33

We asked the participants a set of open-ended questions to understand if the bot met their expectations and to explore future recommendations. We asked them three main questions regarding 1) features of SmellyBot that they like/dislike the most, 2) challenges or difficulties they encountered while using the SmellyBot, and 3) new features they recommend adding. We analyzed the responses and identified specific items mentioned by the developers, reflecting their experiences and perspectives on the tool. This step was important in understanding the diverse aspects of SmellyBot that garnered developers' attention. Then, we categorized the items into five distinct categories: core functionalities, user experience and interface, notification and documentation, installation and performance, advanced features, and collaborative tools. Finally, each participant's responses were mapped to these identified items. This mapping was used to calculate the total number of developers who mentioned each specific item. By calculating the total

number of developers who mentioned each item, a prioritized list of improvements and features emerged based on actual user demand and preference.

Table 50 shows a list of the features liked by developers and are anticipated to be integral components of forthcoming software bots, a list of recommendations, and the challenges they have faced to be avoided in the design and development of future bots, and N, which represents the total number of developers who have identified or emphasized each listed item. Based on the feedback we have gathered, here are some of the main features appreciated by developers.

Core Functionalities: many participants appreciated the automation in detecting code smells, where the bot automatically checks and reports code smells after each commit. **User Experience and Interface:** the clarity of reports was highlighted; they liked the detailed reports at both class and method levels where they can see specifically which method or class suffers from code smells along with the full file path. In the Notification and Documentation category, two developers highlighted the importance of email notifications, with one noting that they are particularly crucial for large projects that require continuous monitoring. Moreover, one participant pointed out that the documentation of how to use the tool is comprehensive, easy to understand, and includes step-by-step guides. **Installation and Performance:** the integration with GitHub and the ability to work directly within this environment was a significant positive. One participant mentioned the simplicity of the SmellyBot installation process. For Detection Speed, four participants complimented the tool's fast processing and detection, while two mentioned that it is time-consuming and further optimization of the tool is needed. It is worth mentioning that 41 developers did not specify any specific features they disliked, and 38 developers did not

encounter any difficulties while using the bot. However, four participants found the abbreviations used in the generated report challenging to understand, where we used the first two letters for each code smell. Therefore, we suggest avoiding abbreviations in report headers to ensure better clarity and understanding. Additionally, one participant suggested adding a short video tutorial to demonstrate the steps for integrating the tool, which could help new users get started with SmellyBot more quickly. Furthermore, developers have recommended the following features.

Core Functional Improvements. Expand the bots's capabilities to support languages like Python, C++, etc., broadening its user base. Enhance the bot to cover a broader range of code smells, ensuring a more comprehensive analysis. Add functionality to detect smells at the package level, offering a more holistic view of the project's quality. Include specific line numbers in the reports to help developers quickly locate and address issues. Provide metrics calculations like coupling and cohesion to give developers a deeper understanding of their code's structural quality. Describe the detected smells with suggestions for fixing them. In addition, develop features for automatic or semi-automatic refactoring, then push the changes to a new branch, saving time and effort for developers.

User Experience and Interface Enhancements. Some participants suggested presenting the results in graphs or charts because visual representations can make it easier for users to understand and analyze the data. Include severity levels of the detected code smells to help prioritize which issues to address first. In the created issues, some participants suggested directly highlighting the lines of code that are problematic, improving the ease of identification. Allow the tool to function within popular code editors or during the coding

process, similar to a real-time linting tool. Enhance navigation by making class or method names in the report clickable, directing users to the exact location in the code.

Notification and Documentation Enhancements. Incorporate pop-up notifications for immediate alerts when a code smell is detected because some developers might miss the email notification while working on the project.

Advanced Features and Collaborative Tools. Include features for task assignment within teams, i.e., assign others to work on specific code smell to enhance collaborative efforts in code quality management. Implement a feature to track code smells over multiple versions, assigning higher priority to recurring issues.

Table 50 Expected features and recommendations

	Category	Item	N
Features	Core Functionalities	+ Automatic detection of code smells	26
	User Experience and Interface	+ Reporting feature and format	20
	Notification and Documentation	+ Email notification	2
		+ Clear documentation	1
	Installation and Performance	+ Integration with GitHub	17
		+ Detection speed	4
		+ Installation process	1
		- Detection speed	2
	-	- None	41
Challenges	Notification and Documentation Enhancements	The abbreviations used in the generated report	4
		A short video tutorial	1
	-	None	38
Recommendations	Core Functional Improvements	Support for other programming languages	8
		Suggestions for fixing	8
		Detect more code smells	6
		Calculate software metrics	5
		Specifying the line number of methods and classes	5
		Refactor the smells automatically	1
		Detecting package-level smells	1
		Present the results in graphs and charts	7
	User Experience and Interface Enhancements	Indicating the severity level of code smells	3
		Highlight the line of code	2

Notification and Documentation Enhancements	Integration with code editors	2
	Making class or method names clickable	2
	A pop-up message notification	1
	Assign others to work on specific code smell	1
Advanced Features and Collaborative Tools	Give high priority to code smells that occurred in multiple versions	1

6.4.2 The efficiency and accuracy of SmellyBot in code smell detection

In RQ3, we evaluated the total time it takes the bot to process each project and generate the detection report. The scatter plot in Figure 35 shows the relationship between the project size, represented by the number of lines of code (LOC), and the processing time in seconds. The processing time includes parsing the source code, extracting all classes and methods, detecting code smells, and generating the report. Each point on the plot corresponds to one of the projects listed in Table 48. Larger projects with approximately 500K lines of code required over an hour to process. Smaller projects with roughly 150K lines of code were processed in about 20 minutes. Medium-sized projects, close to 300K lines of code, were completed in under an hour.

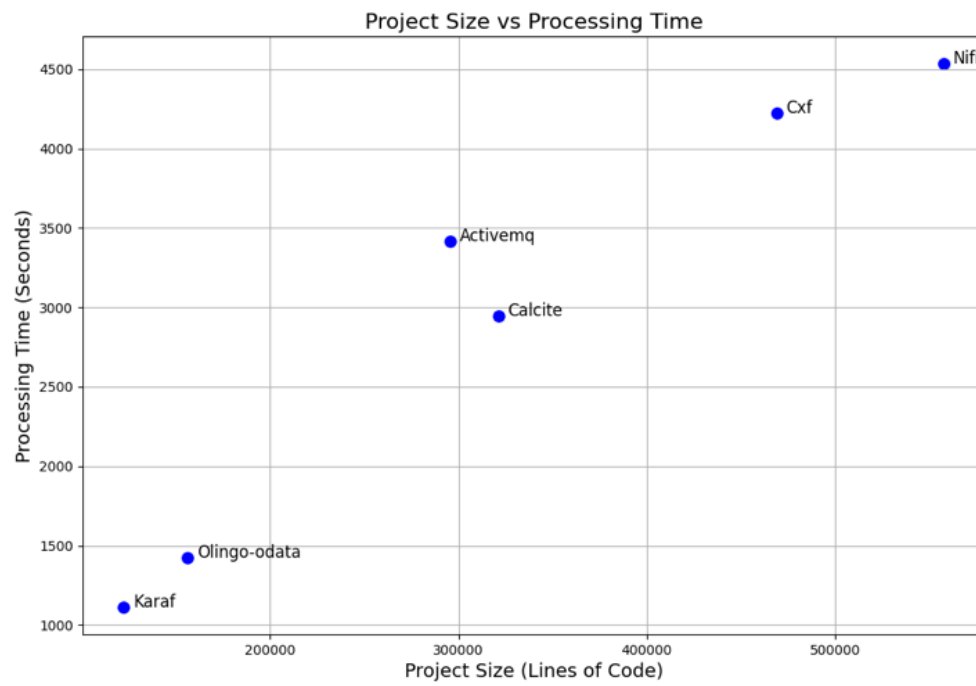


Figure 35 The processing time of SmellyBot

Table 51 shows the performance of CoRT model that was used in SmellyBot to detect code smells. For Data Class smells, the model exhibited a considerable degree of efficacy, achieving an average accuracy of 96.09%, with a high precision rate of 98.56%. However, the recall rate was slightly lower at 93.57%. This suggests a robust ability to accurately identify Data Class smells. In contrast, the detection of God Class smells presented a more challenging scenario. The model demonstrated a good level of accuracy (94.86%) and precision (96.40%), yet these figures were the lowest among the four code smells. The recall rate (93.38%) indicated a modest ability to identify all relevant instances. These metrics, combined with an F1-score of 94.83%, an AUC of 98.18%, and an MCC of 89.84%, suggest that while the model is effective in detecting God Class smells, its performance is less optimal compared to the detection of other code smells.

Conversely, the performance of the model was particularly noteworthy in the detection of Feature Envy and Long Method smells. For Feature Envy, the model achieved an accuracy of 97.69% and a remarkable recall rate of 98.06%, indicating a highly effective identification of almost all instances. The precision was also high at 97.39%. The Long Method smell detection demonstrated the model's peak performance, with the highest accuracy (98.23%) and precision (98.76%) among all smells. The recall rate was also excellent at 97.69%. Both code smells exhibited outstanding F1-scores (97.70% for Feature Envy and 98.21% for Long Method), AUC (99.36% for Feature Envy and 99.37% for Long Method), and MCC (95.42% for Feature Envy and 96.48% for Long Method), indicating a superior balance between precision and recall and a remarkable ability to differentiate between smelly and non-smelly code.

In summary, the model used in SmellyBot demonstrated varying degrees of effectiveness across the four code smells. While it showed exceptional capability in identifying Long Method and Feature Envy smells, its performance in detecting Data Class and God Class smells, though still effective, was comparatively less optimal. The variations in the model effectiveness across different code smells can be attributed to the inherent complexities associated with each smell type and the characteristics of the datasets. Overall, the model's high precision across all smells is particularly noteworthy, suggesting a low rate of false positives, which is crucial for its practical application in software development and maintenance.

Table 51 Detection performance of SmellyBot [176]

Project	Data class						God class					
	Acc	P	R	F1	AUC	MCC	Acc	P	R	F1	AUC	MCC
Activemq	98.4	99.7	97.08	98.37	99.5	96.83	96.97	99.52	94.4	96.89	98.95	94.07
Calcite	97.62	99.55	95.67	97.57	99.11	95.31	99.04	100	98.07	99.03	99.61	98.09
Cxf	96.2	96.63	95.74	96.18	98.55	92.41	95.5	97.61	93.29	95.4	98.53	91.1
Karaf	90.8	99.43	82.08	89.93	91.82	82.88	95.63	98.45	92.72	95.5	97.96	91.42
Nifi	97.78	98.25	97.28	97.76	99.16	95.55	93.83	96.65	90.81	93.64	97.69	87.82
Olingo-odata	95.72	97.77	93.58	95.63	98.11	91.53	88.2	86.17	91.01	88.52	96.35	76.53
Average	96.09	98.56	93.57	95.91	97.71	92.42	94.86	96.40	93.38	94.83	98.18	89.84
Project	Feature envy						Long method					
	Acc	P	R	F1	AUC	MCC	Acc	P	R	F1	AUC	MCC
Activemq	98.25	98.82	97.67	98.24	99.6	96.51	99.88	99.75	100	99.87	100	99.75
Calcite	97.14	98.53	95.71	97.1	99.44	94.32	98.92	99.13	98.71	98.92	99.73	97.85
Cxf	98.56	98.21	98.92	98.56	99.53	97.12	96.34	98.9	93.73	96.25	98.41	92.81
Nifi	97.77	97.58	97.98	97.78	99.6	95.55	99.27	99.27	99.27	99.27	99.85	98.53
Olingo-odata	96.71	93.83	100	96.82	98.65	93.62	96.74	96.74	96.74	96.74	98.84	93.48
Average	97.69	97.39	98.06	97.70	99.36	95.42	98.23	98.76	97.69	98.21	99.37	96.48

6.5 Implications

Automating code smell detection by implementing a software bot integrated with GitHub has several implications. In this section, we present the implications of our framework for researchers and practitioners.

6.5.1 Implications for researchers

Automation. The development of a code smell detection bot represents a significant advancement in automated software engineering tools. This provides researchers with an opportunity to study its effectiveness, compare it with existing tools, and explore areas for further improvement, such as enhancing its user experience design. Furthermore, this development serves as a foundational step for future DL-based approaches. It could facilitate the automation and deployment of various DL models and architectures, allowing for a more nuanced use and comparison.

Detection efficiency and accuracy. The effectiveness of the bot depends on its ability to accurately identify code smells without generating too many false positives. Investigating and improving DL models for detecting code smells with minimal false positives or negatives is essential, as inaccuracies can either lead to unnecessary revisions or overlook important issues. Moreover, the evaluation of these DL-based models should be performed in real-world environments to ensure their practical applicability and reliability.

Data Collection. The quality and quantity of labeled datasets are key aspects to improve the DL models' performance. The proposed bot can serve as a tool for collecting empirical data on code smells in various projects. The collected data can be leveraged to fine-tune the DL models, tailoring them to the specific needs of individual projects. It's important to

consider data sharing and privacy standards in this process, ensuring that the data collection is ethical and respects the confidentiality of the projects involved.

6.5.2 Implications for practitioners

Software development and quality. Automation tools like code smell detection bots can transform software development processes. These bots automate repetitive and time-consuming tasks, e.g., initial code reviews, saving valuable time and resources. The reduced need for manual code reviews significantly increases productivity, accelerating the overall software development cycle. Practitioners can use these bots to maintain higher-quality codebases, as they facilitate quick identification of problematic code. This proactive approach to code quality ensures better long-term maintainability of software products, ultimately leading to enhanced overall code quality of software systems.

Integration and automation. For a code smell detection tool to be effectively adopted, seamless integration with existing workflows like GitHub repositories is essential for user acceptance and effectiveness. Integration and automation of such tools can scale easily with project size, maintaining consistent code quality standards regardless of project complexity or the number of contributors. Furthermore, by automating the detection and review of code smells, organizations may significantly reduce the costs typically associated with manual code reviews and the long-term maintenance of their software products, thereby optimizing resource allocation.

Learning and skill development: Regular feedback from the automated code smell detection tool serves as a continuous learning mechanism for developers. This automation enables developers to receive immediate feedback on their code submissions, thereby accelerating their learning curve and adherence to best practices. This is particularly

valuable for junior developers, as the bot can be integrated into their training processes, serving as an educational tool, and helping them to learn about best practices and common pitfalls in coding.

6.6 Threats to Validity

In this section, we highlight the different threats related to our study and the steps we take to mitigate them.

6.6.1 Construct validity

A possible threat to construct validity is related to the measurement of developer perceptions. While we carefully designed surveys and questions, there is a possibility of measurement error in these assessments. The precision of developer feedback may influence the validity of our findings. To address this threat, we conducted pilot studies to refine our measurement tools. Additionally, we ensured that the questions posed to developers were clear and focused to minimize ambiguity. Moreover, to eliminate the mono-method bias, we did not rely on a single method for collecting the data; we incorporated the mTAM questionnaire and open-ended questions.

6.6.2 Internal validity

In the developer perception evaluation, the participants' background with code smell detection tools and concepts may introduce bias in their assessments. To mitigate this threat, we selected participants with varying levels of experience. Moreover, we ensured that they had a clear understanding of code smell detection concepts through preparation sessions. We also considered participant backgrounds and experiences when analyzing the

results. In project evaluation, the quality and quantity of the datasets used to train the DL models may impact the effectiveness of the bot. To mitigate this threat, we used labeled datasets that are publicly available and have been used in code smell detection research.

6.6.3 External validity

External validity is related to the generalizability of our findings to broader contexts. The participants in our study have used the bot for a simple project with specific tasks. Thus, the applicability of our findings cannot be generalized to real-world work environments. An extended evaluation of our bot in a real-world environment will enhance the generalizability of the bot. In our experiments, we utilized six open-source Java systems. The choice of selected open-source projects may limit the generalizability of our findings to other project types, programming languages, or industry domains. While we conducted our evaluations on diverse real-world projects, the generalizability of our findings is inherently limited. To address this threat, we chose projects from various domains and of different sizes. We encourage future research to replicate our study on a wider range of projects to enhance external validity.

6.6.4 Conclusion validity

The potential variability in developer opinions may lead to differing conclusions. To mitigate this threat, we employed both qualitative and quantitative methods, aiming to provide a comprehensive view. Additionally, we made efforts to ensure the consistency and reliability of data collection and analysis by recording and documenting the discussion during sessions.

6.7 Chapter Summary

In this chapter, we proposed the development and deployment of SmellyBot, an AI-powered bot for code smell detection. The combination of deep learning models, cloud deployment, GitHub integration, and automated reporting enhances the code quality assessment process for software development projects, promoting cleaner and more maintainable codebases. Our evaluation has demonstrated that Smellybot is very useful and easy to use from the developers' point of view. The lack of significant dislikes and challenges indicates a positive user experience and satisfaction with SmellyBot. Also, the evaluation has shown that Smellybot is efficient, however, it has low accuracy in detecting code smells.

CHAPTER 7

CONCLUSION AND FUTURE DIRECTIONS

The development of techniques that can automatically detect code smells in source code can assist developers in improving the source code, hence improving the maintainability and reducing the cost. In this research, we introduced a new approach for detecting bad smells using unlabeled data. The main advantage of our approach over previous machine learning and deep learning approaches is that it does not rely on manually annotated datasets.

We conducted an SLR and identified the current state-of-the-art bad smell types, DL techniques, features used as inputs, evaluation methods of the DL models, and datasets and systems used in the DL model training. We also explored the detection performance of DL models in relation to other factors (i.e., datasets, features, encoding techniques, preprocessing techniques, and bad smell types). A total of 67 studies were selected by following a systematic approach and analyzing the inclusion/exclusion criteria and the quality of the studies. Most primary studies detected bad smells at the code level, while a few detected it at the model level. CNN, ANN, and DNN are the most popular DL models used to build prediction classification models, while the LSTM, Attention model, CNN, and RAE are the most popular DL models used for representation learning. Most studies use supervised learning and utilize labeled datasets, and few studies made their datasets publicly available.

We introduced a code smell detection approach. A novel proxy task for self-supervised learning that trains a transformer model to recognize code representations was defined. We showed that it successfully forces the trained model to learn structural and semantic features that are useful for multiple downstream tasks, such as the detection of Data Class, God Class, Feature Envy, and Long Method code smells. We built our model by comparing multiple DL model architectures and applying hyperparameter optimization. We exhaustively evaluated the proposed method by applying it to two class-level and two method-level code smells. In addition, we evaluate the time complexity of the proposed approach. Moreover, we thoroughly assessed our method by comparing it with supervised and feature-based approaches and achieved high detection performance in all code smells. Specifically, we evaluated the generalizability of our model compared to supervised and feature-based methods by applying a cross-project evaluation. We obtained high detection scores in terms of accuracy, precision, recall, F1-score, AUC, and MCC, significantly surpassing other approaches.

We introduced a model smell detection approach where a Transformer model was trained using XMI representation to recognize the structural and semantic features. We demonstrated that it is effective in enforcing the trained model to acquire structural and semantic characteristics that are helpful for a variety of tasks, including the detection of Blob, FD, SC, and SAK design smells. We carefully evaluated our method by contrasting it with supervised and feature-based methods, and we achieved great detection performance for all design smells. By using cross-project assessment, we specifically assessed the generalizability of our model in comparison to supervised and feature-based

techniques. We outperformed other techniques by a wide margin and achieved good detection scores in terms of accuracy, precision, recall, F1-score, AUC, and MCC.

We automated the detection of bad smells by the development of Smellybot, an AI-powered bot for code smell detection. The combination of deep learning models, cloud deployment, GitHub integration, and automated reporting enhances the code quality assessment process for software development projects, promoting cleaner and more maintainable codebases. Our evaluation has demonstrated that Smellybot is very useful and easy to use from the developers' point of view. Also, the evaluation has shown that Smellybot is effective and efficient in detecting code smells.

7.1 Contributions

In this research, we introduced an approach for the purpose of detecting bad smells using unlabeled data. The contributions of this research are as follows:

Contribution #1: *Conduct a systematic literature review of the current state-of-the-art approaches in deep learning-based bad smell detection.*

Contribution #2: *Introduce an approach for detecting code smells using unlabeled data and utilizing a self-supervised learning paradigm.*

Contribution #3: *Introduce an approach for detecting model smells using unlabeled data and utilizing a self-supervised learning paradigm.*

Contribution #4: *Publicly available large-scale benchmarking code and model smells datasets.*

Contribution #5: *A comparative study between the proposed bad smell detection approach with supervised learning and feature-based approaches.*

Contribution #6: *A comprehensive evaluation to examine the generalizability of the proposed bad smell detection approach.*

Contribution #7: *The design and implementation of a software bot for automatic detection of bad smells.*

Contribution #8: *Provide a set of recommended features for future software bots.*

7.2 Challenges and Limitations

Learning representations from unlabeled data poses several challenges, and the performance of detection models relies on proxy tasks. In this research, we focused on utilizing a single source of features, namely source code and class diagrams. While combining multiple sources of features could potentially enhance performance, the extensive processing and training required limited our exploration to single features.

Another limitation pertains to the validation method employed. We used a train-test split approach instead of cross-validation, which is a more robust technique for validating machine learning (ML) and deep learning (DL) models. Unfortunately, due to hardware constraints, we were unable to employ cross-validation in our experiments. Future research with access to more robust hardware could benefit from implementing cross-validation.

Furthermore, our hyperparameter optimization was conducted within a selected search space that covered the most critical hyperparameters. Nevertheless, it is essential to acknowledge that other hyperparameters might impact the performance of detection

models. Future investigations may consider expanding the search space to explore the influence of additional hyperparameters on model performance.

Another limitation of this research is the feature set implemented in the software bot, which lacks some important features. This limitation could impact the detection performance of our approach. We observed a degradation in detection performance, partly attributed to the absence of fine-tuning within the bot. Incorporating a mechanism for collecting historical data and subsequently fine-tuning the model could potentially enhance the accuracy and effectiveness of our detection methodology.

While these challenges and limitations exist, they offer valuable insights into potential areas for improvement and future research directions. Addressing these limitations could contribute to the refinement and expansion of the proposed detection approach.

7.3 Future Directions

The performance of deep learning (DL) models in bad smell detection is influenced by several factors. Specifically, the detection performance of DL models is closely linked to the datasets used and the techniques employed in DL modeling. While many DL models have achieved comparable detection performance in terms of recall, most struggle to attain high precision. DL-based bad smell detection continues to pose challenges, necessitating ongoing efforts for performance enhancement.

Future research can explore the definition of novel proxy tasks aimed at improving the detection performance of bad smells. Another potential avenue for research involves the fusion of multiple features, such as source code and software metrics. This approach can

open up new research directions and enhance the effectiveness of bad smell detection. Extending the proposed approach to encompass the detection of code smells across multiple programming languages presents an intriguing research direction. This expansion can contribute to the versatility and applicability of the methodology.

To improve the detection of bad smells at the model level, future research may leverage a combination of UML models, including sequence diagrams and use-case diagrams, to train DL models. Furthermore, incorporating the detection of model smells into a software bot aids developers in implementing the suggested method to identify refactoring possibilities in practical contexts.

A promising direction for future work involves expanding SmellyBot's capabilities to support multiple programming languages, suggesting refactoring strategies, including a broader range of code smells, and providing indications of code smell severity levels. Additionally, enhancing the user experience by introducing features such as clickable class or method names and presenting results in visually appealing formats like graphs and charts may be explored in future studies. Finally, investigating methods to enhance detection performance by incorporating a feature that fine-tunes the model based on corrected code smells by developers is also a potential avenue for future research.

APPENDIX A: The selected primary studies

Table 52 List of primary studies

Study ID	Ref	Title
PS1	[202]	A deep learning Approach for a source code detection model using self-attention
PS2	[203]	A Deep Learning Based Approach to Detect Code Clones
PS3	[204]	A Deep Neural Network-Based Approach to Finding Similar Code Segments
PS4	[146]	A Hybrid Approach To Detect Code Smells using Deep Learning
PS5	[205]	A Hybrid Approach to Identify Code Smell Using Machine Learning Algorithms
PS6	[186]	A machine learning approach to software model refactoring
PS7	[206]	A machine learning based framework for code clone validation
PS8	[207]	A Novel Code Stylometry-based Code Clone Detection Strategy
PS9	[208]	A Novel Neural Source Code Representation Based on Abstract Syntax Tree
PS10	[209]	A Source Code Similarity Based on Siamese Neural Network
PS11	[210]	A universal cross language software similarity detector for open source software categorization
PS12	[211]	An Effective Semantic Code Clone Detection Framework using Pairwise Feature Fusion
PS13	[212]	AST-path Based Compare-Aggregate Network for Code Clone Detection
PS14	[213]	Capturing Source Code Semantics via Tree-based Convolution over API-enhanced AST
PS15	[214]	CCDLC Detection Framework-Combining Clustering with Deep Learning Classification for Semantic Clones
PS16	[215]	CCEyes An Effective Tool for Code Clone Detection on Large-Scale Open Source Repositories
PS17	[216]	CCLearner A Deep Learning-Based Clone Detection Approach
PS18	[174]	CLCDSA Cross Language Code Clone Detection using Syntactical Features and API Documentation
PS19	[217]	Clone detection in 5G-enabled social IoT system using graph semantics and deep learning model
PS20	[218]	Clone-Slicer- Detecting Domain Specific Binary Code Clones through Program Slicing
PS21	[219]	Code Clone Detection with Convolutional Neural Networks
PS22	[220]	Code Similarity Detection Based on Siamese Network
PS23	[142]	Code smell detection by deep direct-learning and transfer-learning
PS24	[221]	Cross-language clone detection by learning over abstract syntax trees
PS25	[222]	Deep hybrid features for code smells detection
PS26	[144]	Deep Learning Anti-Patterns from Code Metrics History
PS27	[141]	Deep Learning Based Code Smell Detection
PS28	[164]	Deep Learning Code Fragments for Code Clone Detection
PS29	[223]	Deep Learning Similarities from Different Representations of Source Code
PS30	[147]	Deep Representation Learning for Code Smells Detection using Variational Auto-Encoder
PS31	[143]	Deep semantic-Based Feature Envy Identification
PS32	[224]	DeepSim- Deep Learning Code Functional Similarity
PS33	[225]	Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree
PS34	[226]	Detecting Code Smells using Deep Learning

PS35	[227]	Disentangled Code Representation Learning for Multiple Programming Languages
PS36	[228]	Enhancing code clone detection using control flow graphs
PS37	[229]	Enhancing the Software Clone Detection in BigCloneBench- A Neural Network Approach
PS38	[149]	Exploiting Multi-aspect Interactions for God Class Detection with Dataset Fine-tuning
PS39	[230]	Fast Code Clone Detection Based on Weighted Recursive Autoencoders
PS40	[173]	FCCA Hybrid Code Representation for Functional Clone Detection Using Attention Networks
PS41	[150]	Feature Envy Detection based on Bi-LSTM with Self-Attention Mechanism
PS42	[231]	From Local to Global Semantic Clone Detection
PS43	[232]	funcGNN- A Graph Neural Network Approach to Program Similarity
PS44	[233]	Functional Code Clone Detection with Syntax and Semantics Fusion Learning
PS45	[234]	Go-Clone- Graph-Embedding Based Clone Detector for Golang
PS46	[148]	Improving the Identification of Code Smells by Combining Structural and Semantic Information
PS47	[235]	InferCode Self-Supervised Learning of Code Representations by Predicting Subtrees
PS48	[96]	Keep it simple Is deep learning good for linguistic smell detection-
PS49	[236]	Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection
PS50	[151]	Local and Global Feature Based Explainable Feature Envy Detection
PS51	[237]	Modeling Functional Similarity in Source Code with Graph-Based Siamese Networks
PS52	[238]	Modular Tree Network for Source Code Representation Learning
PS53	[152]	Multi-Granularity Code Smell Detection using Deep Learning Method based on Abstract Syntax Tree
PS54	[239]	Neural Detection of Semantic Code Clones via Tree-Based Convolution
PS55	[240]	Oreo- Detection of Clones in the Twilight Zone
PS56	[241]	Positive and Unlabeled Learning for Detecting Software Functional Clones with Adversarial Training.
PS57	[97]	Review Sharing via Deep Semi-Supervised Code Clone Detection
PS58	[242]	SCDetector- Software Functional Clone Detection Based on Semantic Tokens Analysis
PS59	[243]	Semantic code clone detection via event embedding tree and gat network
PS60	[244]	Sia-RAE A Siamese Network based on Recursive AutoEncoder for Effective Clone Detection
PS61	[245]	Siamese-Based BiLSTM Network for Scratch Source Code Similarity Measuring
PS62	[110]	Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code.
PS63	[246]	Teccd- A tree embedding approach for code clone detection
PS64	[247]	Temporal convolutional networks for just-in-time design smells prediction using fine-grained software metrics
PS65	[248]	Towards Automating Precision Studies of Clone Detectors
PS66	[249]	Two-Pass Technique for Clone Detection and Type Classification Using Tree-Based Convolution Neural Network
PS67	[250]	Code Clone Detection with Hierarchical Attentive Graph Embedding

Table 53 Primary studies venues and publication type

Venue	Publication Type	#PS
IEEE/ACM international conference on automated software engineering (ASE)	C	3
Journal of systems and software	J	3
IEEE access	J	3
International conference on software analysis evolution and reengineering (SANER)	C	3
IEEE/ACM international conference on software engineering (ICSE)	C	2
ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering (ESEC/FSE)	C	2
International joint conference on artificial intelligence (IJCAI)	C	2
International joint conference on neural networks (IJCNN)	C	2
IEEE international conference on software maintenance and evolution (ICSME)	C	2
IEEE international conference on information communication and software engineering (ICICSE)	C	2
IEEE/ACM international conference on mining software repositories (MSR)	C	2
IEEE Transactions on software engineering	J	2
International journal of open source software and processes (IJOSSP)	J	2
International wireless communications and mobile computing (IWCMC)	J	2
ACM sigsoft international symposium on software testing and analysis	C	2
Applied sciences	J	2
International journal of machine learning and cybernetics	J	1
International conference on program comprehension (ICPC)	C	1
International journal of software engineering and knowledge engineering	J	1
Journal of theoretical and applied information technology	J	1
International journal of electrical computer engineering	J	1
International journal of computers and applications	J	1
International conference on software quality reliability and security (QRS)	C	1
International conference on software maintenance and evolution (ICSME)	C	1
International conference on software engineering and knowledge engineering (SEKE)	C	1
International conference on software engineering (ICSE)	J	1
Neurocomputing	J	1
ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)	C	1
International conference on intelligent computing and human computer interaction (ICHCI)	C	1
Findings of the association for computational linguistics (ACL-IJCNLP)	C	1
ACM transactions on software engineering and methodology (TOSEM)	J	1
Annual computers software and applications conference (COMPSAC)	C	1
Asia pacific software engineering conference (APSEC)	C	1
Asia pacific symposium on internetwork	C	1
Bili im teknolojileri dergisi (journal of information technologies)	J	1
Complexity journal	J	1
Evaluation of Novel Approaches to Software Engineering (ENASE)	C	1
IEEE annual computers software and applications conference (COMPSAC)	C	1
International conference on dependable systems and their applications (DSA)	C	1
IEEE international conference on machine learning and applications	C	1

IEEE intl conf on parallel distributed processing with applications big data cloud computing sustainable computing communications social computing networking (ISPA BDCLOUD SOCIALCOM SUSTAINCOM)	C	1
IEEE region 10 conference (TENCON)	C	1
IEEE transactions on reliability	J	1
ACM international conference on computing frontiers	C	1
IEICE transactions on information and systems	J	1
International conference on neural information processing	C	1
Workshop on forming an ecosystem around software transformation	W	1
J: Journal, C: Conference, W: Workshop		

APPENDIX B: Types of bad smells detected using DL

Table 54 Types of bad smells detected using DL

Category	Bad smell	DL models *	#PS
Code smells	Code clone	Class: GGNN (1): [PS33], GMN (1): [PS33], RAE (1): [PS29] Code fragments: GAT (1): [PS59], VAE (1): [PS35], LSTM (1): [PS35], CNN (1): [PS22], ANN (1): [PS7], DNN (1): [PS7] File: TBCNN (3): [PS14, PS47, PS54], LSTM (2): [PS1, PS52], AST-LSTM (2): [PS62, PS56], Attention model (2): [PS1, PS47], RvNN (2): [PS28, PS52], T-LSTM (1): [PS52], Bi-LSTM (1): [PS1], Discriminative model (1): [PS1], Adversarial network (1): [PS56], RtNN (1): [PS28], BPTS (1): [PS28], AE (1): [PS28] Method: CNN (8): [PS2, PS15, PS57, PS37, PS10, PS12, PS21, PS3], DNN (8): [PS55, PS17, PS16, PS36, PS18, PS65, PS60, PS45], LSTM (6): [PS1, PS24, PS40, PS51, PS9, PS49], ANN (5): [PS24, PS32, PS44, PS40, PS66], RAE (6): [PS42, PS16, PS39, PS60, PS20, PS29], Attention model (6): [PS1, PS67, PS43, PS40, PS51, PS47], Bi-LSTM (3): [PS1, PS8, PS13], RvNN (3): [PS9, PS20, PS28], TBCNN (4): [PS66, PS47, PS14, PS54], AST-LSTM (3): [PS62, PS40, PS56], GCN (3): [PS40, PS67, PS42], GAT (1): [PS51], RtNN (1): [PS28], AE (2): [PS5, PS28], AST-RNN (1): [PS49], ASTNN (1): [PS9], BPTS (1): [PS28], Bi-GRU (1): [PS9], Compare-Aggregate Model (1): [PS13], DAE (1): [PS57], GRU (1): [PS58], GraphSAGE (1): [PS43], MLP (1): [PS51], NTN (1): [PS43], Adversarial network (1): [PS56], Discriminative model (1): [PS1], Skip-gram model (1): [PS63] Project: RtNN (1): [PS19], LSTM (1): [PS61], PV-DM (1): [PS11]	58
	Feature envy	Class: ASTNN (1): [PS53], Bi-GRU (1): [PS53], RvNN (1): [PS53], CNN (1): [PS23] RtNN (1): [PS23], AE (1): [PS23] Method: CNN (3): [PS50, PS31, PS27], LSTM (2): [PS50, PS3], ANN (2): [PS4, PS41], MLP (2): [PS50, PS31], AE (1): [PS4], VAE (2): [PS46, PS30], Bi-LSTM (1): [PS41], Attention model (1): [PS50]	9
	Long method	Method: VAE (2): [PS46, PS30], AE (2): [PS4, PS5], ANN (1): [PS4], DNN (1): [PS27]	5
	Large class	Class: DNN (1): [PS27], LSTM (1): [PS27], AE (1): [PS5]	2
	Data class	Class: AE (1): [PS4], ANN (1): [PS4]	1
	Misplaced class	Class: CNN (1): [PS27]	1
	Brain class	Class: CNN (1): [PS34]	1
	Brain method	Method: CNN (1): [PS34]	1
	Complex conditional	Method: AE (1): [PS23], CNN (1): [PS23], RtNN (1): [PS23]	1
	Complex method	Method: AE (1): [PS23], CNN (1): [PS23], RtNN (1): [PS23]	1
	Empty catch block	Method: ASTNN (1): [PS53], Bi-GRU (1): [PS53], RvNN (1): [PS53]	1
	Attribute name and type are opposite	Attribute: CNN (1): [PS48]	1
	Attribute signature and comment are opposite	Attribute: CNN (1): [PS48]	1
	Expecting but not getting a collection	Method: CNN (1): [PS48]	1

	Expecting but not getting a single instance	Method: CNN (1): [PS48]	1
	Get method does not return	Method: CNN (1): [PS48]	1
	Get method does not return corresponding attribute	Method: CNN (1): [PS48]	1
	Get more than an accessor	Method: CNN (1): [PS48]	1
	Is returns more than a Boolean	Method: CNN (1): [PS48]	1
	Method name and return type are opposite	Method: CNN (1): [PS48]	1
	Method signature and comment are opposite	Method: CNN (1): [PS48]	1
	Name suggests Boolean but type does not	Attribute: CNN (1): [PS48]	1
	Not answered question	Method: CNN (1): [PS48]	1
	Not implemented condition	Method: CNN (1): [PS48]	1
	Says many but contains one	Attribute: CNN (1): [PS48]	1
	Says one but contains many	Attribute: CNN (1): [PS48]	1
	Set method returns	Method: CNN (1): [PS48]	1
	Transform method does not return	Method: CNN (1): [PS48]	1
	Validation method does not confirm	Method: CNN (1): [PS48]	1
Design smells	God class	Class: CNN (3): [PS25, PS38, PS26], LSTM (2): [PS25, PS38], ANN (2): [PS4, PS38], AE (1): [PS4], GRU (1): [PS25], Attention model (1): [PS38]	4
	Blob	Class: VAE (2): [PS46, PS30]	2
	Deficient encapsulation	Class: ASTNN (1): [PS53], Bi-GRU (1): [PS53], RvNN (1): [PS53], TCN (1): [PS64]	2
	Insufficient modularization	Class: ASTNN (1): [PS53], Bi-GRU (1): [PS53], RvNN (1): [PS53], TCN (1): [PS64]	2
	Multifaceted abstraction	Class: TCN (1): [PS64], CNN (1): [PS23], AE (1): [PS23], Rtnn (1): [PS23]	2
	Broken modularization	Class: TCN (1): [PS64]	1
	Cyclically-dependent Modularization	Class: TCN (1): [PS64]	1
	Cyclic hierarchy	Class: TCN (1): [PS64]	1
	Deep hierarchy	Class: TCN (1): [PS64]	1
	Functional decomposition	Class: ANN (1): [PS6]	1
	Hub-like modularization	Class: TCN (1): [PS64]	1
	Missing hierarchy	Class: TCN (1): [PS64]	1
	Multipath hierarchy	Class: TCN (1): [PS64]	1
	Rebellious hierarchy	Class: TCN (1): [PS64]	1
	Unexploited encapsulation	Class: TCN (1): [PS64]	1
	Unutilized abstraction	Class: TCN (1): [PS64]	1
	Wide hierarchy	Class: TCN (1): [PS64]	1
	Broken hierarchy	Class: TCN (1): [PS64]	1
	Imperative abstraction	Class: TCN (1): [PS64]	1

	Unnecessary abstraction	Class: TCN (1): [PS64]	1
ANN: Artificial neural networks, AST-LSTM: AST-based long short-term memory, ASTNN: AST-based neural network, AST-RNN: AST-based recursive neural network, AE: Autoencoder, BPTS: Backpropagation through structure, Bi-GRU: Bidirectional gated recurrent unit, Bi-LSTM: Bidirectional long short-term memory, Bi LSTM: Bidirectional long-short term memory, CNN: Convolutional neural network, DNN: Deep neural networks, DAE: Denoise autoencoder, GGNN: Gated graph neural networks, GRU: Gated recurrent unit, GAT: Graph attention networks, GCN: Graph convolutional network, GMN: Graph matching networks, LSTM: Long short-term memory, MLP: Multilayer perceptron, NTN: Neural tensor network, PV-DM: Paragraph vector distributed memory, RtNN: Recurrent neural network, RAE: Recursive autoencoders, RvNN: Recursive neural network, TCN: Temporal convolutional networks, T-LSTM: Tree long short-term memory, TB-CNN: Tree-based convolutional neural network, VAE: Variational autoencoder			

**A PS may belong to one or more fields.*

APPENDIX C: Publicly available datasets for each bad smell

Table 55 Publicly available datasets

PS	Bad smells	Link
PS7	Code clone	https://github.com/pseudoPixels/CloneCognition
PS9	Code clone	https://github.com/zhangj1994/astnn
PS11	Code clone	https://github.com/Kawser-nerd/CroLSim
PS14	Code clone	https://github.com/milkfan/TBCAA
PS18	Code clone	https://github.com/Kawser-nerd/CLCDSA
PS23	CM, CC, FE, MA	Sharma, T., 2021. tushartushar/DeepLearningSmells: public release. Zenodo, https://doi.org/10.5281/zenodo.4571626 .
PS24	Code clone	https://www.csg.ci.i.u-tokyo.ac.jp/projects/clone/
PS26	God Class	https://github.com/antoineBarbez/CAME/
PS27	FE, LM, LC, MC	https://github.com/liuhuigmail/DeepSmellDetection
PS29	Code clone	https://sites.google.com/view/learningcodesimilarities
PS32	Code clone	https://github.com/parasol-asr/deepsim
PS39	Code clone	https://github.com/zyj183247166/Recursive_autoencoder_xiaojie
PS40	Code clone	https://github.com/preesee/CodeCloneDetection
PS43	Code clone	https://github.com/aravi11/funcGNN
PS44	Code clone	https://github.com/shiyy123/FCDetector
PS48	Linguistic smells	https://github.com/Smfakhoury/SANER-2018-KeepItSimple-
PS51	Code clone	https://drive.google.com/drive/folders/1x6ePcgJVtFVOycENEiYBpz5FYzsx2D-?usp=sharing
PS54	Code clone	http://github.com/yh1105/datasetforTBCCD
PS64	IA, MA, UA, UNA, DE, UE, BM, CDM, IM, HLM, BH, CH, DH, MH, MPH, RH, WH	https://bit.ly/3oltvpN
PS65	Code clone	https://github.com/Mondego/InspectorCloneArtifacts

Complex method (CM), Complex conditional (CC), Feature envy (FE), Long method (LM), Large class (LC), Misplaced class (MC), Imperative Abstraction (IA), Multifaceted Abstraction (MA), Unnecessary Abstraction (UA), Unutilized Abstraction (UNA), Deficient Encapsulation (DE), Unexploited Encapsulation (UE), Broken Modularization (BM), Cyclic D. Modularization (CDM), Insufficient Modularization (IM), Hub-like Modularization (HLM), Broken Hierarchy (BH), Cyclic Hierarchy (CH), Deep Hierarchy (DH), Missing Hierarchy (MH), Multipath Hierarchy (MPH), Rebellious Hierarchy (RH), Wide Hierarchy (WH)

APPENDIX D: Detailed projects' description

Table 56 Pre-training dataset description

#	Project	Version	NOC	NOM	LOC	Time (Mins)
1	Accumulo	1.7.3	3254	33810	419546	135.22
2	Activemq-parent	5.8.0	3534	28867	295784	105.85
3	Airavata	0.16	2221	33192	391359	149.03
4	Apache-ant	1.7.0	1267	11282	117076	37.53
5	Apache-archiva	1.4.M3	760	6955	92045	23.57
6	Apache-artemis	2.6.3	4255	39407	484235	169.38
7	Apache-atlas	0.8.2	923	8429	108950	41.92
8	Apache-calcite	1.21.0	2762	23918	321172	81.6
9	Apache-camel	1.6.0	2009	10507	99145	22.4
10	Apache-cassandra	0.7.4	636	6876	88612	23.45
11	Apache-continuum	1.4.0	607	6326	79262	18.62
12	Apache-crunch	0.9.0	568	3384	35261	15.9
13	Apache-curator	4.3.0	646	3667	57609	21.67
14	Apache-cxf	2.7.0	5251	35479	468928	105.78
15	Apache-directory-server	2.0.0.M24	1891	12017	224976	42.9
16	Apache-directory-studio	2.0.0.v20180908-M14	1850	12709	210981	64.82
17	Apache-drill	1.11.0	2942	31495	334522	65.37
18	Apache-hive	0.13.1	4697	50614	629986	156.55
19	Apache-hyracks	0.3.1	2277	12557	154261	45.03
20	Apache-ivy	2.0.0	488	5082	53540	24.87
21	Apache-karaf	4.2.7	1575	9075	122109	47.9
22	Apache-kylin	2.3.2	1477	12248	142468	49.5
23	Apache-manifoldcf	1.1	1040	7080	153040	47.28
24	Apache-maven	3.5.0.alpha	974	6529	79861	19.73
25	Apache-metron	0.5.0.rc2	1128	8296	103283	43.87
26	Apache-nutch	1.15	619	3433	57761	27.13
27	Apache-ofbiz	13.07.01	1539	13543	243216	128.77

28	Apache-openjpa	2.3.0	4722	45007	424623	95.68
29	Apache-openmeetings	4.0.2	604	4596	55224	30.53
30	Apache-opennlp	1.7.2	831	3580	59778	19.55
31	Apache-parquet	1.8.1	761	5237	58563	18.78
32	Apache-plc4x	0.6.0	1023	5219	61029	29
33	Apache-pulsar	2.4.1	2118	20641	287382	115.02
34	Apache-qpidd-jms	0.5.0	548	6164	64618	16.95
35	Apache-ranger	1.0.0	1003	10287	159818	54.2
36	Apache-reef	0.15.1	2346	6475	81187	23.02
37	Apache-samza	1.2.0	1305	6720	89538	38.28
38	Apache-storm	1.2.3	1940	19380	214258	58.43
39	Apache-tez	0.9.1	1328	10948	179571	62.25
40	Apache-tomcat	7.0.57	2225	21073	252668	75.77
41	Apache-wicket	6.23.0	3349	16419	210824	43.62
42	Avro	1.9.1	943	7032	77722	25.33
43	Beam	2.9.0	4330	32952	422866	152.53
44	Bookkeeper	4.6.0	793	5829	85650	35.9
45	Carbondata	2.0.0	1152	9822	134500	52.03
46	Cayenne	4.1.B2	3860	23673	265988	62.18
47	Clerezza	201508.1	55	328	5612	4.22
48	Cocoon	2.1.6	2771	17057	285918	63.35
49	Commons-beanutils	1.9.3	284	2702	33421	9.27
50	Commons-cli	1.3.1	48	499	6446	2.73
51	Commons-codec	1.11	122	1414	19900	7.85
52	Commons-collections4	4.3	626	6330	62869	14.85
53	Commons-compress	1.19	373	3551	46063	12.93
54	Commons-csv	1.7	31	550	6931	2.73
55	Commons-dbcp2	2.5.0	140	3467	30346	6.08
56	Commons-email	1.3.3	47	422	6478	2.5
57	Commons-io	2.5	238	2331	29546	7.58
58	Commons-lang3	3.3.2	361	5189	64300	16.03
59	Commons-math3	3.3	1646	12531	187519	62.37
60	Commons-net	3.3	254	1823	25787	7.42

61	Commons-validator	1.5.1	140	1128	15239	4.35
62	Commons-vfs	2.5.0	501	3276	33554	10.73
63	Deltaspikes-project	0.6	1288	4268	55122	12.85
64	Flink	1.3.2.rc3	5685	36290	530683	130.73
65	Fop	2.3	2341	17710	212088	56.07
66	Giraph	1.1.0	1001	6238	68757	17.37
67	Hadoop	2.2.0	5636	49887	651434	147.08
68	Hbase	2.1.10	4508	51266	694464	248.77
69	Helix	0.7.1	778	4792	78858	28.7
70	Isis	1.16.1.20180316-1549	3972	17522	177636	31.23
71	Jackrabbit	2.0.0	2571	20877	247728	71.68
72	Jackrabbit	0.8	1196	8914	113679	30.35
73	Jasml	0.1	46	216	5818	4.6
74	Jedit	4.2	603	4848	90531	29.38
75	Jena	3.14.0	6630	58330	553442	102.9
76	Jfreechart	1.0.13	982	10570	143110	28.88
77	Jgrapht	0.8.1	269	1235	17232	3.1
78	Johnzon	1.1.12	461	2876	31367	13.25
79	Kafka	2.4.0	2616	22721	298808	95.4
80	Kerby	1.1.1	926	5482	52749	10.2
81	Knox	1.3.0	1306	7883	105564	25.32
82	Logging-log4j1	1.2.0	417	3077	35262	10.38
83	Lucene	2.4.0	1046	7932	125946	35.92
84	Nifi	1.9.2	5105	37262	557625	187.23
85	Olingo-odata	4.3.0	1751	12449	156166	43.63
86	Openwebbeans	1.1.7	1145	4311	57545	14.18
87	Phoenix	4.7.0-HBase-0	1498	16362	228318	67.87
88	Poi	3.0.rc4	1410	11325	140394	31.4
89	Roller-weblogger	5.2.4	621	5008	53630	18.68
90	Shiro	1.4.1	721	3719	34656	7.92
91	Struts	2.1.6	1069	6566	62961	18.02
92	Surefire	2.18.1	942	4136	44290	9.55

93	Syncope	2.0.4	1934	10354	144861	71.37
94	Tajo	0.11.2	1935	16407	215049	95.98
95	Tapestry	5.1.0.0	1860	8938	93320	51.37
96	Tika	1.14	882	5476	84592	96.75
97	Tinkerpop	3.4.3	1920	13278	147074	265.83
98	Uima-ruta	2.8.0	1226	6333	98991	129.78
99	Velocity	1.6	385	2615	37007	10.8
100	Xalan	2.7.0	1004	9341	165720	35.3
101	Xerces2-j-trunk	2.0.0	883	9799	142696	42.23
102	Xmlsec	2.1.4	751	5117	79095	24.62
103	Zeppelin	0.7.3	620	6397	77699	25.6
Total			165K NOC	1.3M NOM	16M LOC	78h

Table 57 Fine-tuning datasets description

Project	Data class			God class			Feature envy			Long method		
	Smelly	Non-smelly	Total	Smelly	Non-smelly	Total	Smelly	Non-smelly	Total	Smelly	Non-smelly	Total
Accumulo	31	2219	2250	6	2244	2250	265	1204	1469	264	1094	1358
Activemq	84	2286	2370	168	2202	2370	33	1141	1174	23	1339	1362
Airavata	144	1214	1358	0	1358	1358	440	1072	1512	319	974	1293
Archiva	34	429	463	73	390	463	4	159	163	1	181	182
Atlas-sources	18	469	487	52	435	487	5	175	180	6	174	180
Avro	3	468	471	13	458	471	7	195	202	7	195	202
Beam	17	2096	2113	92	2021	2113	37	1265	1302	3	1568	1571
Bookkeeper	5	398	403	23	380	403	4	149	153	4	144	148
Calcite	45	1541	1586	29	1557	1586	26	699	725	12	772	784
Camel	23	894	917	121	798	919	10	444	454	2	496	498
Carbondata	43	572	615	81	534	615	9	222	231	5	252	257
Cassandra	6	383	389	3	386	389	8	227	235	2	212	214
Cayenne	110	2040	2150	178	1972	2150	10	753	763	8	845	853
Clerezza	0	25	25	2	23	25	1	12	13	1	5	6

Cocoon	89	1167	1256	90	1166	1256	12	702	714	12	619	631
Commons-beanutils	10	137	147	12	135	147	0	50	50	1	45	46
Commons-cli	0	33	33	4	29	33	0	8	8	0	8	8
Commons-codec	0	69	69	5	64	69	0	21	21	0	19	19
Commons-collections4	0	216	216	19	197	216	0	148	148	0	157	157
Commons-compress	7	176	183	25	158	183	3	55	58	0	65	65
Commons-csv	0	17	17	4	13	17	0	4	4	0	2	2
Commons-dbc2	7	85	92	14	78	92	0	37	37	0	35	35
Commons-email	1	19	20	5	15	20	2	5	7	0	8	8
Commons-io	0	111	111	8	103	111	2	51	53	0	53	53
Commons-lang3	2	185	187	21	166	187	1	45	46	0	38	38
Commons-math3	15	973	988	82	906	988	21	263	284	11	303	314
Commons-net	4	146	150	15	135	150	7	43	50	5	52	57
Commons-validator	1	85	86	17	69	86	1	25	26	0	29	29
Commons-vfs	7	230	237	23	214	237	4	90	94	2	105	107
Continuum	53	311	364	38	326	364	3	119	122	1	134	135
Crunch	3	257	260	21	239	260	3	141	144	1	176	177
Curator	2	210	212	39	173	212	0	82	82	0	87	87
Cxf	178	2191	2369	184	2185	2369	38	923	961	40	955	995
Directory-server	34	889	923	100	823	923	5	401	406	3	367	370
Directory-studio	49	936	985	138	847	985	15	242	257	7	268	275
Drill	53	1397	1450	5	1445	1450	25	862	887	14	831	845
Flink	5	166	171	12	159	171	1	109	110	1	118	119
Fop	85	1265	1350	49	1301	1350	19	547	566	13	599	612
Giraph	14	742	756	49	707	756	24	429	453	4	486	490

Hadoop	91	2949	3040	56	2984	3040	40	1100	1140	22	1207	1229
Hbase	75	2656	2731	9	2722	2731	99	1238	1337	102	1288	1390
Helix	26	399	425	37	388	425	8	166	174	8	166	174
Hive	163	2764	2927	4	2923	2927	217	1610	1827	236	1481	1717
Isis	23	1170	1193	137	1056	1193	11	598	609	4	643	647
Jackrabbit	73	1114	1187	124	1063	1187	11	613	624	6	663	669
Jackrabbit	23	613	636	54	582	636	5	237	242	2	270	272
Jasml	-	-	-	8	5	13	22	16	38	36	39	75
Jena	42	3299	3341	148	3193	3341	45	1399	1444	21	1536	1557
Johnzon	10	160	170	12	158	170	0	85	85	1	77	78
Kafka	3	1434	1437	79	1358	1437	19	405	424	9	479	488
Karaf	34	703	737	53	684	737	10	333	343	9	302	311
Kerby	56	414	470	63	407	470	6	191	197	4	209	213
Knox	13	613	626	43	583	626	6	230	236	5	233	238
Kylin	34	761	795	34	761	795	11	316	327	3	332	335
Lucene	7	614	621	43	579	622	9	230	239	5	233	238
Manifoldcf	52	934	986	8	978	986	19	528	547	27	503	530
Maven	36	367	403	34	369	403	2	135	137	3	142	145
Metron	1	50	51	8	43	51	1	26	27	0	27	27
Nifi	164	2695	2859	212	2647	2859	36	821	857	15	909	924
Nutch	5	307	312	28	284	312	10	103	113	11	75	86
Ofbiz	67	937	1004	84	920	1004	48	331	379	40	305	345
Olingo-odata	75	624	699	107	592	699	33	251	284	18	304	322
Openjpa	359	1929	2288	57	2231	2288	13	1221	1234	8	1263	1271
Openmeetings	28	382	410	37	373	410	4	129	133	1	143	144
Opennlp	4	405	409	30	379	409	3	135	138	4	137	141
Openwebbeans	18	366	384	40	344	384	0	106	106	2	109	111
Parquet	6	361	367	28	339	367	0	169	169	0	177	177
Phoenix	0	142	142	18	124	142	1	42	43	1	36	37
Plc4x	3	85	88	4	84	88	0	30	30	0	33	33
Pulsar	54	1140	1194	19	1174	1193	10	437	447	13	405	418
Qpid-jms	22	236	258	32	226	258	0	133	133	3	134	137

Ranger	77	573	650	56	594	650	12	257	269	6	247	253
Reef	11	610	621	66	555	621	1	123	124	1	140	141
Roller-weblogger	58	302	360	42	318	360	4	126	130	3	137	140
Samza	15	578	593	56	537	593	2	252	254	2	280	282
Shiro	13	259	272	50	222	272	2	171	173	3	172	175
Storm	20	712	732	2	730	732	74	690	764	77	621	698
Struts	75	428	503	69	434	503	5	258	263	1	301	302
Syncope	72	947	1019	107	912	1019	19	429	448	10	448	458
Tajo	25	1077	1102	83	1019	1102	21	400	421	13	424	437
Tez	29	655	684	23	661	684	15	293	308	8	300	308
Tika	13	512	525	48	477	525	7	184	191	4	183	187
Tinkerpop	3	397	400	25	375	400	30	221	251	4	270	274
Tomcat	87	1014	1101	77	1023	1100	32	496	528	23	521	544
Uima-ruta	28	680	708	68	640	708	24	296	320	8	274	282
Wicket	37	1280	1317	114	1203	1317	9	474	483	4	583	587
Zeppelin	5	401	406	2	404	406	15	197	212	15	183	198
Grand Total	3317	68095	71412	4438	67075	71513	2016	31350	33366	1581	32765	34346

APPENDIX E: Cross-project performance heatmap

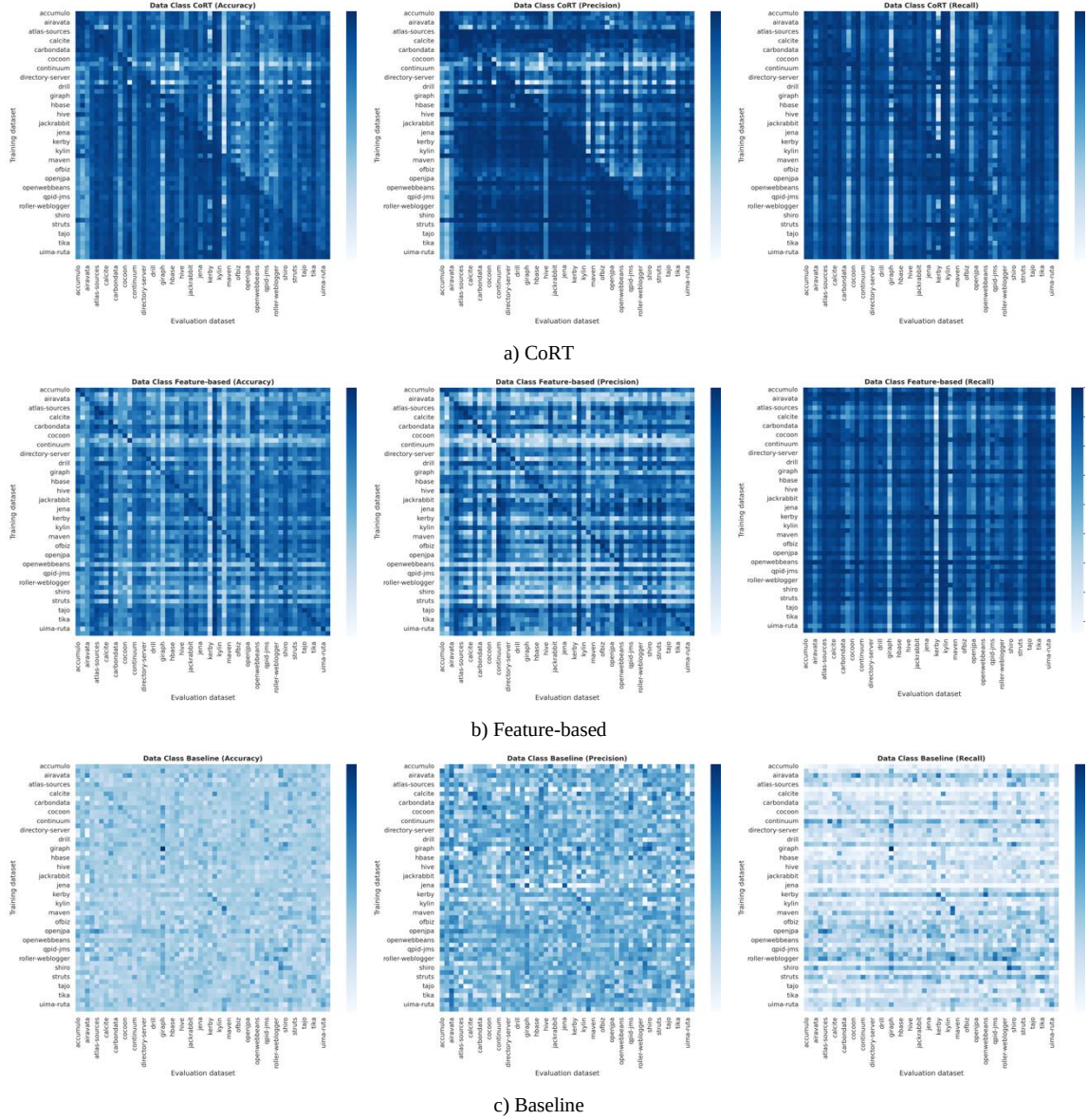
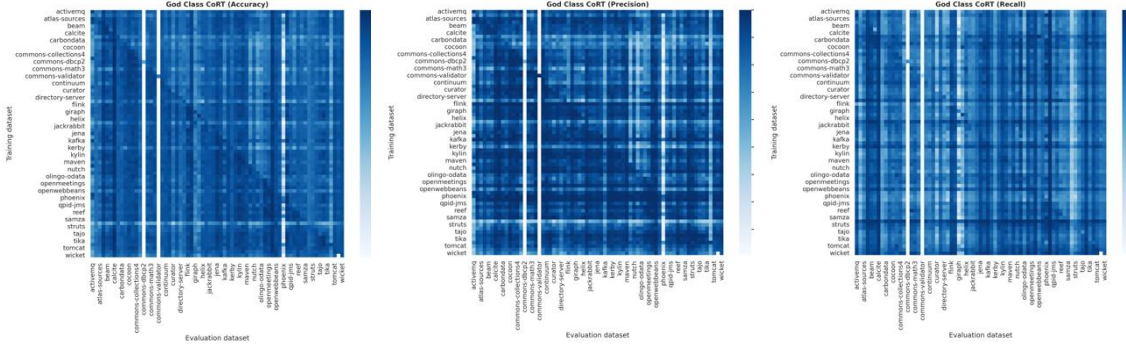
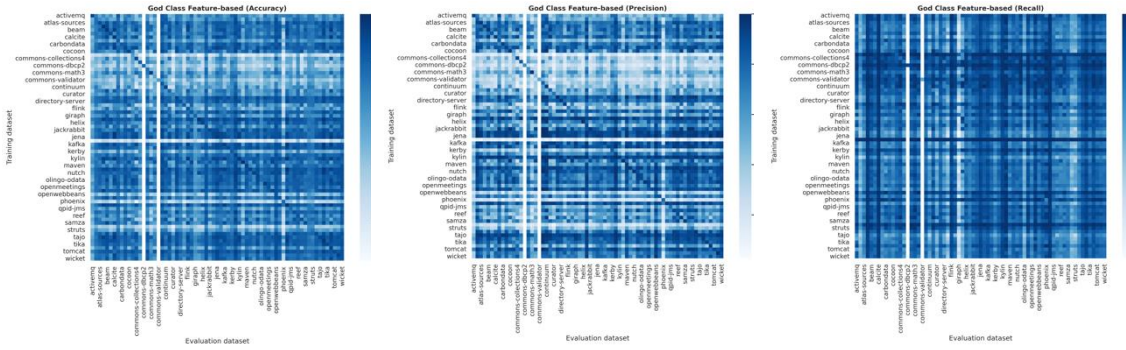


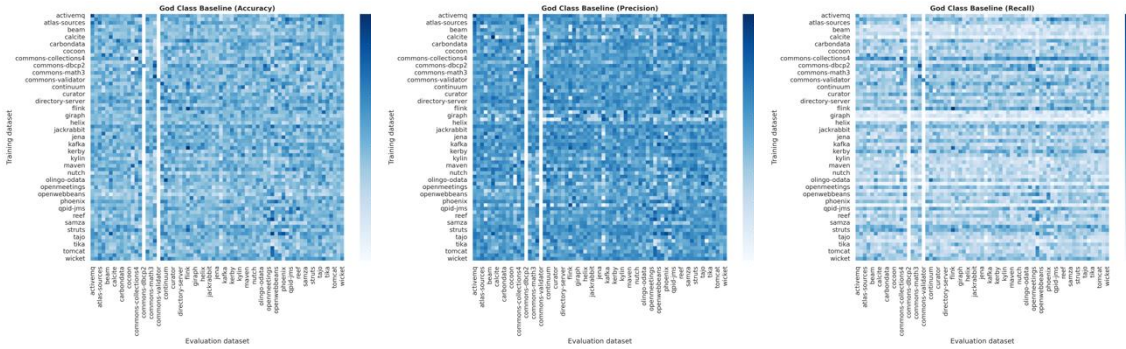
Figure 36 Cross-project performance heatmap for Data Class



a) CoRT

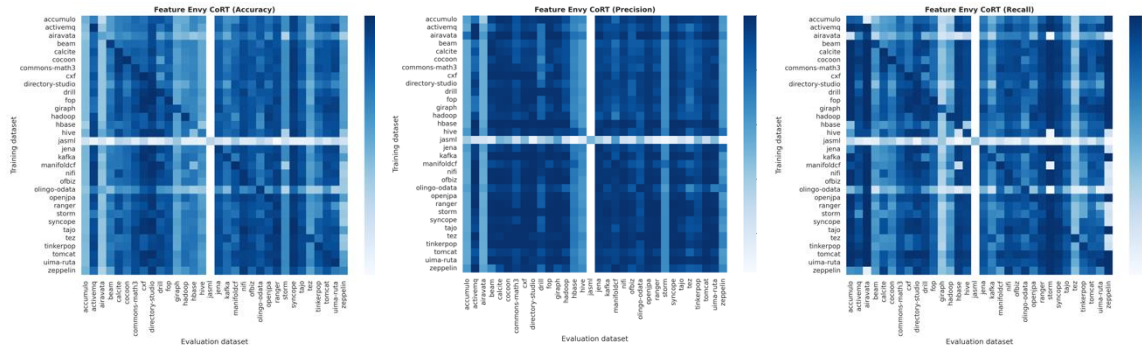


b) Feature-based

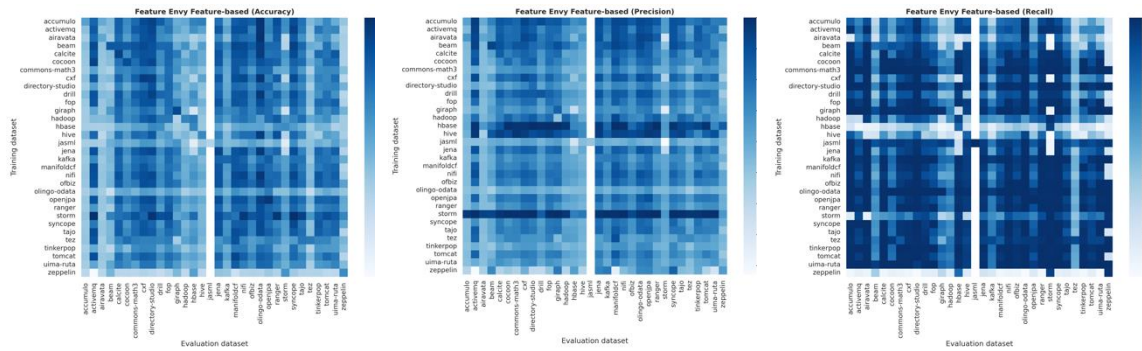


c) Baseline

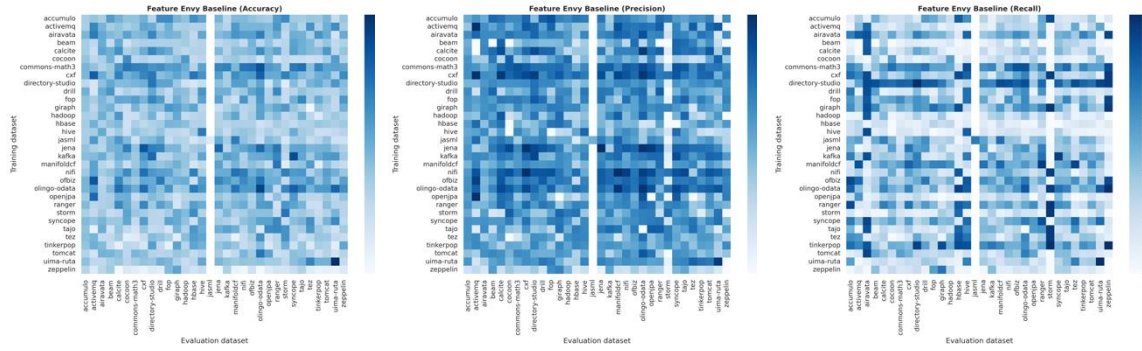
Figure 37 Cross-project performance heatmap for God Class



a) CoRT

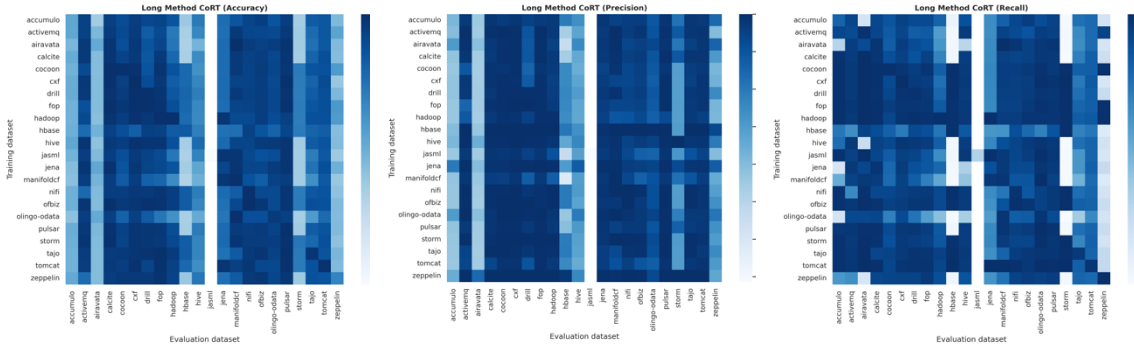


b) Feature-based

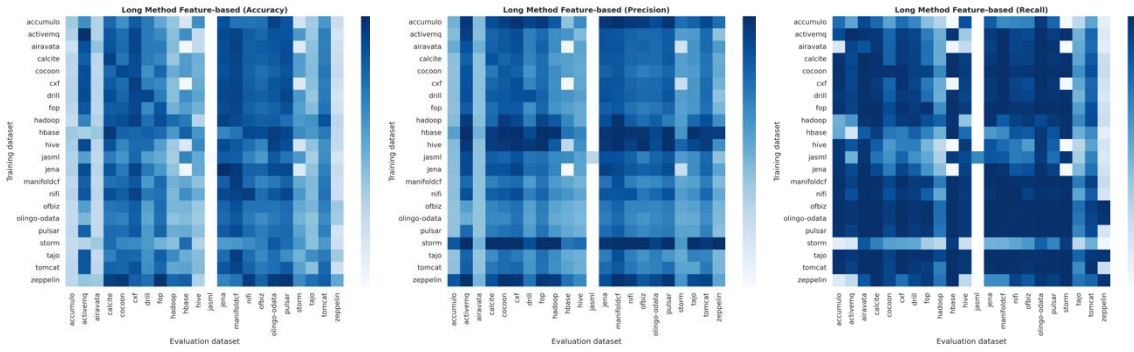


c) Baseline

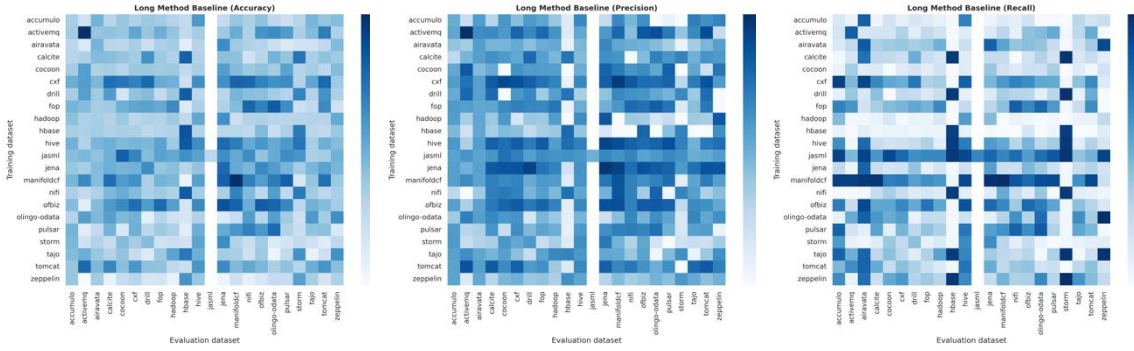
Figure 38 Cross-project performance heatmap for Feature Env



a) CoRT



b) Feature-based



c) Baseline

Figure 39 Cross-project performance heatmap for Long Method

References

- [1] W. H. Brown, R. C. Malveau, H. W. “Skip” McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st ed. USA: John Wiley & Sons, Inc., 1998.
- [2] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, and E. Gamma, *Refactoring: Improving the Design of Existing Code*, 1 edition. Reading, MA: Addison-Wesley Professional, 1999.
- [3] A. Yamashita and S. Counsell, “Code smells as system-level indicators of maintainability: An empirical study,” *J. Syst. Softw.*, vol. 10, no. 86, pp. 2639–2653, 2013, doi: 10.1016/j.jss.2013.05.007.
- [4] J. Pérez, “Refactoring Planning for Design Smell Correction: Summary, Opportunities and Lessons Learned,” in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, in ICSM ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 572–577. doi: 10.1109/ICSM.2013.98.
- [5] H. Züllighoven, “12 - The Development Process,” in *Object-Oriented Construction Handbook*, H. Züllighoven, Ed., San Francisco: Morgan Kaufmann, 2005, pp. 393–457. doi: 10.1016/B978-155860687-6/50012-8.
- [6] M. Kim, T. Zimmermann, and N. Nagappan, “A Field Study of Refactoring Challenges and Benefits,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, in FSE ’12. New York, NY, USA: ACM, 2012, p. 50:1-50:11. doi: 10.1145/2393596.2393655.
- [7] G. G. Roy and V. E. Veraart, “Software engineering education: from an engineering perspective,” in *Proceedings 1996 International Conference Software Engineering: Education and Practice*, Jan. 1996, pp. 256–262. doi: 10.1109/SEEP.1996.534008.
- [8] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, “Software complexity and maintenance costs,” *Commun. ACM*, vol. 36, no. 11, pp. 81–94, Nov. 1993, doi: 10.1145/163359.163375.
- [9] M. M. Lehman and J. F. Ramil, “Software Evolution and Software Evolution Processes,” *Ann. Softw. Eng.*, vol. 14, no. 1, pp. 275–309, Dec. 2002, doi: 10.1023/A:1020557525901.
- [10] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, “Quantifying the Effect of Code Smells on Maintenance Effort,” *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1144–1156, Aug. 2013, doi: 10.1109/TSE.2012.89.
- [11] A. Lozano, M. Wermelinger, and B. Nuseibeh, “Assessing the impact of bad smells using historical information,” in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, in IWPSE ’07. New York, NY, USA: Association for Computing Machinery, Sep. 2007, pp. 31–34. doi: 10.1145/1294948.1294957.
- [12] R. Arcoverde, A. Garcia, and E. Figueiredo, “Understanding the longevity of code smells: preliminary results of an explanatory survey,” in *Proceedings of the 4th Workshop on Refactoring Tools*, in WRT ’11. New York, NY, USA: Association for Computing Machinery, May 2011, pp. 33–36. doi: 10.1145/1984732.1984740.
- [13] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, “Do They Really Smell Bad? A Study on Developers’ Perception of Bad Code Smells,” in *2014 IEEE*

- International Conference on Software Maintenance and Evolution*, Sep. 2014, pp. 101–110. doi: 10.1109/ICSME.2014.32.
- [14] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, “An exploratory study of the impact of antipatterns on class change- and fault-proneness,” *Empir. Softw. Eng.*, vol. 17, no. 3, pp. 243–275, Jun. 2012, doi: 10.1007/s10664-011-9171-y.
 - [15] A. AbuHassan, M. Alshayeb, and L. Ghouti, “Software smell detection techniques: A systematic literature review,” *J. Softw. Evol. Process*, vol. 33, no. 3, p. e2320, 2021, doi: <https://doi.org/10.1002/smr.2320>.
 - [16] B. B. Mayvan, A. Rasoolzadegan, and A. J. Jafari, “Bad smell detection using quality metrics and refactoring opportunities,” *J. Softw. Evol. Process*, vol. 32, no. 8, p. e2255, 2020, doi: <https://doi.org/10.1002/smr.2255>.
 - [17] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, “DECOR: A Method for the Specification and Detection of Code and Design Smells,” *IEEE Trans Softw Eng*, vol. 36, no. 1, pp. 20–36, Jan. 2010, doi: 10.1109/TSE.2009.50.
 - [18] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, “Code smells and refactoring: A tertiary systematic review of challenges and observations,” *J. Syst. Softw.*, vol. 167, p. 110610, Sep. 2020, doi: 10.1016/j.jss.2020.110610.
 - [19] A. Al-Shaaby, H. Aljamaan, and M. Alshayeb, “Bad Smell Detection Using Machine Learning Techniques: A Systematic Literature Review,” *Arab. J. Sci. Eng.*, Jan. 2020, doi: 10.1007/s13369-019-04311-w.
 - [20] Z. Chen, L. Chen, W. Ma, X. Zhou, Y. Zhou, and B. Xu, “Understanding metric-based detectable smells in Python software: A comparative study,” *Inf. Softw. Technol.*, vol. 94, pp. 14–29, Feb. 2018, doi: 10.1016/j.infsof.2017.09.011.
 - [21] S. B. Kotsiantis, “Supervised Machine Learning: A Review of Classification Techniques,” in *Proceedings of the 2007 conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*, NLD: IOS Press, Jun. 2007, pp. 3–24.
 - [22] X. Liu *et al.*, “Self-supervised Learning: Generative or Contrastive,” *ArXiv200608218 Cs Stat*, Mar. 2021, Accessed: Apr. 26, 2021. [Online]. Available: <http://arxiv.org/abs/2006.08218>
 - [23] H. Mumtaz, M. Alshayeb, S. Mahmood, and M. Niazi, “A survey on UML model smells detection techniques for software refactoring,” *J. Softw. Evol. Process*, vol. 31, no. 3, p. e2154, 2019, doi: 10.1002/smr.2154.
 - [24] A. Alazba, H. Aljamaan, and M. Alshayeb, “Deep learning approaches for bad smell detection: a systematic literature review,” *Empir. Softw. Eng.*, vol. 28, no. 3, p. 77, May 2023, doi: 10.1007/s10664-023-10312-z.
 - [25] M. Misbhaudhin and M. Alshayeb, “UML model refactoring: a systematic literature review,” *Empir. Softw. Eng.*, vol. 20, no. 1, pp. 206–251, Feb. 2015, doi: 10.1007/s10664-013-9283-7.
 - [26] M.-A. Storey and A. Zagalsky, “Disrupting developer productivity one bot at a time,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, in FSE 2016. New York, NY, USA: Association for Computing Machinery, Nov. 2016, pp. 928–931. doi: 10.1145/2950290.2983989.

- [27] A. Abdellatif, K. Badran, and E. Shihab, “MSRBot: Using bots to answer questions from software repositories,” *Empir. Softw. Eng.*, vol. 25, no. 3, pp. 1834–1863, May 2020, doi: 10.1007/s10664-019-09788-5.
- [28] M. Wyrich and J. Bogner, “Towards an autonomous bot for automatic source code refactoring,” in *Proceedings of the 1st International Workshop on Bots in Software Engineering*, in BotSE ’19. Montreal, Quebec, Canada: IEEE Press, May 2019, pp. 24–28. doi: 10.1109/BotSE.2019.00015.
- [29] V. Alizadeh, M. A. Ouali, M. Kessentini, and M. Chater, “RefBot: Intelligent Software Refactoring Bot,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2019, pp. 823–834. doi: 10.1109/ASE.2019.00081.
- [30] S. Rebai, O. Ben Sghaier, V. Alizadeh, M. Kessentini, and M. Chater, “Interactive Refactoring Documentation Bot,” in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sep. 2019, pp. 152–162. doi: 10.1109/SCAM.2019.00026.
- [31] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus, “How to design a program repair bot? insights from the repairnator project,” in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, in ICSE-SEIP ’18. New York, NY, USA: Association for Computing Machinery, May 2018, pp. 95–104. doi: 10.1145/3183519.3183540.
- [32] V. Balachandran, “Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation,” in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 931–940. doi: 10.1109/ICSE.2013.6606642.
- [33] V. Balachandran, “Fix-it: An extensible code auto-fix component in Review Bot,” in *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sep. 2013, pp. 167–172. doi: 10.1109/SCAM.2013.6648198.
- [34] M. Wessel *et al.*, “The Power of Bots: Characterizing and Understanding Bots in OSS Projects,” *Proc. ACM Hum.-Comput. Interact.*, vol. 2, no. CSCW, p. 182:1-182:19, Nov. 2018, doi: 10.1145/3274451.
- [35] M. Wyrich, R. Hebig, S. Wagner, and R. Scandariato, “Perception and Acceptance of an Autonomous Refactoring Bot,” *Proc. 12th Int. Conf. Agents Artif. Intell.*, pp. 303–310, 2020, doi: 10.5220/0009168803030310.
- [36] W. C. Wake, *Refactoring Workbook*, 1st ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2003.
- [37] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. USA: Prentice Hall PTR, 2003.
- [38] A. Hunt and D. Thomas, *The pragmatic programmer: from journeyman to master*. USA: Addison-Wesley Longman Publishing Co., Inc., 2000.
- [39] W. H. Brown, R. C. Malveau, H. W. “Skip” McCormick, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st ed. USA: John Wiley & Sons, Inc., 1998.
- [40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

- [41] M. Mantyla, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, Sep. 2003, pp. 381–384. doi: 10.1109/ICSM.2003.1235447.
- [42] T. Sharma, "A survey on software smells.," *J Syst Softw*, vol. 138, pp. 158–173, 2018, doi: 10.1016/j.jss.2017.12.034.
- [43] S. G. Ganesh, "Towards a Principle-based Classification of Structural Design Smells.," *J Object Technol*, vol. 12, no. 2, p. 1: 1-29, 2013, doi: 10.5381/jot.2013.12.2.a1.
- [44] C. Bishop, *Pattern Recognition and Machine Learning*. in Information Science and Statistics. New York: Springer-Verlag, 2006. Accessed: Mar. 17, 2020. [Online]. Available: <https://www.springer.com/gp/book/9780387310732>
- [45] Ethem Alpaydin, "Introduction to Machine Learning," MIT Press. Accessed: May 18, 2020. [Online]. Available: <https://mitpress.mit.edu/books/introduction-machine-learning>
- [46] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE Trans. Syst. Man Cybern.*, vol. 21, no. 3, pp. 660–674, Jun. 1991, doi: 10.1109/21.97458.
- [47] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, Sep. 1995, doi: 10.1007/BF00994018.
- [48] P. M. Domingos and M. J. Pazzani, "Beyond independence: conditions for the optimality of the simple Bayesian classifier," in *Proceedings of the Thirteenth International Conference on International Conference on Machine Learning*, in ICML'96. Bari, Italy: Morgan Kaufmann Publishers Inc., Jul. 1996, pp. 105–112.
- [49] C. W. Dawson and R. L. Wilby, "Hydrological modelling using artificial neural networks," *Prog. Phys. Geogr. Earth Environ.*, vol. 25, no. 1, pp. 80–108, Mar. 2001, doi: 10.1177/030913330102500104.
- [50] L. Bottou, "Large-Scale Machine Learning with Stochastic Gradient Descent," in *Proceedings of COMPSTAT'2010*, Y. Lechevallier and G. Saporta, Eds., Heidelberg: Physica-Verlag HD, 2010, pp. 177–186. doi: 10.1007/978-3-7908-2604-3_16.
- [51] C. E. Rasmussen, "Gaussian Processes in Machine Learning," in *Advanced Lectures on Machine Learning: ML Summer Schools 2003, Canberra, Australia, February 2 - 14, 2003, Tübingen, Germany, August 4 - 16, 2003, Revised Lectures*, O. Bousquet, U. von Luxburg, and G. Rätsch, Eds., in Lecture Notes in Computer Science. , Berlin, Heidelberg: Springer, 2004, pp. 63–71. doi: 10.1007/978-3-540-28650-9_4.
- [52] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Trans. Inf. Theory*, vol. 13, no. 1, pp. 21–27, Jan. 1967, doi: 10.1109/TIT.1967.1053964.
- [53] K. Fukunaga, *Introduction to statistical pattern recognition (2nd ed.)*. USA: Academic Press Professional, Inc., 1990.
- [54] R. Gentleman and V. J. Carey, "Unsupervised Machine Learning," in *Bioconductor Case Studies*, F. Hahne, W. Huber, R. Gentleman, and S. Falcon, Eds., in Use R! , New York, NY: Springer, 2008, pp. 137–157. doi: 10.1007/978-0-387-77240-0_10.
- [55] S. Lloyd, "Least squares quantization in PCM," *IEEE Trans. Inf. Theory*, vol. 28, no. 2, pp. 129–137, Mar. 1982, doi: 10.1109/TIT.1982.1056489.
- [56] S. C. Johnson, "Hierarchical clustering schemes," *Psychometrika*, vol. 32, no. 3, pp. 241–254, Sep. 1967, doi: 10.1007/BF02289588.

- [57] H. Hotelling, "Analysis of a complex of statistical variables into principal components," *J. Educ. Psychol.*, vol. 24, no. 6, pp. 417–441, 1933, doi: 10.1037/h0071325.
- [58] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, Art. no. 7553, May 2015, doi: 10.1038/nature14539.
- [59] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bull. Math. Biophys.*, vol. 5, no. 4, pp. 115–133, Dec. 1943, doi: 10.1007/BF02478259.
- [60] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, Jul. 2006, doi: 10.1162/neco.2006.18.7.1527.
- [61] J. Patterson and A. Gibson, *Deep Learning: A Practitioner's Approach*, 1st edition. O'Reilly Media, 2017.
- [62] Y. Yang, X. Xia, D. Lo, and J. Grundy, "A Survey on Deep Learning for Software Engineering," *ArXiv201114597 Cs*, Nov. 2020, Accessed: Apr. 04, 2021. [Online]. Available: <http://arxiv.org/abs/2011.14597>
- [63] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep Learning*. MIT Press, 2016.
- [64] A. M. TURING, "I.—COMPUTING MACHINERY AND INTELLIGENCE," *Mind*, vol. LIX, no. 236, pp. 433–460, Oct. 1950, doi: 10.1093/mind/LIX.236.433.
- [65] C. R. Lebeuf, "A taxonomy of software bots: towards a deeper understanding of software bot characteristics," Thesis, 2018. Accessed: Apr. 27, 2021. [Online]. Available: <https://dspace.library.uvic.ca/handle/1828/10004>
- [66] C. Lebeuf, M.-A. Storey, and A. Zagalsky, "How Software Developers Mitigate Collaboration Friction with Chatbots," *ArXiv170207011 Cs*, Feb. 2017, Accessed: Apr. 27, 2021. [Online]. Available: <http://arxiv.org/abs/1702.07011>
- [67] L. K. Fryer, M. Ainley, A. Thompson, A. Gibson, and Z. Sherlock, "Stimulating and sustaining interest in a language course: An experimental comparison of Chatbot and Human task partners," *Comput. Hum. Behav.*, vol. 75, pp. 461–468, Oct. 2017, doi: 10.1016/j.chb.2017.05.045.
- [68] J. Pereira, "Leveraging chatbots to improve self-guided learning through conversational quizzes," in *Proceedings of the Fourth International Conference on Technological Ecosystems for Enhancing Multiculturality*, in TEEM '16. New York, NY, USA: Association for Computing Machinery, Nov. 2016, pp. 911–918. doi: 10.1145/3012430.3012625.
- [69] S. Tamayo-Moreno and D. Pérez-Marín, "Designing and Evaluating Pedagogic Conversational Agents to Teach Children," *Int. J. Educ. Pedagog. Sci.*, vol. 11, no. 3, pp. 521–526, Jan. 2017.
- [70] N. T. Thomas, "An e-business chatbot using AIML and LSA," in *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, Sep. 2016, pp. 2740–2742. doi: 10.1109/ICACCI.2016.7732476.
- [71] U. Gnewuch, S. Morana, and A. Maedche, "Towards Designing Cooperative and Social Conversational Agents for Customer Service," *Proc. 38th Int. Conf. Inf. Syst. ICIS Seoul ROK Dec. 10-13 2017 Res.--Prog. Pap.*, 2017, Accessed: Apr. 27, 2021. [Online]. Available: <https://publikationen.bibliothek.kit.edu/1000077139>

- [72] Winarsky, Norman, Mark, Bill, and Kressel, Henry, “The Development of Siri and the SRI Venture Creation Process.” SRI International, Menlo Park, USA, Tech. Rep, 2012.
- [73] “Google Assistant, your own personal Google,” Assistant. Accessed: Apr. 27, 2021. [Online]. Available: <https://assistant.google.com/>
- [74] M. Lei, H. Li, J. Li, N. Aundhkar, and D.-K. Kim, “Deep learning application on code clone detection: A review of current knowledge,” *J. Syst. Softw.*, vol. 184, no. C, Feb. 2022, doi: 10.1016/j.jss.2021.111141.
- [75] F. Sabir, F. Palma, G. Rasool, Y.-G. Guéhéneuc, and N. Moha, “A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems,” *Softw. Pract. Exp.*, vol. 49, no. 1, pp. 3–39, 2019, doi: <https://doi.org/10.1002/spe.2639>.
- [76] R. S. Menshaw, A. H. Yousef, and A. Salem, “Code Smells and Detection Techniques: A Survey,” in *2021 International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC)*, May 2021, pp. 78–83. doi: 10.1109/MIUCC52538.2021.9447669.
- [77] K. Alkharabsheh, Y. Crespo, E. Manso, and J. A. Taboada, “Software Design Smell Detection: a systematic mapping study,” *Softw. Qual. J.*, vol. 27, no. 3, pp. 1069–1148, Sep. 2019, doi: 10.1007/s11219-018-9424-8.
- [78] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, “Machine learning techniques for code smell detection: A systematic literature review and meta-analysis,” *Inf. Softw. Technol.*, vol. 108, pp. 115–138, Apr. 2019, doi: 10.1016/j.infsof.2018.12.009.
- [79] F. L. Caram, B. R. D. O. Rodrigues, A. S. Campanelli, and F. S. Parreiras, “Machine Learning Techniques for Code Smells Detection: A Systematic Mapping Study,” *Int. J. Softw. Eng. Knowl. Eng.*, vol. 29, no. 02, pp. 285–316, Feb. 2019, doi: 10.1142/S021819401950013X.
- [80] A. Kaur, S. Jain, S. Goel, and G. Dhiman, “A Review on Machine-learning Based Code Smell Detection Techniques in Object-oriented Software System(s),” 2020. doi: 10.2174/2352096513999200922125839.
- [81] T. Lewowski and L. Madeyski, “Code Smells Detection Using Artificial Intelligence Techniques: A Business-Driven Systematic Review,” in *Developments in Information & Knowledge Management for Business Applications : Volume 3*, N. Kryvinska and A. Poniszewska-Marańda, Eds., in *Studies in Systems, Decision and Control.*, Cham: Springer International Publishing, 2022, pp. 285–319. doi: 10.1007/978-3-030-77916-0_12.
- [82] B. Kitchenham and S. Charters, “Guidelines for performing Systematic Literature Reviews in Software Engineering.” 2007.
- [83] Anne-Wil Harzing, “Publish or Perish,” Harzing.com. Accessed: Jan. 23, 2022. [Online]. Available: <https://harzing.com/resources/publish-or-perish>
- [84] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, “Systematic literature reviews in software engineering – A systematic literature review,” *Inf. Softw. Technol.*, vol. 51, no. 1, pp. 7–15, Jan. 2009, doi: 10.1016/j.infsof.2008.09.009.
- [85] J. Wen, S. Li, Z. Lin, Y. Hu, and C. Huang, “Systematic literature review of machine learning based software development effort estimation models,” *Inf. Softw. Technol.*, vol. 54, no. 1, pp. 41–59, Jan. 2012, doi: 10.1016/j.infsof.2011.09.002.

- [86] B. Kitchenham, "Procedures for Performing Systematic Reviews," *undefined*, 2004, Accessed: Jan. 23, 2022. [Online]. Available: <https://www.semanticscholar.org/paper/Procedures-for-Performing-Systematic-Reviews-Kitchenham/29890a936639862f45cb9a987dd599dce9759bf5>
- [87] A. Jaiswal, A. R. Babu, M. Z. Zadeh, D. Banerjee, and F. Makedon, "A Survey on Contrastive Self-Supervised Learning," *Technologies*, vol. 9, no. 1, Art. no. 1, Mar. 2021, doi: 10.3390/technologies9010002.
- [88] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a Big Data Curated Benchmark of Inter-project Code Clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sep. 2014, pp. 476–480. doi: 10.1109/ICSME.2014.77.
- [89] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empir. Softw. Eng.*, vol. 21, no. 3, pp. 1143–1191, Jun. 2016, doi: 10.1007/s10664-015-9378-4.
- [90] F. Palomba *et al.*, "Landfill: An Open Dataset of Code Smells with Public Evaluation," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, May 2015, pp. 482–485. doi: 10.1109/MSR.2015.69.
- [91] Martin Fowler, *Refactoring: improving the design of existing code*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
- [92] G. Suryanarayana, G. Samarthayam, and T. Sharma, "Refactoring for Software Design Smells: Managing Technical Debt, Chapter 2 - Design Smells," in *Refactoring for Software Design Smells*, G. Suryanarayana, G. Samarthayam, and T. Sharma, Eds., Boston: Morgan Kaufmann, 2015, pp. 9–19. doi: 10.1016/B978-0-12-801397-7.00002-3.
- [93] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: Identification and Removal of Type-Checking Bad Smells," in *2008 12th European Conference on Software Maintenance and Reengineering*, Apr. 2008, pp. 329–331. doi: 10.1109/CSMR.2008.4493342.
- [94] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu, and R. Wettel, "iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design," in *Proceedings of the 21st IEEE International Conference on Software Maintenance - Industrial and Tool volume, ICSM 2005, 25-30 September 2005, Budapest, Hungary, 2005*, pp. 77–80.
- [95] Y. Bengio, A. C. Courville, and P. Vincent, "Representation Learning: A Review and New Perspectives," *IEEE Trans Pattern Anal Mach Intell*, vol. 35, no. 8, pp. 1798–1828, 2013, doi: 10.1109/TPAMI.2013.50.
- [96] S. Fakhoury, V. Arnaoudova, C. Noiseux, F. Khomh, and G. Antoniol, "Keep it simple: Is deep learning good for linguistic smell detection?," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Campobasso: IEEE, Mar. 2018, pp. 602–611. doi: 10.1109/SANER.2018.8330265.
- [97] C. Guo *et al.*, "Review Sharing via Deep Semi-Supervised Code Clone Detection," *IEEE Access*, vol. 8, pp. 24948–24965, 2020, doi: 10.1109/ACCESS.2020.2966532.
- [98] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated Graph Sequence Neural Networks." arXiv, Sep. 22, 2017. doi: 10.48550/arXiv.1511.05493.

- [99] F. Pecorelli, D. D. Nucci, C. D. Roover, and A. D. Lucia, “A large empirical assessment of the role of data balancing in machine-learning-based code smell detection,” *J Syst Softw*, 2020, doi: 10.1016/j.jss.2020.110693.
- [100] E. M. Karabulut, S. A. Özel, and T. İbrikçi, “A comparative study on the effect of feature selection on classification accuracy,” *Procedia Technol.*, vol. 1, pp. 323–327, Jan. 2012, doi: 10.1016/j.protcy.2012.02.068.
- [101] M. A. Khan *et al.*, “Memory-Augmented Neural Networks for Predictive Process Analytics,” *undefined*, 2018, Accessed: Jan. 07, 2022. [Online]. Available: <https://www.semanticscholar.org/paper/Memory-Augmented-Neural-Networks-for-Predictive-Khan-Le/af3fc5112f90fe96fa9ad93e29d5b7cca1d6e35e>
- [102] I. H. Sarker, “Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions,” *SN Comput. Sci.*, vol. 2, no. 6, p. 420, Aug. 2021, doi: 10.1007/s42979-021-00815-1.
- [103] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, “On the importance of initialization and momentum in deep learning,” in *Proceedings of the 30th International Conference on Machine Learning*, PMLR, May 2013, pp. 1139–1147. Accessed: Jan. 17, 2022. [Online]. Available: <https://proceedings.mlr.press/v28/sutskever13.html>
- [104] Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Y. Ng, “On optimization methods for deep learning,” in *Proceedings of the 28th International Conference on International Conference on Machine Learning*, in ICML’11. Madison, WI, USA: Omnipress, Jun. 2011, pp. 265–272.
- [105] C. Sammut and G. I. Webb, *Encyclopedia of Machine Learning*. Springer Science & Business Media, 2011.
- [106] H. He and E. A. Garcia, “Learning from Imbalanced Data,” *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 9, pp. 1263–1284, Sep. 2009, doi: 10.1109/TKDE.2008.239.
- [107] G. Brier, “VERIFICATION OF FORECASTS EXPRESSED IN TERMS OF PROBABILITY,” 1950, doi: 10.1175/1520-0493(1950)078<0001:VOFEIT>2.0.CO;2.
- [108] D. Chicco and G. Jurman, “The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation,” *BMC Genomics*, vol. 21, p. 6, Jan. 2020, doi: 10.1186/s12864-019-6413-7.
- [109] Y. Ma and H. He, Eds., *Imbalanced Learning: Foundations, Algorithms, and Applications*, 1st edition. Hoboken, New Jersey: Wiley-IEEE Press, 2013.
- [110] H. Wei and M. Li, “Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code,” in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, Melbourne, Australia: International Joint Conferences on Artificial Intelligence Organization, Aug. 2017, pp. 3034–3040. doi: 10.24963/ijcai.2017/423.
- [111] L. Jiang, G. Mishnerghi, Z. Su, and S. Glondu, “DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones,” in *29th International Conference on Software Engineering (ICSE’07)*, May 2007, pp. 96–105. doi: 10.1109/ICSE.2007.30.
- [112] H. Sajjani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, “SourcererCC: Scaling Code Clone Detection to Big-Code,” in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 1157–1168. doi: 10.1145/2884781.2884877.

- [113] C. E. Rasmussen and Z. Ghahramani, "Occam's Razor," in *In Advances in Neural Information Processing Systems 13*, MIT Press, 2001, pp. 294–300.
- [114] D. S. Cruzes and T. Dybå, "Research synthesis in software engineering," *Inf. Softw. Technol.*, vol. 53, no. 5, pp. 440–455, May 2011, doi: 10.1016/j.infsof.2011.01.004.
- [115] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A Systematic Literature Review on Fault Prediction Performance in Software Engineering," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, Nov. 2012, doi: 10.1109/TSE.2011.103.
- [116] S. Hosseini, B. Turhan, and D. Gunarathna, "A Systematic Literature Review and Meta-Analysis on Cross Project Defect Prediction," *IEEE Trans. Softw. Eng.*, vol. 45, no. 2, pp. 111–147, Feb. 2019, doi: 10.1109/TSE.2017.2770124.
- [117] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjøberg, "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems," in *2010 IEEE International Conference on Software Maintenance*, Sep. 2010, pp. 1–10. doi: 10.1109/ICSM.2010.5609564.
- [118] M. Zhang, T. Hall, and N. Baddoo, "Code Bad Smells: a review of current knowledge," *J. Softw. Maint. Evol. Res. Pract.*, vol. 23, no. 3, pp. 179–202, Apr. 2011, doi: 10.1002/smr.521.
- [119] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *ArXiv181004805 Cs*, May 2019, Accessed: Mar. 07, 2022. [Online]. Available: <http://arxiv.org/abs/1810.04805>
- [120] K. Ohri and M. Kumar, "Review on self-supervised image recognition using deep neural networks," *Knowl.-Based Syst.*, vol. 224, p. 107090, Jul. 2021, doi: 10.1016/j.knosys.2021.107090.
- [121] T. Guggulothu and S. A. Moiz, "Code smell detection using multi-label classification approach," *Softw. Qual. J.*, vol. 28, no. 3, pp. 1063–1086, Sep. 2020, doi: 10.1007/s11219-020-09498-y.
- [122] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The Impact of Automated Parameter Optimization on Defect Prediction Models," *IEEE Trans. Softw. Eng.*, vol. 45, no. 7, pp. 683–711, Jul. 2019, doi: 10.1109/TSE.2018.2794977.
- [123] T.-S. Lim, W.-Y. Loh, and Y.-S. Shih, "A Comparison of Prediction Accuracy, Complexity, and Training Time of Thirty-Three Old and New Classification Algorithms," *Mach. Learn.*, vol. 40, no. 3, pp. 203–228, Sep. 2000, doi: 10.1023/A:1007608224229.
- [124] A. Sheneamer, H. Hazazi, S. Roy, and J. Kalita, "Schemes for Labeling Semantic Code Clones using Machine Learning," in *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, Dec. 2017, pp. 981–985. doi: 10.1109/ICMLA.2017.00-25.
- [125] X. Zhou, Y. Jin, H. Zhang, S. Li, and X. Huang, "A Map of Threats to Validity of Systematic Literature Reviews in Software Engineering," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, Dec. 2016, pp. 153–160. doi: 10.1109/APSEC.2016.031.
- [126] S. Gidaris, P. Singh, and N. Komodakis, "Unsupervised Representation Learning by Predicting Image Rotations." *arXiv*, Mar. 20, 2018. doi: 10.48550/arXiv.1803.07728.
- [127] S. Charalampidou, A. Ampatzoglou, and P. Avgeriou, "Size and cohesion metrics as indicators of the long method bad smell: An empirical study," in *Proceedings of*

- the 11th International Conference on Predictive Models and Data Analytics in Software Engineering, in PROMISE '15. Beijing, China: Association for Computing Machinery, Oct. 2015, pp. 1–10. doi: 10.1145/2810146.2810155.
- [128] K. Alkharabsheh, Y. Crespo, E. Manso, and J. A. Taboada, “Software Design Smell Detection: a systematic mapping study,” *Softw. Qual. J.*, vol. 27, no. 3, pp. 1069–1148, Sep. 2019, doi: 10.1007/s11219-018-9424-8.
 - [129] A. Maiga *et al.*, “Support vector machines for anti-pattern detection,” in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2012, pp. 278–281. doi: 10.1145/2351676.2351723.
 - [130] A. Kaur, S. Jain, and S. Goel, “A Support Vector Machine Based Approach for Code Smell Detection,” in *2017 International Conference on Machine Learning and Data Science (MLDS)*, Dec. 2017, pp. 9–14. doi: 10.1109/MLDS.2017.8.
 - [131] S. Hassaine, F. Khomh, Y.-G. Gueheneuc, and S. Hamel, “IDS: An Immune-Inspired Approach for the Detection of Software Design Smells,” in *2010 Seventh International Conference on the Quality of Information and Communications Technology*, Sep. 2010, pp. 343–348. doi: 10.1109/QUATIC.2010.61.
 - [132] S. Bryton, F. Brito e Abreu, and M. Monteiro, “Reducing Subjectivity in Code Smells Detection: Experimenting with the Long Method,” in *2010 Seventh International Conference on the Quality of Information and Communications Technology*, Sep. 2010, pp. 337–342. doi: 10.1109/QUATIC.2010.60.
 - [133] Lucas Amorim, N. Antunes, B. Fonseca, and M. Ribeiro, “Experience report: Evaluating the effectiveness of decision trees for detecting code smells,” in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, Nov. 2015, pp. 261–269. doi: 10.1109/ISSRE.2015.7381819.
 - [134] F. Arcelli Fontana and M. Zanoni, “Code smell severity classification using machine learning techniques,” *Knowl.-Based Syst.*, vol. 128, pp. 43–58, Jul. 2017, doi: 10.1016/j.knosys.2017.04.014.
 - [135] D. D. Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. D. Lucia, “Detecting code smells using machine learning techniques: Are we there yet?,” *2018 IEEE 25th Int. Conf. Softw. Anal. Evol. Reengineering SANER*, pp. 612–621, 2018, doi: 10.1109/SANER.2018.8330266.
 - [136] A. Barbez, F. Khomh, and Y.-G. Guéhéneuc, “A Machine-learning Based Ensemble Method For Anti-patterns Detection,” *J. Syst. Softw.*, p. 110486, Nov. 2019, doi: 10.1016/j.jss.2019.110486.
 - [137] D. K. Kim, “Finding Bad Code Smells with Neural Network Models,” *Int. J. Electr. Comput. Eng. IJECE*, vol. 7, no. 6, pp. 3613–3621, Dec. 2017, doi: 10.11591/ijece.v7i6.pp3613-3621.
 - [138] S. Dewangan, R. S. Rao, A. Mishra, and M. Gupta, “A Novel Approach for Code Smell Detection: An Empirical Study,” *IEEE Access*, vol. 9, pp. 162869–162883, 2021, doi: 10.1109/ACCESS.2021.3133810.
 - [139] A. Abdou and N. Darwish, “Severity classification of software code smells using machine learning techniques: A comparative study,” *J. Softw. Evol. Process*, vol. n/a, no. n/a, p. e2454, Apr. 2022, doi: 10.1002/smr.2454.
 - [140] R. Sandouka and H. Aljamaan, “Python code smells detection using conventional machine learning models,” *PeerJ Comput. Sci.*, vol. 9, p. e1370, May 2023, doi: 10.7717/peerj-cs.1370.

- [141] H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, and L. Zhang, “Deep Learning Based Code Smell Detection,” *IEEE Trans. Softw. Eng.*, pp. 1–1, 2019, doi: 10.1109/TSE.2019.2936376.
- [142] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, “Code smell detection by deep direct-learning and transfer-learning,” *J. Syst. Softw.*, vol. 176, p. 110936, Jun. 2021, doi: 10.1016/j.jss.2021.110936.
- [143] X. Guo, C. Shi, and H. Jiang, “Deep semantic-Based Feature Envy Identification,” in *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, in Internetware ’19. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 1–6. doi: 10.1145/3361242.3361257.
- [144] A. Barbez, F. Khomh, and Y.-G. Gueheneuc, “Deep Learning Anti-Patterns from Code Metrics History,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Cleveland, OH, USA: IEEE, Sep. 2019, pp. 114–124. doi: 10.1109/ICSME.2019.00021.
- [145] N. A. A. Khleel and K. Nehéz, “Deep convolutional neural network model for bad code smells detection based on oversampling method,” *Indones. J. Electr. Eng. Comput. Sci.*, vol. 26, no. 3, Art. no. 3, Jun. 2022, doi: 10.11591/ijeecs.v26.i3.pp1725-1735.
- [146] M. Hadj-Kacem and N. Bouassida, “A Hybrid Approach To Detect Code Smells using Deep Learning,” in *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering*, in ENASE 2018. Setubal, PRT: SCITEPRESS - Science and Technology Publications, Lda, Mar. 2018, pp. 137–146. doi: 10.5220/0006709801370146.
- [147] M. Hadj-Kacem and N. Bouassida, “Deep Representation Learning for Code Smells Detection using Variational Auto-Encoder,” in *2019 International Joint Conference on Neural Networks (IJCNN)*, Jul. 2019, pp. 1–8. doi: 10.1109/IJCNN.2019.8851854.
- [148] M. Hadj-Kacem and N. Bouassida, “Improving the Identification of Code Smells by Combining Structural and Semantic Information,” in *Neural Information Processing*, T. Gedeon, K. W. Wong, and M. Lee, Eds., in Communications in Computer and Information Science. Cham: Springer International Publishing, 2019, pp. 296–304. doi: 10.1007/978-3-030-36808-1_32.
- [149] S. Ren, C. Shi, and S. Zhao, “Exploiting Multi-aspect Interactions for God Class Detection with Dataset Fine-tuning,” in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, Madrid, Spain: IEEE, Jul. 2021, pp. 864–873. doi: 10.1109/COMPSAC51774.2021.00119.
- [150] H. Wang, J. Liu, J. Kang, W. Yin, H. Sun, and H. Wang, “Feature Envy Detection based on Bi-LSTM with Self-Attention Mechanism,” in *2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCLOUD/SocialCom/SustainCom)*, Exeter, United Kingdom: IEEE, Dec. 2020, pp. 448–457. doi: 10.1109/ISPA-BDCLOUD-SocialCom-SustainCom51426.2020.00082.
- [151] X. Yin, C. Shi, and S. Zhao, “Local and Global Feature Based Explainable Feature Envy Detection,” in *2021 IEEE 45th Annual Computers, Software, and Applications*

- Conference (COMPSAC)*, Madrid, Spain: IEEE, Jul. 2021, pp. 942–951. doi: 10.1109/COMPSAC51774.2021.00127.
- [152] W. Xu, “Multi-Granularity Code Smell Detection using Deep Learning Method based on Abstract Syntax Tree,” presented at the The 33rd International Conference on Software Engineering and Knowledge Engineering, Jul. 2021, pp. 503–509. doi: 10.18293/SEKE2021-014.
 - [153] Z. Feng *et al.*, “CodeBERT: A Pre-Trained Model for Programming and Natural Languages.” arXiv, Sep. 18, 2020. doi: 10.48550/arXiv.2002.08155.
 - [154] D. Guo *et al.*, “GraphCodeBERT: Pre-training Code Representations with Data Flow.” arXiv, Sep. 13, 2021. doi: 10.48550/arXiv.2009.08366.
 - [155] S. Liu, B. Wu, X. Xie, G. Meng, and Y. Liu, “ContraBERT: Enhancing Code Pre-trained Models via Contrastive Learning.” arXiv, Jan. 22, 2023. doi: 10.48550/arXiv.2301.09072.
 - [156] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, “CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation.” arXiv, Sep. 02, 2021. doi: 10.48550/arXiv.2109.00859.
 - [157] Y. Wang, H. Le, A. D. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, “CodeT5+: Open Code Large Language Models for Code Understanding and Generation.” arXiv, May 20, 2023. doi: 10.48550/arXiv.2305.07922.
 - [158] S. Lu *et al.*, “CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation.” arXiv, Mar. 16, 2021. doi: 10.48550/arXiv.2102.04664.
 - [159] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, “CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning.” arXiv, Nov. 03, 2022. doi: 10.48550/arXiv.2207.01780.
 - [160] T. Alkhaeir and B. Walter, “The Effect of Code Smells on the Relationship Between Design Patterns and Defects,” *IEEE Access*, vol. 9, pp. 3360–3373, 2021, doi: 10.1109/ACCESS.2020.3047870.
 - [161] J. P. dos Reis, F. B. e. Abreu, and G. de F. Carneiro, “Crowdsmelling: A preliminary study on using collective knowledge in code smells detection,” *Empir. Softw. Eng.*, vol. 27, no. 3, p. 69, Mar. 2022, doi: 10.1007/s10664-021-10110-5.
 - [162] B. Sotto-Mayor, A. Elmishali, M. Kalech, and R. Abreu, “Exploring Design smells for smell-based defect prediction,” *Eng. Appl. Artif. Intell.*, vol. 115, p. 105240, Oct. 2022, doi: 10.1016/j.engappai.2022.105240.
 - [163] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed. Pragmatic Bookshelf, 2013.
 - [164] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, “Deep learning code fragments for code clone detection,” in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 2016, pp. 87–98.
 - [165] A. Vaswani *et al.*, “Attention Is All You Need.” arXiv, Dec. 05, 2017. doi: 10.48550/arXiv.1706.03762.
 - [166] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, “Optuna: A Next-generation Hyperparameter Optimization Framework,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, in KDD ’19. New York, NY, USA: Association for Computing Machinery, Jul. 2019, pp. 2623–2631. doi: 10.1145/3292500.3330701.

- [167] S. Watanabe and F. Hutter, “c-TPE: Generalizing Tree-structured Parzen Estimator with Inequality Constraints for Continuous and Categorical Hyperparameter Optimization.” arXiv, Nov. 25, 2022. doi: 10.48550/arXiv.2211.14411.
- [168] S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.” arXiv, Mar. 02, 2015. doi: 10.48550/arXiv.1502.03167.
- [169] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition.” arXiv, Dec. 10, 2015. doi: 10.48550/arXiv.1512.03385.
- [170] I. J. Myung, “The Importance of Complexity in Model Selection,” *J. Math. Psychol.*, vol. 44, no. 1, pp. 190–204, Mar. 2000, doi: 10.1006/jmps.1999.1283.
- [171] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, “UniXcoder: Unified Cross-Modal Pre-training for Code Representation.” arXiv, Mar. 07, 2022. doi: 10.48550/arXiv.2203.03850.
- [172] A. Alazba and H. Aljamaan, “Code smell detection using feature selection and stacking ensemble: An empirical investigation,” *Inf. Softw. Technol.*, vol. 138, p. 106648, Oct. 2021, doi: 10.1016/j.infsof.2021.106648.
- [173] W. Hua, Y. Sui, Y. Wan, G. Liu, and G. Xu, “FCCA: Hybrid Code Representation for Functional Clone Detection Using Attention Networks,” *IEEE Trans. Reliab.*, vol. 70, no. 1, pp. 304–318, Mar. 2021, doi: 10.1109/TR.2020.3001918.
- [174] K. W. Nafi, T. S. Kar, B. Roy, C. K. Roy, and K. A. Schneider, “CLCDSA: Cross Language Code Clone Detection using Syntactical Features and API Documentation,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, San Diego, CA, USA: IEEE, Nov. 2019, pp. 1026–1037. doi: 10.1109/ASE.2019.00099.
- [175] E. Hoffer, I. Hubara, and D. Soudry, “Train longer, generalize better: closing the generalization gap in large batch training of neural networks.” arXiv, Jan. 01, 2018. doi: 10.48550/arXiv.1705.08741.
- [176] A. Alazba, H. Aljamaan, and M. Alshayeb, “CoRT: Transformer-based Code Representations with Self-supervision by Predicting Reserved Words for Code Smell Detection,” *Empir. Softw. Eng. J.*, in press.
- [177] A. AbuHassan, M. Alshayeb, and L. Ghouti, “Detection of Design Smells Using Adaptive Neuro-Fuzzy Approaches,” *Int. J. Fuzzy Syst.*, vol. 24, no. 4, pp. 1927–1943, Jun. 2022, doi: 10.1007/s40815-022-01248-5.
- [178] R. Fourati, N. Bouassida, and H. B. Abdallah, “A Metric-Based Approach for Anti-pattern Detection in UML Designs,” in *Computer and Information Science 2011*, R. Lee, Ed., in Studies in Computational Intelligence. Berlin, Heidelberg: Springer, 2011, pp. 17–33. doi: 10.1007/978-3-642-21378-6_2.
- [179] M. Maddeh and S. Ayouni, “Extracting and Modeling Design Defects Using Gradual Rules and UML Profile,” in *Computer Science and Its Applications*, in IFIP Advances in Information and Communication Technology. Cham: Springer International Publishing, 2015, pp. 574–583. doi: 10.1007/978-3-319-19578-0_47.
- [180] M. H. Alalfi, E. P. Antony, and J. R. Cordy, “An approach to clone detection in sequence diagrams and its application to security analysis,” *Softw. Syst. Model.*, vol. 17, no. 4, pp. 1287–1309, Oct. 2018, doi: 10.1007/s10270-016-0557-6.

- [181] M. Misbhauddin and M. Alshayeb, “An integrated metamodel-based approach to software model refactoring,” *Softw. Syst. Model.*, vol. 18, no. 3, pp. 2013–2050, Jun. 2019, doi: 10.1007/s10270-017-0628-3.
- [182] D. Rattan, R. Bhatia, and M. Singh, “Model clone detection based on tree comparison,” in *2012 Annual IEEE India Conference (INDICON)*, Dec. 2012, pp. 1041–1046. doi: 10.1109/INDICON.2012.6420770.
- [183] A. Ghannem, G. El Boussaidi, and M. Kessentini, “On the use of design defect examples to detect model refactoring opportunities,” *Softw. Qual. J.*, vol. 24, no. 4, pp. 947–965, Dec. 2016, doi: 10.1007/s11219-015-9271-9.
- [184] M. Alshayeb, H. Mumtaz, S. Mahmood, and M. Niazi, “Improving the Security of UML Sequence Diagram Using Genetic Algorithm,” *IEEE Access*, vol. 8, pp. 62738–62761, 2020, doi: 10.1109/ACCESS.2020.2981742.
- [185] N. Maneerat and P. Muenchaisri, “Bad-smell prediction from software design model using machine learning techniques,” in *2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE)*, May 2011, pp. 331–336. doi: 10.1109/JCSSE.2011.5930143.
- [186] B. K. Sidhu, K. Singh, and N. Sharma, “A machine learning approach to software model refactoring,” *Int. J. Comput. Appl.*, pp. 1–12, Jan. 2020, doi: 10.1080/1206212X.2020.1711616.
- [187] M. Maddeh, S. Ayouni, S. Alyahya, and F. Hajjej, “Decision tree-based Design Defects Detection,” *IEEE Access*, vol. 9, pp. 71606–71614, 2021, doi: 10.1109/ACCESS.2021.3078724.
- [188] A. Barriga Rodriguez, L. Bettini, L. Iovino, A. Rutle, and R. Heldal, “Addressing the trade off between smells and quality when refactoring class diagrams,” 20, 2021, doi: 10.5381/jot.2021.20.3.a1.
- [189] R. Hebig, T. H. Quang, M. R. V. Chaudron, G. Robles, and M. A. Fernandez, “The quest for open source projects that use UML: mining GitHub,” in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, in MODELS ’16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 173–183. doi: 10.1145/2976767.2976778.
- [190] J. A. H. López, J. L. Cánovas Izquierdo, and J. S. Cuadrado, “ModelSet: a dataset for machine learning in model-driven engineering,” *Softw. Syst. Model.*, vol. 21, no. 3, pp. 967–986, Jun. 2022, doi: 10.1007/s10270-021-00929-3.
- [191] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, “BDTEX: A GQM-based Bayesian approach for the detection of antipatterns,” *J. Syst. Softw.*, vol. 84, no. 4, pp. 559–572, Apr. 2011, doi: 10.1016/j.jss.2010.11.921.
- [192] “TensorFlow | Google Open Source Projects,” Google Open Source. Accessed: Jan. 27, 2023. [Online]. Available: <https://opensource.google/projects/tensorflow>
- [193] D. Rosca and L. Domingues, “A Systematic Comparison of Roundtrip Software Engineering Approaches applied to UML Class Diagram,” *Procedia Comput. Sci.*, vol. 181, pp. 861–868, Jan. 2021, doi: 10.1016/j.procs.2021.01.240.
- [194] A. Carvalho, W. Luz, D. Marcílio, R. Bonifácio, G. Pinto, and E. Dias Canedo, “C-3PR: A Bot for Fixing Static Analysis Violations via Pull Requests,” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb. 2020, pp. 161–171. doi: 10.1109/SANER48275.2020.9054842.

- [195] D. Serban, B. Golsteijn, R. Holdorp, and A. Serebrenik, “SAW-BOT: Proposing Fixes for Static Analysis Warnings with GitHub Suggestions,” in *2021 IEEE/ACM Third International Workshop on Bots in Software Engineering (BotSE)*, Jun. 2021, pp. 26–30. doi: 10.1109/BotSE52550.2021.00013.
- [196] K. Etemadi *et al.*, “Sorald: Automatic Patch Suggestions for SonarQube Static Analysis Violations,” *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 04, pp. 2794–2810, Jul. 2023, doi: 10.1109/TDSC.2022.3167316.
- [197] A. Silva, M. Martinez, B. Danglot, D. Ginelli, and M. Monperrus, “FLACOCO: Fault Localization for Java based on Industry-grade Coverage.” arXiv, Mar. 16, 2023. doi: 10.48550/arXiv.2111.12513.
- [198] A. Marginean *et al.*, “SapFix: Automated End-to-End Repair at Scale,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, May 2019, pp. 269–278. doi: 10.1109/ICSE-SEIP.2019.00039.
- [199] T. Nakagawa, Y. Higo, and S. Kusumoto, “CLIONE: Clone Modification Support for Pull Request Based Development,” in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, Dec. 2020, pp. 455–459. doi: 10.1109/APSEC51365.2020.00055.
- [200] B. Baudry *et al.*, “A Software-Repair Robot Based on Continual Learning,” *IEEE Softw.*, vol. 38, no. 4, pp. 28–35, Jul. 2021, doi: 10.1109/MS.2021.3070743.
- [201] U. Lah, J. R. Lewis, and B. Šumak, “Perceived Usability and the Modified Technology Acceptance Model,” *Int. J. Human–Computer Interact.*, vol. 36, no. 13, pp. 1216–1230, Aug. 2020, doi: 10.1080/10447318.2020.1727262.
- [202] Y. Meng and L. Liu, “A Deep Learning Approach for a Source Code Detection Model Using Self-Attention,” *Complexity*, vol. 2020, pp. 1–15, Sep. 2020, doi: 10.1155/2020/5027198.
- [203] G. Li, Y. Tang, X. Zhang, and B. Yi, “A Deep Learning Based Approach to Detect Code Clones,” in *2020 International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI)*, Sanya, China: IEEE, Dec. 2020, pp. 337–340. doi: 10.1109/ICHCI51889.2020.00078.
- [204] D. K. Kim, “A Deep Neural Network-Based Approach to Finding Similar Code Segments,” *IEICE Trans. Inf. Syst.*, vol. E103.D, no. 4, pp. 874–878, Apr. 2020, doi: 10.1587/transinf.2019EDL8195.
- [205] A. Patnaik and N. Padhy, “A Hybrid Approach to Identify Code Smell Using Machine Learning Algorithms,” *Int. J. Open Source Softw. Process.*, vol. 12, no. 2, pp. 21–35, Apr. 2021, doi: 10.4018/IJOSSP.2021040102.
- [206] G. Mostaeen, B. Roy, C. K. Roy, K. Schneider, and J. Svajlenko, “A machine learning based framework for code clone validation,” *J. Syst. Softw.*, vol. 169, p. 110686, Nov. 2020, doi: 10.1016/j.jss.2020.110686.
- [207] W. Dong, Z. Feng, H. Wei, and H. Luo, “A Novel Code Stylometry-based Code Clone Detection Strategy,” in *2020 International Wireless Communications and Mobile Computing (IWCMC)*, Limassol, Cyprus: IEEE, Jun. 2020, pp. 1516–1521. doi: 10.1109/IWCMC48107.2020.9148302.
- [208] 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), Montreal, QC, Canada: IEEE, May 2019, pp. 783–794. doi: 10.1109/ICSE.2019.00086.
- [209] C. Xie, X. Wang, C. Qian, and M. Wang, “A Source Code Similarity Based on Siamese Neural Network,” *Appl. Sci.*, vol. 10, no. 21, p. 7519, Oct. 2020, doi: 10.3390/app10217519.
- [210] K. W. Nafi, B. Roy, C. K. Roy, and K. A. Schneider, “A universal cross language software similarity detector for open source software categorization,” *J. Syst. Softw.*, vol. 162, p. 110491, Apr. 2020, doi: 10.1016/j.jss.2019.110491.
- [211] A. Sheneamer, S. Roy, and J. Kalita, “An Effective Semantic Code Clone Detection Framework Using Pairwise Feature Fusion,” *IEEE Access*, vol. 9, pp. 84828–84844, 2021, doi: 10.1109/ACCESS.2021.3079156.
- [212] H. Liang and L. Ai, “AST-path Based Compare-Aggregate Network for Code Clone Detection,” in *2021 International Joint Conference on Neural Networks (IJCNN)*, Shenzhen, China: IEEE, Jul. 2021, pp. 1–8. doi: 10.1109/IJCNN52387.2021.9534099.
- [213] L. Chen, W. Ye, and S. Zhang, “Capturing source code semantics via tree-based convolution over API-enhanced AST,” in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, Alghero Italy: ACM, Apr. 2019, pp. 174–182. doi: 10.1145/3310273.3321560.
- [214] A. Sheneamer, “CCDLC Detection Framework-Combining Clustering with Deep Learning Classification for Semantic Clones,” in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, Orlando, FL: IEEE, Dec. 2018, pp. 701–706. doi: 10.1109/ICMLA.2018.00111.
- [215] Y. Zhang and T. Wang, “CCEyes: An Effective Tool for Code Clone Detection on Large-Scale Open Source Repositories,” in *2021 IEEE International Conference on Information Communication and Software Engineering (ICICSE)*, Chengdu, China: IEEE, Mar. 2021, pp. 61–70. doi: 10.1109/ICICSE52190.2021.9404141.
- [216] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, “CCLearner: A Deep Learning-Based Clone Detection Approach,” in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Shanghai: IEEE, Sep. 2017, pp. 249–260. doi: 10.1109/ICSME.2017.46.
- [217] F. Ullah, M. R. Naeem, L. Mostarda, and S. A. Shah, “Clone detection in 5G-enabled social IoT system using graph semantics and deep learning model,” *Int. J. Mach. Learn. Cybern.*, vol. 12, no. 11, pp. 3115–3127, Nov. 2021, doi: 10.1007/s13042-020-01246-9.
- [218] H. Xue, G. Venkataramani, and T. Lan, “Clone-Slicer: Detecting Domain Specific Binary Code Clones through Program Slicing,” in *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation - FEAST '18*, Toronto, Canada: ACM Press, 2018, pp. 27–33. doi: 10.1145/3273045.3273047.
- [219] H. Dişli and A. Tosun, “Code Clone Detection with Convolutional Neural Networks,” *Bilişim Teknol. Derg.*, vol. 13, no. 1, Art. no. 1, Jan. 2020, doi: 10.17671/gazibtd.541476.
- [220] Y. Wu and W. Wang, “Code Similarity Detection Based on Siamese Network,” in *2021 IEEE International Conference on Information Communication and Software Engineering (ICICSE)*, Chengdu, China: IEEE, Mar. 2021, pp. 47–51. doi: 10.1109/ICICSE52190.2021.9404110.

- [221] D. Perez and S. Chiba, “Cross-Language Clone Detection by Learning Over Abstract Syntax Trees,” in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, Montreal, QC, Canada: IEEE, May 2019, pp. 518–528. doi: 10.1109/MSR.2019.00078.
- [222] A. Hamdy and M. Tazy, “Deep Hybrid Features for Code Smells Detection,” *J. Theor. Appl. Inf. Technol.*, vol. 98, pp. 2684–2696, Jul. 2020.
- [223] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, “Deep learning similarities from different representations of source code,” in *Proceedings of the 15th International Conference on Mining Software Repositories*, Gothenburg Sweden: ACM, May 2018, pp. 542–553. doi: 10.1145/3196398.3196431.
- [224] G. Zhao and J. Huang, “DeepSim: deep learning code functional similarity,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Lake Buena Vista FL USA: ACM, Oct. 2018, pp. 141–151. doi: 10.1145/3236024.3236068.
- [225] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, “Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree,” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, London, ON, Canada: IEEE, Feb. 2020, pp. 261–271. doi: 10.1109/SANER48275.2020.9054857.
- [226] A. K. Das, S. Yadav, and S. Dhal, “Detecting Code Smells using Deep Learning,” in *TENCON 2019 - 2019 IEEE Region 10 Conference (TENCON)*, Kochi, India: IEEE, Oct. 2019, pp. 2081–2086. doi: 10.1109/TENCON.2019.8929628.
- [227] J. Zhang, H. Hong, Y. Zhang, Y. Wan, Y. Liu, and Y. Sui, “Disentangled Code Representation Learning for Multiple Programming Languages,” in *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, Online: Association for Computational Linguistics, 2021, pp. 4454–4466. doi: 10.18653/v1/2021.findings-acl.391.
- [228] D. K. Kim, “Enhancing code clone detection using control flow graphs,” *Int. J. Electr. Comput. Eng. IJECE*, vol. 9, no. 5, p. 3804, Oct. 2019, doi: 10.11591/ijece.v9i5.pp3804-3812.
- [229] A. Kaur and M. Saini, “Enhancing the Software Clone Detection in BigCloneBench: A Neural Network Approach,” *Int. J. Open Source Softw. Process. IJOSSP*, vol. 12, no. 3, pp. 17–31, Jul. 2021, doi: 10.4018/IJOSSP.2021070102.
- [230] J. Zeng, K. Ben, X. Li, and X. Zhang, “Fast Code Clone Detection Based on Weighted Recursive Autoencoders,” *IEEE Access*, vol. 7, pp. 125062–125078, 2019, doi: 10.1109/ACCESS.2019.2938825.
- [231] Y. Yuan, W. Kong, G. Hou, Y. Hu, M. Watanabe, and A. Fukuda, “From Local to Global Semantic Clone Detection,” in *2019 6th International Conference on Dependable Systems and Their Applications (DSA)*, Harbin, China: IEEE, Jan. 2020, pp. 13–24. doi: 10.1109/DSA.2019.00012.
- [232] A. Nair, A. Roy, and K. Meinke, “funcGNN: A Graph Neural Network Approach to Program Similarity,” in *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Bari Italy: ACM, Oct. 2020, pp. 1–11. doi: 10.1145/3382494.3410675.

- [233] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, “Functional code clone detection with syntax and semantics fusion learning,” in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Virtual Event USA: ACM, Jul. 2020, pp. 516–527. doi: 10.1145/3395363.3397362.
- [234] C. Wang *et al.*, “Go-clone: graph-embedding based clone detector for Golang,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Beijing China: ACM, Jul. 2019, pp. 374–377. doi: 10.1145/3293882.3338996.
- [235] N. D. Q. Bui, Y. Yu, and L. Jiang, “InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, Madrid, ES: IEEE, May 2021, pp. 1186–1197. doi: 10.1109/ICSE43902.2021.00109.
- [236] L. Buch and A. Andrzejak, “Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection,” in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Hangzhou, China: IEEE, Feb. 2019, pp. 95–104. doi: 10.1109/SANER.2019.8668039.
- [237] N. Mehrotra, N. Agarwal, P. Gupta, S. Anand, D. Lo, and R. Purandare, “Modeling Functional Similarity in Source Code with Graph-Based Siamese Networks,” *IEEE Trans. Softw. Eng.*, pp. 1–1, 2021, doi: 10.1109/TSE.2021.3105556.
- [238] W. Wang, G. Li, S. Shen, X. Xia, and Z. Jin, “Modular Tree Network for Source Code Representation Learning,” *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, pp. 1–23, Oct. 2020, doi: 10.1145/3409331.
- [239] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, “Neural Detection of Semantic Code Clones Via Tree-Based Convolution,” in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, Montreal, QC, Canada: IEEE, May 2019, pp. 70–80. doi: 10.1109/ICPC.2019.00021.
- [240] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, “Oreo: detection of clones in the twilight zone,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Lake Buena Vista FL USA: ACM, Oct. 2018, pp. 354–365. doi: 10.1145/3236024.3236026.
- [241] H.-H. Wei and M. Li, “Positive and Unlabeled Learning for Detecting Software Functional Clones with Adversarial Training,” in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, Stockholm, Sweden: International Joint Conferences on Artificial Intelligence Organization, Jul. 2018, pp. 2840–2846. doi: 10.24963/ijcai.2018/394.
- [242] Y. Wu *et al.*, “SCDetector: software functional clone detection based on semantic tokens analysis,” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, in ASE ’20. New York, NY, USA: Association for Computing Machinery, Dec. 2020, pp. 821–833. doi: 10.1145/3324884.3416562.
- [243] B. Li, C. Ye, S. Guan, and H. Zhou, “Semantic Code Clone Detection Via Event Embedding Tree and GAT Network,” in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, Macau, China: IEEE, Dec. 2020, pp. 382–393. doi: 10.1109/QRS51102.2020.00057.
- [244] C. Feng, T. Wang, Y. Yu, Y. Zhang, Y. Zhang, and H. Wang, “Sia-RAE: A Siamese Network based on Recursive AutoEncoder for Effective Clone Detection,” in *2020*

- 27th Asia-Pacific Software Engineering Conference (APSEC), Singapore, Singapore: IEEE, Dec. 2020, pp. 238–246. doi: 10.1109/APSEC51365.2020.00032.
- [245] L. Zhang, Z. Feng, W. Ren, and H. Luo, “Siamese-Based BiLSTM Network for Scratch Source Code Similarity Measuring,” in *2020 International Wireless Communications and Mobile Computing (IWCMC)*, Limassol, Cyprus: IEEE, Jun. 2020, pp. 1800–1805. doi: 10.1109/IWCMC48107.2020.9148382.
- [246] Y. Gao, Z. Wang, S. Liu, L. Yang, W. Sang, and Y. Cai, “TECCD: A Tree Embedding Approach for Code Clone Detection,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Cleveland, OH, USA: IEEE, Sep. 2019, pp. 145–156. doi: 10.1109/ICSME.2019.00025.
- [247] P. Ardimento, L. Aversano, M. L. Bernardi, M. Cimitile, and M. Iammarino, “Temporal convolutional networks for just-in-time design smells prediction using fine-grained software metrics,” *Neurocomputing*, vol. 463, pp. 454–471, Nov. 2021, doi: 10.1016/j.neucom.2021.08.010.
- [248] V. Saini *et al.*, “Towards Automating Precision Studies of Clone Detectors,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, Montreal, QC, Canada: IEEE, May 2019, pp. 49–59. doi: 10.1109/ICSE.2019.00023.
- [249] Y.-B. Jo, J. Lee, and C.-J. Yoo, “Two-Pass Technique for Clone Detection and Type Classification Using Tree-Based Convolution Neural Network,” *Appl. Sci.*, vol. 11, no. 14, Art. no. 14, Jan. 2021, doi: 10.3390/app11146613.
- [250] X. Ji, L. Liu, and J. Zhu, “Code Clone Detection with Hierarchical Attentive Graph Embedding,” *Int. J. Softw. Eng. Knowl. Eng.*, vol. 31, no. 06, pp. 837–861, Jun. 2021, doi: 10.1142/S021819402150025X.

Vitae

Name :Amal Abdulrahman Ali Alazba |

Nationality :Saudi |

Date of Birth :5/25/1994|

Email :aalazba@ksu.edu.sa|

Address :King Saud University, Riyadh 11362, Saudi Arabia|

Academic Background :Amal Alazba received her B.Sc. and M.Sc. degrees in information technology from King Saud University (KSU) in 2016 and 2018, respectively. She joined the Department of Information Systems as a teaching assistant at King Saud University in 2017. In September 2019, she embarked on a full-time PhD program at King Fahd University of Petroleum and Minerals, successfully completing her studies in December 2023. Amal has also achieved significant academic success, having published several conference and journal papers. Her current research interests encompass machine learning, deep learning, transformers, code smell detection, and empirical software engineering. |

Publications:

1. **Amal Alazba**, Hamoud Aljamaan, and Mohammad Alshayeb. "Deep learning approaches for bad smell detection: a systematic literature review." Empirical Software Engineering 28.3 (2023): 77.

2. **Amal Alazba**, and Hamoud Aljamaan. "Software defect prediction using stacking generalization of optimized tree-based ensembles." *Applied Sciences* 12.9 (2022): 4577.
3. **Amal Alazba**, Leina Abouhagar, Randah Al-Harbi, Hamdi A. Al-Jamimi, Abdullah Sultan, and Rabah A. Al-Zaidy. "Detection of Research Trends using Dynamic Topic Modeling." 2022 7th International Conference on Data Science and Machine Learning Applications (CDMA). IEEE, 2022.
4. **Amal Alazba**, and Hamoud Aljamaan. "Code smell detection using feature selection and stacking ensemble: An empirical investigation." *Information and Software Technology* 138 (2021): 106648.
5. Aljamaan, Hamoud, and **Amal Alazba**. "Software defect prediction using tree-based ensembles." *Proceedings of the 16th ACM international conference on predictive models and data analytics in software engineering*. 2020.
6. **Amal Alazba**, Nora S. Alturayef, Nouf Alturaief, and Zainab Alhathloul. "Saudi Stock Market Sentiment Analysis using Twitter Data." In *KDIR*, pp. 36-47. 2020.
7. **Amal Alazba**, Hend Al-Khalifa, and Hana AlSobayel. "RabbitRun: an immersive virtual reality game for promoting physical activities among people with low back pain." *Technologies* 7.1 (2018): 2.
8. **Amal Alazba**, Hend Al-Khalifa, and Hana AlSobayel. "A proposed game for promoting physical activities among people with low back pain using virtual reality." *Proceedings of the 11th Pervasive Technologies Related to Assistive Environments Conference*. 2018.

Submitted papers:

1. **Amal Alazba**, Hamoud Aljamaan, and Mohammad Alshayeb. "CoRT: Transformer-based Code Representations with Self-supervision by Predicting Reserved Words for Code Smell Detection." Empirical Software Engineering, *revised manuscript submitted*.
2. **Amal Alazba**, Hamoud Aljamaan, and Mohammad Alshayeb. "Automated Detection of Class Diagram Smells Using Self-Supervised Learning." Automated Software Engineering, *revised manuscript submitted*.