Journal of Software Engineering Research and Development, 2019, 6:1, doi: 10.5753/jserd.2019.xxx

Exploring Transfer Learning for Multilingual Software Quality: Code Smells, Bugs, and Harmful Code

Rodrigo Lima [**Universidade Federal de Pernambuco** | ***rsl@cin.ufpe.br***]
Jairo Souza [**Universidade Federal de Pernambuco** | ***jrmcs@cin.ufpe.br***]
Baldoino Fonseca [**Universidade Federal de Alagoas** | ***baldoino@ic.ufal.br***]
Leopoldo Teixeira [**Universidade Federal de Pernambuco** | ***lmt@cin.ufpe.br***]
Durval Pereira [**Universidade Federal de Alagoas** | ***durval@ic.ufal.br***]
Caio Barbosa [**Pontifícia Universidade Católica do Rio de Janeiro** | ***csilva@inf.puc-rio.br***]
Leonardo Leite [**Universidade Federal de Alagoas** | ***leo.leite@ic.ufal.br***]
Davy Baia [**Universidade Federal de Alagoas** | ***davy.baia@penedo.ufal.br***]

Abstract Code smells are indicators of poor design implementation and decision-making that can potentially harm the quality of software. Therefore, detecting these smells is crucial to prevent such issues. Some studies aim to comprehend the impact of Code smells on software quality, while others propose rules or machine learning-based approaches to identify code smells. Previous research has focused on labeling and analyzing code snippets that significantly impair software quality using machine learning techniques. These snippets are classified as Clean, Smelly, Buggy, and Harmful Code. Harmful Code refers to Smelly code segments that have one or more reported bugs, whether fixed or not. Consequently, the presence of a Harmful Code increases the risk of introducing new defects and/or design issues during the remediation process. We perform our study as an extension of the previous study, with the scope of 5 smell types, The total number of commits across all four tables (Java, C++, C#, and Python projects) is 641,736. versions of 91 open-source projects, 17,022 bugs and 24,737 code smells. The findings revealed promising transferability of knowledge between Java and C# in the presence of various code smell types, while C++ and Python exhibited more challenging transferability. Also, our study discovered that a sample size of 32 demonstrated favorable outcomes for most harmful codes, underscoring the efficiency of transfer learning even with limited data. Moreover, the exploration of transfer learning between bugs and code smells represents a not-very-ineffective avenue within the realm of software engineering.

Keywords: *code smells, software quality, machine learning, transfer learning*

1 Introduction

Modern software systems are increasingly developed using multiple programming languages, which introduces challenges in maintaining software quality. While code smells and bugs have been studied within individual languages, there is limited research on transferring knowledge about these issues across languages. This study aims to address this gap by exploring transfer learning as a solution, enabling models trained in one language to detect harmful code in another, thereby improving code quality in multilingual environments (Barbosa et al., 2020, 2023).

Code smells, indicators of poor design choices (Fowler, 1999; Roberta Arcoverde et al., 2012), and bugs introduced through changes (Śliwerski et al., 2005) can degrade software quality, making systems harder to maintain (Sharma and Spinellis, 2018). Identifying and mitigating these issues early can reduce the impact on software projects. Transfer learning, a machine learning technique that reuses knowledge from one domain to another (Amorim et al., 2016), offers the potential to automate this process, especially in multilingual software systems, by transferring learned knowledge between languages and contexts (Fontana Arcelli et al.).

Recent studies (Sharma et al., 2019; Kovacevic et al., 2022) have explored transfer learning in software engineering, particularly for code smell detection, but few have focused on detecting harmful code (Rodrigo Lima et al., 2020). This study aims to assess how transfer learning can detect harmful code across languages, building on prior work (Pereira Cesar et al., 2023; Rodrigo Lima et al., 2020). Harmful code is defined as code that is both smelly and buggy.

We collected source code from 91 open-source projects and evaluated five types of code smells: *Multifaceted Abstraction*, *Insufficient Modularization*, *Wide Hierarchy*, *Long Method*, and *Complex Method*, representing design and implementation issues. Our findings revealed promising transferability between Java and C#, while C++ and Python posed more challenges. Code smells were found to be less effective for bug detection in the context of transfer learning.

The paper is structured as follows: Section 3 reviews related work, Section 4 outlines the study design, Section 5 presents the results, Section 6 discusses threats to validity, and Section 7 concludes with future directions.

2 Background

In this section, we will briefly summarize concepts about important topics of our research.

2.1 Transfer Learning

Machine learning involves various techniques for sharing and adapting knowledge from one specific task in a domain to a broader task in the same domain. For instance, a healthcare provider employs predictive modeling to anticipate patient re-admissions, enabling early intervention and improving patient care outcomes. On the other hand, human beings demonstrate a unique capability, the ability to transfer knowledge across related domains to efficiently address novel challenges. This human-like approach becomes particularly advantageous when the new task shares fundamental similarities with the existing knowledge, enabling us to expedite problem-solving by leveraging our prior insights.

Transfer learning, at its core, involves transferring knowledge acquired in one source task to enhance learning in a related target task (Torrey and Shavlik, 2010). One of the major advantages of transfer learning is that it is valuable in scenarios characterized by a scarcity of training data. When collecting ample training data for a target task proves challenging and resource-intensive, we can identify a source task with similar underlying characteristics and access to a vast training dataset. Subsequently, we train a machine learning model on this source task, utilizing the abundant dataset, and then fine-tune the model on the target task, leveraging the available yet limited training data. This strategic process empowers us to harness prior knowledge effectively, significantly improving performance on the target task. For example, if we initially trained a model to classify animals and later wished to classify specific breeds of dogs, we could apply transfer learning by building upon the animal classification knowledge.

2.2 Code Smells

Code smells are indicators of quality issues that can affect software maintainability and scalability (Oizumi et al., 2015, 2014). They are categorized into implementation smells, design smells, and architecture smells, based on scope and impact (Oizumi et al., 2016, 2018; Sharma and Spinellis, 2018). Implementation smells, such as long method, complex method, and magic numbers, typically affect individual methods (Fowler, 1999), while design smells, like God class and multifaceted abstraction, involve larger abstractions, such as classes or class groups.

These smells manifest across languages like Java, C#, C++, and Python. In Java, common smells include Multifaceted Abstraction and Long Method, which degrade maintainability by combining unrelated functionalities or creating overly complex methods (Sousa et al., 2017). Similar issues arise in C#, where poor namespace organization leads to Insufficient Modularization, and in C++, where mixing low-level memory management with high-level operations is a frequent design smell. Python, although dynamically typed, suffers from code smells like large monolithic scripts and complex functions, which impact readability and maintenance.

3 Related Work

Transfer Learning of Code Smells. Previous studies have explored the potential of transfer learning for detecting code smells. Sharma *et al.*(Sharma et al., 2019) applied deep learning to detect code smells and demonstrated the feasibility of transfer learning for identifying smells in languages where specialized detection tools are unavailable. Kovacevic *et al.*(Kovacevic et al., 2022) extended this work by transferring knowledge between languages using manually labeled datasets, focusing on the smells *Long Method* and *God Class*. While both works explored transfer learning for code smell detection, our study goes further by assessing harmfulness—using metrics to identify if a code snippet is both smelly and buggy.

Code Smells and Bugs. Several studies have examined the relationship between code smells and bugs. Takahashi *et al.*(Takahasi et al., 2018) improved bug localization by integrating code smells into the bug detection process, while Palomba *et al.*(Palomba et al., 2019) developed a smell-aware bug prediction model. Lima *et al.* (Rodrigo Lima et al., 2020) introduced the concept of *harmful code*, identifying code that is both smelly and buggy. Their work found that Random Forests were effective for classifying harmful code in Java projects, but transfer learning for detecting harmful code remains largely unexplored.

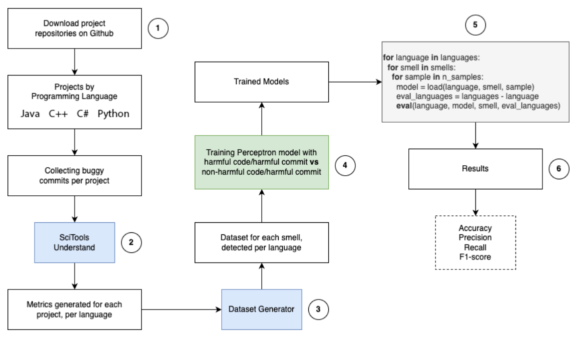
Our study combines these approaches, using transfer learning not only for detecting code smells but also for assessing code harmfulness across multiple programming languages. Unlike previous work, which focused on individual smells or bugs, we evaluate transfer learning's ability to generalize across languages and provide insights at the commit level, considering the historical aspects of software development.

This version is more succinct, addressing the key points from contemporary studies while framing the unique contributions of your research.

4 Study Design

Recent studies (Sharma et al., 2019; Kovacevic et al., 2022; Krishna and Menzies, 2019; Ardimento and et al; De Stefano et al.) have demonstrated the potential of transfer learning in software engineering, particularly for code smell detection and prediction. However, none have focused on applying knowledge from other systems to detect harmful code. To address this gap, our study explores transfer learning for detecting harmful code across five code smells and three programming languages. Our research aims to provide insights into the effectiveness of transfer learning in improving code analysis and software quality assurance, as outlined in the following research questions:

RQ$_1$: How effective is transfer learning in detecting harmful code? This question evaluates transfer learning's effectiveness (measured by the F-measure) in detecting harmful code across three programming languages. The goal is to assess whether transfer learning can generalize harmful code detection in a language-agnostic way, providing valuable guidance for improving software security and reliability.


Download project repositories on Github
1
Projects by Programming Language
Java C++ C# Python
Collecting buggy commits per project
SciTools Understand
2
Metrics generated for each project, per language
Dataset Generator
3
Dataset for each smell, detected per language
Training Perceptron model with harmful code/harmful commit **vs** non-harmful code/harmful commit
4
Trained Models
5
Results
6
Accuracy
Precision
Recall
F1-score

```
for language in languages:
  for smell in smells:
    for sample in n_samples:
      model = load(language, smell, sample)
      eval_languages = languages - language
      eval(language, model, smell, eval_languages)
```

Figure 1. Study design steps for creating dataset and evaluating transfer learning

RQ$_2$: How efficient is transfer learning in detecting code smells? We explore the impact of varying sample sizes on transfer learning performance, evaluating its effectiveness in detecting code smells across languages. This analysis covers dataset sizes from small (2) to large (630), addressing the trade-offs between model generalization and computational efficiency.

RQ$_3$: How effective is transfer learning in detecting bugs from code smells? This question investigates how well transfer learning can identify software bugs originating from code smells, using an oracle to identify code smells across projects. Precision, Recall, and F-measure metrics are used to quantify how effectively transfer learning detects bugs caused by existing code smells.

RQ$_4$: How effective is transfer learning in detecting code smells from bugs? We reverse the relationship, exploring how well transfer learning detects code smells using bug-related data from three languages. By analyzing this inverse relationship, we aim to enhance bug detection processes and contribute to more comprehensive software quality improvement strategies.

4.1 Programming Languages

We selected Java, C#, C++, and Python for this study due to their widespread use and significance in various domains. Java is prominent in enterprise applications, C# in the Microsoft ecosystem and game development, C++ in-system programming and performance-critical tasks, and Python in web development, data science, and AI. These languages represent a diverse set of programming paradigms, each with unique features and conventions that can influence the effectiveness of transfer learning for code analysis. By including them, we aim to evaluate how transfer learning performs across different contexts, offering insights that can guide the development of more adaptable code analysis tools and improve software quality across multiple programming environments.

4.2 Project Selection

In order to avoid well-known mining perils (Kalliamvakou et al., 2016), we applied the following methodology to select the projects for this study: (i) systems that have at least 500 commits; (ii) systems that are at least 3 years old, and are currently active; and (iii) Java, C++, C# and Python based systems, as previously mentioned in 4.1. We can see the project

Table 1. Java Software projects analyzed in this work

System	Domain	# Commits	# Smells	# Bugs
exoplayer	Library	12393	134	71
baritone	Library	2796	2025	2863
junit5	Framework	7076	0	2
lombok	Library	3325	1530	1346
mindustry	Game	12854	20	39
mockito	Framework	5695	14	73
okhttp	Http Client	4958	128	62
termux-app	App	1179	28	10
bisq	App	15809	661	781
l2jorg	Library	2228	238	260
IPED	Forensic	4883	127	83
PhotoView	Library	462	11	6
hudi	Big Data	2272	192	148
AgentWeb	Web	1022	47	32
Signal-Server	Communication	2147	312	197
archiva	Build Management	8701	389	254
tomcat	Server	23824	493	308
TelegramBots	Library	941	18	12
gson	Library	1611	13	9
copybara	Tool	2503	108	74
poi	Library	11576	314	194
skywalking	Monitoring	6872	263	178
caffeine	Cache	1367	23	16
airbyte	Data Integration	4303	153	104
netty	Networking	10673	396	247
moshi	Library	968	12	7
Java-WebSocket	Library	1093	31	21
h2database	Database	13710	507	342
druid	Database	11461	423	291
zxing	Library	3623	52	37

list in Tables 1, 2 3. Each table represents the projects selected to be analyzed in our study for the programming languages Java, C++, C#, and Python respectively. The tables not only lists the projects names but also describes the domain of it - if it is a library, a framework, an app - as well as the number of: i) commits; ii) smells and iii) bugs for each one of the projects.

4.3 Metrics and Code Smells

We utilized detection rules from the DesigniteJava [1] tool (Sharma and Spinellis, 2018) to identify code smells in our projects, along with their corresponding thresholds. However, since we are collecting data from projects written in C++, C#, and Python languages, we cannot directly use the tool, as it only detects smells in Java code. In this case, we used the Understand [2] tool to collect the metrics in all systems. Moreover, we needed to make a pair relation between the DesigniteJava metrics names and the Understand metrics, as they do not have the same name. Finally, the code smell list and their respective thresholds can be seen in Table 6, and the metrics name pair relation can be seen in Table 5.

4.4 Finding Bugs

We rely on previous work (Rodrigo Lima et al., 2020) methodology to collect the bugs used in our dataset. This methodology utilizes a GitHub macro present in commit messages that fix bugs. These macros typically include keywords such as ”Fixes”, ”Fixed”, ”Fix”, ”Closes”, ”Closed” or ”Close”, followed by a # and the issue/pull request number, *e.g.,* ”Fixes #12345”. This macro automatically clos-

[1]https://www.designite-tools.com/designitejava/
[2]https://scitools.com/

Table 2. C++ Software projects analyzed in this work

System	Domain	# Commits	# Smells	# Bugs
gdal	Library	46604	3446	312
keepassxc	Software	4192	113	83
osquery	Framework	6019	17	12
tdesktop	Software	11296	181	59
px4-autopilot	Framework	39499	187	26
qtox	Software	7945	115	64
bitcoin	Cryptocurrency	32033	432	273
faiss	Library	585	7	5
ImHex	Software	1037	14	9
imgui	Library	6719	52	35
libzmq	Networking	8318	126	82
terminal	Software	2704	39	22
folly	Library	10283	145	98
server	Server	2374	31	19
ethminer	Cryptocurrency	14334	89	54
gitahead	Tool	380	5	3
Hazel	Software	243	4	2
flameshot	Software	1598	21	14
xmrig	Cryptocurrency	3120	43	29
monero	Cryptocurrency	10701	148	97

Table 3. C# Software projects analyzed in this work

System	Domain	# Commits	# Smells	# Bugs
efcore	Library	12451	1923	1484
humanizer	Library	2236	12	10
jellyfin	Software	22217	518	292
omnisharp-roslyn	Http Server	5738	0	2
quicklook	Software	806	12	15
neo	Blockchain	1330	159	277
ryujinx	Software	1915	45	38
OrchardCore	CMS	5729	150	75
eShopOnContainers	Microservices	3937	200	125
Files	App	3076	180	90
ModAssistant	Library	669	20	10
SharpZipLib	Library	980	30	18
pinvoke	Library	1629	40	22
EnhancePoEApp	App	409	15	8
TwitchLeecher	App	322	12	7
privatezilla	Security	307	10	6
StockSharp	Trading	7726	280	150
msgpack-cli	Library	3304	110	65
reverse-proxy	Networking	665	20	12
runner	Tool	454	25	14

Table 4. Python Software projects analyzed in this work

System	Domain	# Commits	# Smells	# Bugs
Tensorflow	Library	14104	1923	1484
FastAPI	Library	2319	12	10
Flask	Software	4495	518	292
Tornado	Library	6113	0	2
Pyramid	Library	838	12	15
Dash	Library	1404	159	277
Vibora	Software	2840	45	38
django	Framework	30320	1120	630
backtrader	Trading	2385	80	55
ansible	Automation	52191	1350	890
antlr4	Parsing	8269	210	140
cltk	NLP	3592	90	60
pygame	Game Development	7230	320	210
SerpentAI	AI	250	5	3
electrum	Cryptocurrency	14049	450	300
EasyOCR	OCR	474	15	10
PyBoy	Emulator	435	10	6
renpy	Game Engine	11537	380	240
openage	Game Engine	4047	160	110
pgadmin4	Database Tool	5225	190	130
faceswap	Machine Learning	1410	50	35

Table 5. Understand Software Metrics

Name	Abbrev.	*SciTools Understand*	Granularity
Lack of Cohesion in Methods	LCOM	PercentLackOfCohesion	Class
Number of Fields	NOF	CountDecClassVariable + CountDeclInstanceVariable	Class
Number of Methods	NOM	CountDeclMethod	Class
Number of Public Methods	NOPM	CountDeclMethodPublic	Class
Weighted Methods per Class	WMPC	SumCyclomaticModified	Class
Number of Children	NC	CountClassDerived	Class
Lines of Code	LOC	CountLine	Class/Method
Cyclomatic Complexity	CC	Cyclomatic	Method
Strict Cyclomatic Complexity	SC	SumCyclomaticStrict	Method
Modified Cyclomatic Complexity	MCC	SumCyclomaticModified	Method
Derived Classes	DC	CountClassDerived	Class
Lines of Code (Code Only)	LOC-Code	CountLineCode	Class/Method
Class Coupling (Modified)	CCM	CountClassCoupledModified	Class
Executable Statements	ES	CountStmtExe	Method

Table 6. Detection Rules for the Code Smells

Name	Granularity	Type	Metric	Logical Op.
Multifaceted Abstraction	Class	Design	LCOM >= 0.8 NOF >= 7 NOM >= 7	AND
Insufficient Modularization	Class	Design	NOPM >= 20 NOM >= 30 WMC >= 100	OR
Wide Hierarchy	Class	Design	NC >= 10	N/A
Long Method	Method	Implementation	LOC >= 100	N/A
Complex Method	Method	Implementation	CC >= 8	N/A

es/merges the issue/pull request, and by examining their label list, we can check if that issue/pull request contains the label 'bug' or 'defect', showing that the commit was a bug-fixing commit. Finally, to collect the commit that contains the bug, we got the parent commit of the bug-fixing commit.

4.5 Model Selection and Configuration Rationale

In this study, we opted to experiment with the perceptron, a simple neural network architecture, for its simplicity, interpretability, and computational efficiency. This design allowed us to focus on the core mechanics of transfer learning, avoiding complexities of deeper models, and enabled faster training and evaluation across languages and code smells. The perceptron served as a baseline, demonstrating transfer learning potential, with future work planned to explore advanced models. We used well-known configurations to ensure reproducibility and comparability with existing studies. A Github repository [3] is provided for further experimentation.

4.6 Discovering Harmful Code

Harmful code is a term introduced by Lima *et al.* (Rodrigo Lima et al., 2020) to determine a code snippet has two characteristics: (i) *smelly*, when the code contains a code smell; and (ii) *buggy*, when the code contains a bug. When containing both characteristics, we say that a code snippet is *harmful*.

4.6.1 Application of Transfer Learning

We trained our transfer learning models using the dataset generated in previous steps, following the pseudo-code in Fig. 1. The model, based on a perceptron architecture (Kanal, 2003),

[3] https://github.com/harmful-code/jserd_harmful_transfer

was designed for simplicity and efficiency, employing key parameters as follows:

Embedding layer. Configured to process input sequences of integers (tokens) with a vocabulary of 20,000 words. Each token is embedded into an 8-dimensional vector, and the input_length specifies the sequence length.

Flatten layer. Converts the 2D output from the embedding layer into a 1D array.

Dense layer. A fully connected layer with a single neuron using a 'sigmoid' activation function for binary classification.

Compilation parameters. The model uses the 'adam' optimizer with a learning rate of 0.001 and 'binary_crossentropy' as the loss function, tailored for binary classification tasks.

Training Configuration. The perceptron model was trained for 100 epochs with accuracy as the evaluation metric. No regularization techniques were applied, providing a balance between speed and stability.

CNN Configuration. We also trained a CNN model with the same configuration, using 'rmsprop' as the optimizer with a learning rate of 0.001. Both models were designed for efficiency and scalability across multiple languages and code smells.

.

```
for language in languages:
    for smell in smells:
        for sample in n_sample:
            model = load(language, smell, sample)
            model = Sequential()
            model.add(Embedding(20000, 8,
                input_length=padding))
            model.add(Flatten())
            model.add(Dense(1, activation='sigmoid'))
            model.compile(optimizer='adam',
                loss='binary_crossentropy')
            eval_languages = languages - language
            eval(language, model, smell, eval_languages)
```

4.6.2 Evaluation

To evaluate the models, we first need to compute the results of TP, TN, FP, and FN, that are described as follows:

- **TP**: True Positive, when the model correctly predicts the "YES" target class.
- **TN**: True Negative, when the model correctly predicts the "NO" target class.
- **FP**: False Positive, when the model incorrectly predicts the target class as "YES" when it should be "NO".
- **FN**: False Negative, when the model incorrectly predicts the target class as "NO" when it should be "YES".

Then, we are able to calculate the metrics that are the output of our model:

- **Accuracy**: The rate of correct classification by the model (as either a smell or not).

$$A = (TP + TN)/(TP + TN + FP + FN)$$

- **Precision**: Of the samples classified by the model as smells, how many were actually smells.

$$P = TP/(TP + FP)$$

- **Recall**: The proportion of correctly classified smells out of the total number of samples that were actually smells.

$$R = TP/(TP + FN)$$

- **F1-score**: The harmonic mean of precision and Recall.

$$F = 2 * [(P * R)/(P + R)]$$

5 Results and Discussion

In this section, we present the key findings and main results derived from our study.

5.1 Effective Transfer Learning to Detect Harmful Code

Table 7. Transfer learning of harmful code trained in C++, C#, and Java and tested in C++, C#, Python, and Java.

		Java	C#	C++	Python
Complex Method	Java	89%	82%	92%	80%
	C#	84%	82%	75%	75%
	C++	97%	75%	97%	91%
	Python	76%	75%	91%	81%
Long Method	Java	79%	75%	92%	80%
	C#	75%	82%	87%	80%
	C++	74%	82%	92%	80%
	Python	75%	80%	80%	80%
Wide Hierarchy	Java	10%	82%	10%	10%
	C#	5%	82%	5%	5%
	C++	0%	82%	100%	5%
	Python	5%	5%	5%	5%
Insufficient Modularization	Java	84%	82%	92%	80%
	C#	84%	82%	87%	80%
	C++	99%	82%	99%	92%
	Python	80%	80%	92%	80%
Multifaceted Abstraction	Java	83%	82%	92%	80%
	C#	79%	82%	87%	80%
	C++	98%	82%	98%	92%
	Python	80%	80%	92%	80%

This research question evaluates the effectiveness of transfer learning using the Perceptron (Kanal, 2003) model for detecting harmful code. Our approach involves training individual models for each programming language with buggy commits and distinct code smell types using dedicated training datasets that consist of relevant code snippets and corresponding smells in buggy commits. Subsequently, we check the performance of each trained model is meticulously evaluated on testing datasets containing code snippets from all programming languages studied to understand if harmful code from one language can be detected in the others.

The study enables us to identify the model's strengths and limitations in different contexts, offering a nuanced understanding of its performance in detecting harmful code beyond the initial training dataset. The resulting insights are presented in Table 7, where the first and second columns highlight the smell types involved in the buggy snippet and programming languages used for training, while the horizontal arrangement corresponds to the programming languages in the testing datasets.

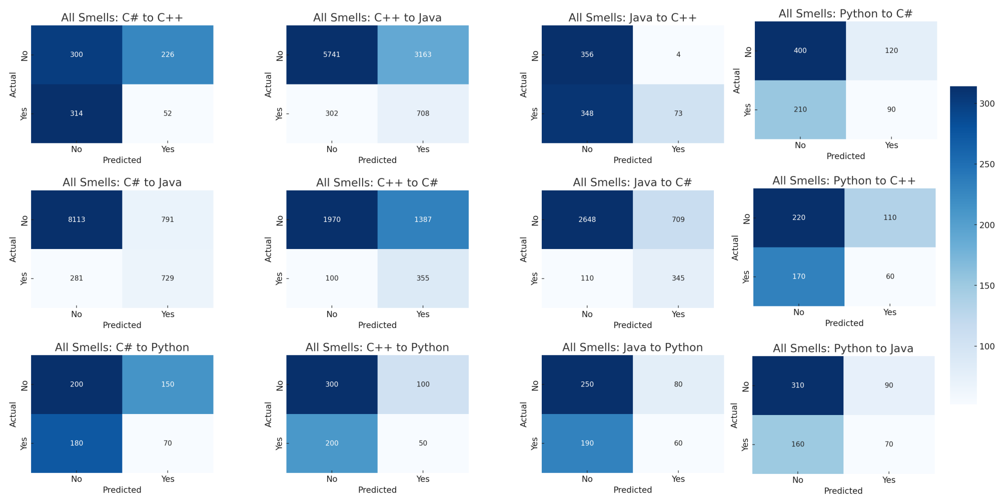


Figure 2. Harmful Code results for the transfer learning combined for all code smells and divided by each language.

Figure 2 presents the results of our transfer learning model for harmful code detection. The confusion matrices provide a detailed breakdown of the model's performance across different programming languages and all code smells analyzed combined. Moreover, we also have a confusion matrix for each smell type for each language type, those can be seen in our replication package [4].

By analyzing the confusion matrices, we gained valuable insights into the model's strengths and limitations, identifying areas where the model excelled and areas that could benefit from further refinement. These results contribute to a better understanding of the transfer learning model's effectiveness in detecting code smells and lay the groundwork for future research and improvements in code analysis and software quality assurance.

5.1.1 Harmful Code between Java and C#

In Table 7, the analysis highlights interesting observations concerning transfer learning between Java and C# programming languages. We notice a noteworthy variation in effectiveness, ranging from 5% in the Wide Hierarchy smell to a maximum of 84% in the Complex Method. This variability indicates that different types of smells exhibit distinct levels of transferability between these languages. Specifically, the Complex Method demonstrates promising results, with a respectable accuracy of 84% when evaluated within Java itself and an equivalent 84% when applied to C#. This suggests that the rule set for this smell translates effectively between the two languages.

Furthermore, examining the Long Method and Insufficient Modularization, we observe a similar pattern, with minor differences in effectiveness, ranging from 2% to 9%. These findings imply that certain code smells have relatively consistent transferability between Java and C#, while others may necessitate more targeted adjustments for optimal cross-language detection. These insights elucidate us on the intricate relationship between different smells and their transferability across programming languages. Understanding such nuances is crucial for devising more effective and versatile transfer learning approaches in code analysis, ultimately improving software quality and maintainability across diverse language ecosystems.

Finally, the outcomes for the Wide Hierarchy and Multifaceted Abstraction smells were less promising. We obtained an accuracy of 5% for Wide Hierarchy detection between the two languages and encountered a difference of 4% in transferring the knowledge of how to detect the Multifaceted Abstraction smell to the other language. These results lead us to the conclusion that not all code smells are equally effective in detecting harmful code between these two languages.

Finding 1: The smells Complex Method, Long Method, and Insufficient Modularization demonstrate a high level of effectiveness in transferring knowledge between C# and Java.

The varying degrees of transferability indicate that some smells may not generalize well across different language contexts, underscoring the importance of carefully considering the choice of code smells and their applicability when employing transfer learning techniques for code analysis in such scenarios.

5.1.2 C++ and Python: Low Knowledge Transferability

In the context of the C++ and Python languages, the observed poor transferability across all smell types suggests that the application of transfer learning techniques for harmful code detection faces significant challenges in these languages. The relatively low scores highlight the difficulty in effectively adapting harmful code detection rules between C++/Python and other programming languages.

For the Complex Method, the model achieved excellent results when training with C++ and applying to Java (97%), but not as well when applied to C# (75%), compared with the 97% when applied to itself. Similarly, training with Python and applying to Java yielded a moderate score of 76%, but performance remained consistent when applied to C# (75%). Moreover, in smell types Long Method, Multifaceted Abstraction, and Insufficient Modularization, both C++ and Python showed scores around 80%, which emphasizes the

[4] https://github.com/harmful-code/jserd_harmful_transfer

Table 8. F-Measure of Transfer Learning trained with multiple sample sizes

		Samples = 4				Samples = 8				Samples = 16				Samples = 32			
		Java	C#	C++	Python	Java	C#	C++	Python	Java	C#	C++	Python	Java	C#	C++	Python
Complex Method	Java	50%	68%	84%	50%	79%	76%	84%	79%	82%	84%	92%	74%	89%	84%	92%	81%
	C#	54%	75%	93%	54%	79%	76%	84%	75%	93%	75%	92%	77%	75%	84%	92%	81%
	C++	55%	93%	75%	55%	93%	84%	84%	75%	93%	75%	92%	75%	93%	84%	92%	81%
	Python	54%	75%	92%	54%	79%	75%	84%	79%	92%	75%	92%	75%	92%	84%	92%	81%
Long Method	Java	50%	68%	84%	50%	79%	76%	84%	79%	82%	84%	92%	74%	89%	84%	92%	81%
	C#	54%	75%	93%	54%	79%	76%	84%	75%	93%	75%	92%	77%	75%	84%	92%	81%
	C++	55%	93%	75%	55%	93%	84%	84%	75%	93%	75%	92%	75%	93%	84%	92%	81%
	Python	54%	75%	92%	54%	79%	75%	84%	79%	92%	75%	92%	75%	92%	84%	92%	81%
Wide Hierarchy	Java	0%	1%	1%	1%	3%	1%	0%	3%	2%	1%	2%	0%	3%	2%	1%	2%
	C#	0%	1%	1%	1%	1%	1%	1%	1%	0%	1%	1%	0%	1%	1%	1%	1%
	C++	0%	1%	1%	1%	1%	1%	0%	1%	1%	1%	0%	1%	1%	1%	1%	1%
	Python	0%	1%	1%	1%	1%	1%	1%	1%	0%	1%	1%	0%	1%	1%	1%	1%
Insufficient Modularization	Java	30%	54%	30%	54%	54%	93%	75%	54%	93%	75%	92%	75%	93%	93%	92%	93%
	C#	50%	75%	93%	54%	79%	75%	84%	75%	93%	75%	92%	77%	75%	84%	92%	81%
	C++	55%	93%	75%	55%	93%	84%	84%	75%	93%	75%	92%	75%	93%	84%	92%	81%
	Python	54%	75%	92%	54%	79%	75%	84%	79%	92%	75%	92%	75%	92%	84%	92%	81%
Multifaceted Abstraction	Java	32%	2%	-	56%	23%	-	62%	37%	7%	61%	40%	-	61%	40%	-	40%
	C#	19%	14%	20%	25%	26%	23%	42%	34%	29%	46%	40%	25%	46%	40%	25%	40%
	C++	19%	34%	7%	32%	23%	83%	31%	25%	77%	33%	22%	88%	33%	22%	88%	22%
	Python	19%	34%	7%	32%	23%	83%	31%	25%	77%	33%	22%	88%	33%	22%	88%	22%

complexity of these smells in C++ and Python code. These findings indicate that the underlying structures and coding practices in C++ and Python present unique nuances that hinder the straightforward transfer of knowledge learned from other languages.

The results underscore the importance of considering language-specific characteristics when applying transfer learning techniques in code analysis. As C++ is known for its intricacies and versatility, and Python for its dynamic and flexible nature, both may require tailored approaches and specialized models to achieve more accurate and effective harmful code detection.

Finding 2: C++ and Python show varying knowledge transferability across languages, with Complex Method achieving up to 97% accuracy when trained and tested within the same language.

Further research and exploration of domain-specific features and transfer learning strategies can aid in improving the transferability of knowledge across programming languages, ultimately enhancing the overall performance of harmful code transfer learning approaches in the context of C++ and Python.

5.1.3 Important Features in Transfer Learning for Harmful Code

Table 9. features in transfer learning of harmful code smells trained in different languages and tested in others.

	C#	**C++**	**Python**	**Java**
C#		SC (100.00%)	SC (100.00%)	SC (100.00%)
C++	MCC (99%) DC (1%)		LOC-Code (100.00%)	LOC-Code (100.00%)
Python	MCC (100%)	MCC (92.31%) CCM (7.69%)		ES (100.00%)
Java	SC (100%)	MCC (91.30%) DC (8.70%)	MCC (100.00%)	

Table 9 highlights the most influential metrics for detecting harmful code across different languages. The metrics reflect how transfer learning models trained in one language perform when applied to detect harmful code in others.

A key finding is the importance of the *SumCyclomaticStrict (SC)* metric, which consistently plays a crucial role across C#, C++, Python, and Java. This metric captures structural complexity, a central factor in identifying harmful code across languages.

Notably, in C++ and Python, *SumCyclomaticModified (MCC)* and *CountClassCoupledModified (CCM)* stand out. MCC and *CountClassDerived (DC)* together account for 99% of importance in C++, highlighting the influence of code complexity and inheritance. Similarly, MCC (92.31%) and CCM (7.69%) underscore the relevance of cyclomatic complexity and class coupling in Python, reflecting its object-oriented nature.

The *Lines of Code (LOC)* metric is significant when models trained in C++ are applied to Java, and vice versa, as larger codebases are more prone to bugs. Python's *Executable Statements (ES)* metric accounts for 100% importance, reflecting its reliance on concise, dynamic code compared to Java's more static nature.

In conclusion, cyclomatic complexity metrics remain central across languages, but language-specific features, such as class coupling in Python and inheritance in C++, require specialized approaches for effective transfer learning. Tailoring models to these features can enhance harmful code detection.

Finding 3: Cyclomatic complexity metrics are fundamental to harmful code detection across languages, while language-specific features like class coupling (Python) and inheritance (C++) demand specialized approaches for effective transfer learning.

5.2 Efficient Transfer Learning to Detect Harmful Code

5.2.1 Java High Transferability with Small Samples

Tables 8 and 10 show that for most smells, 32 or 64 samples are sufficient, with minimal improvements beyond 128 samples.

For Complex Method, Java-to-C# performs best with 16 samples, while Java-to-C++ requires 128. In Long Method, Java-to-C# peaks with 32 samples, and Java-to-C++ with 64.

Table 10. F-Measure of Transfer Learning trained with multiple sample sizes

		Samples = 32			**Samples = 64**			**Samples = 128**			**Samples = 256**		
		Java	C#	C++	Python	Java	C#	C++	Python	Java	C#	C++	Python
Complex Method	Java	77%	59%	10%	75%	82%	56%	32%	80%	75%	54%	33%	
	C#	79%	84%	14%	75%	84%	75%	60%	12%	75%	67%	3%	
	C++	74%	54%	97%	75%	92%	75%	47%	90%	66%	47%	94%	
	Python	80%	82%	75%	75%	82%	82%	75%	75%	75%	80%	75%	
Long Method	Java	71%	58%	-	71%	56%	58%	73%	57%	3%	69%	57%	3%
	C#	65%	56%	-	63%	60%	-	70%	66%	-	64%	60%	-
	C++	28%	34%	75%	35%	39%	84%	35%	38%	92%	36%	41%	90%
	Python	80%	75%	75%	-	-	75%	80%	75%	75%	80%	80%	80%
Wide Hierarchy	Java	12%	-	-	-	-	-	-	-	-	-	-	-
	C#	0%	0%	1%	0%	-	1%	-	0%	1%	0%	1%	1%
	C++	0%	1%	100%	-	-	100%	-	-	100%	-	-	-
	Python	5%	5%	5%	5%	5%	5%	5%	5%	5%	5%	5%	5%
Insufficient Modularization	Java	63%	72%	13%	76%	67%	68%	68%	73%	68%	76%	67%	68%
	C#	59%	73%	10%	65%	67%	-	53%	74%	22%	60%	76%	27%
	C++	23%	47%	92%	25%	48%	95%	29%	52%	95%	34%	63%	96%
	Python	92%	92%	92%	92%	92%	92%	92%	92%	92%	92%	92%	92%
Multifaceted Abstraction	Java	61%	40%	-	64%	40%	2%	70%	40%	2%	76%	40%	2%
	C#	46%	40%	25%	50%	44%	4%	49%	44%	8%	50%	50%	20%
	C++	33%	22%	88%	30%	23%	88%	36%	24%	92%	41%	29%	95%
	Python	91%	91%	98%	91%	91%	91%	91%	91%	91%	91%	91%	91%

For Wide Hierarchy, 8 samples suffice for Java-to-C#. Insufficient Modularization reaches near-optimal results with 32 samples for Java-to-C# and 16 for Java-to-C++. For Multifaceted Abstraction, Java-to-C# performs best with 64 samples, while Java-to-C++ yields a sub-optimal result with 8 samples.

These results highlight the efficiency of transfer learning in Java, where 32 or 64 samples often suffice for effective harmful code detection, helping improve software quality and maintainability.

Finding 4: For most harmful code trained in Java, 32 or 64 samples are sufficient for effective transfer learning, with minimal gains beyond 128 samples.

This finding shows that developers can achieve effective results with moderate sample sizes, suggesting that larger datasets may not significantly enhance accuracy, thus optimizing resource allocation in code detection.

5.2.2 Harmful Code Between C# and C++

The results presented in Tables 8 and 10 provide compelling evidence of successful knowledge transfer between the C++ and C# languages for most code smells, demonstrating the potential for cross-language applicability in code smell detection.

An intriguing observation from the results is that a sample size of 32 already yields promising outcomes for most of the analyzed smells. This highlights the efficiency of transfer learning even with a relatively modest amount of data, which can be advantageous when dealing with limited resources or large-scale software projects. Nevertheless, to achieve optimal performance, larger sample sizes are necessary, as evidenced by the Multifaceted Abstraction smell, where a sample size of 512 was required to achieve favorable results. This discrepancy in sample size requirements underlines the importance of tailoring the transfer learning approach based on the specific code smell and language combination, ensuring more accurate and efficient detection.

Finding 5: Effective knowledge transfer between most code smells in C++ and C# languages using transfer learning, and optimal performance often necessitates larger sample sizes, particularly observed in the case of Multifaceted Abstraction.

These findings have significant implications for practitioners and researchers in software engineering. Understanding the transferability of knowledge between programming languages can guide the development of more effective code analysis tools that can be applied across diverse language ecosystems. Moreover, the insights on sample size requirements shed light on the trade-offs between computational resources and detection accuracy, aiding in the optimization of transfer learning techniques for code smell detection in real-world software projects. Overall, this study contributes valuable knowledge towards advancing transfer learning applications in software engineering and promoting better code quality and maintainability.

5.3 Effective Transfer Learning for Detecting Bugs

Our goal was to evaluate the effectiveness of transfer learning for detecting bugs using the Perceptron model. Separate models were trained for each language (Java, C#, C++) using code snippets with code smells from commits. We aimed to see if models trained in one language could detect bugs in another. Table 12 shows that all models performed poorly, but C++ had the highest accuracy, precision, recall, and F1 score, suggesting better suitability for this task compared to Java and C#.

Table 11 further highlights the F-measure across languages. C++ achieved the highest F-measure (44%), while C# had the lowest (8%). Java scored in between (9%). These results, though suboptimal, indicate that code smells are not effective for detecting bugs across languages. Models with '-' in the table failed to run due to data correlations above 0.75.

Table 11. Results for Model Trained with Code Smells and Tested with Bugs

	Java	C#	C++
Java	9%	26%	-
C#	8%	21%	-
C++	26%	44%	-

Table 12. Accuracy, precision, recall, and F-measure for the bug correctly classified by the prediction model trained with smells

Model Language	Accuracy	Precision	Recall	F1
Java	0.195	0.241	0.283	0.260
C#	0.187	0.204	0.216	0.210
C++	0.291	0.364	0.558	0.440

Finding 1: Code smells are not suited, in the context of transfer learning models, for the detection of bugs.

Figure 3 presents the results of our transfer learning model for bug detection. The confusion matrices provide a detailed breakdown of the model's performance across different programming languages. In general, we can see in the matrices that the number of FP and FN are bigger except for the C++ vs C#. In other words, the model fails to identify actual positive instances (higher FN) while also incorrectly labeling negative instances as positive (higher FP). Finally, Tables 12 and 13 show the results for the accuracy, precision, recall, and f-measure for the results of our trained model. Consistently uniform values were noticed across all languages. In an effort to decipher this phenomenon, a closer examination of our dataset revealed a crucial insight. The application of a correlation threshold, removing columns with coefficients exceeding 0.75, inadvertently resulted in a reduced dataset size. As a consequence, the models, regardless of the programming language, were operating with a substantially similar set of data.

5.4 Effective Transfer Learning for Detecting Code Smells

We investigated how a transfer learning model trained with bug-related code snippets detects code smells across languages. Using the Perceptron algorithm, models were trained on Java, C#, and C++ snippets. As shown in Table 13, all models had identical accuracy, precision, recall (100%), and F1 score (0.6), indicating consistent performance across languages.

Table 14 compares success rates across languages, with Java-to-C#, C#-to-C++, and Java-to-C++ showing 67%, while the lowest was 63%. Overall, the rates ranged from 64-65%.

The predictive power of bug-related snippets stems from shared characteristics like high cyclomatic complexity, large code size, and strong class coupling, commonly seen in smells like Long Method, God Class, and Complex Method.

Table 13. Accuracy, precision, recall, and F-measure for the smell correctly classified by the prediction model trained with bugs

Model Language	Accuracy	Precision	Recall	F1
Java	0.500	0.500	1.000	0.666
C#	0.500	0.500	1.000	0.666
C++	0.500	0.500	1.000	0.666

Table 14. Results for Model Trained with Bugs and Tested with Code Smells

	Java	C#	C++
Java	65%	64%	67%
C#	67%	63%	67%
C++	67%	64%	67%

We also acknowledge that some smells share similar characteristics, like the Long Method and the Complex Method, which could hinder differentiation. Future work will refine feature sets to address this overlap.

Finding 2: Code snippets with bugs are strong predictors for detecting code smells in transfer learning models.

The outcomes of our transfer learning model for code smell detection are displayed in Figure 4. Through confusion matrices, a comprehensive breakdown of the model's performance across various programming languages is depicted. In the matrices, we can see that most cases involving the C++ language had a small sample because the dataset with C++ bugs only had 18 cases. Moreover, we can see that in this model, the results are better since we can see a higher number of True positive cases. However, the number of false positives is still high.

General Finding: Transfer learning models trained with code smells are poorly suited to detect bugs, while the ones trained with bugs are better suited to detect code smells. However, both models still need improvements.

6 Threats to Validity

6.1 Construct and Internal Validity.

The accuracy of code smell detection and thresholds used could influence results. To mitigate this, we based our techniques on established work (Palomba et al., 2018; Amorim et al., 2016; Kovacevic et al., 2022) and aligned data collection and algorithm selection with relevant research (Fontana Arcelli et al.; Sharma et al., 2019; Ardimento and et al; De Stefano et al.; Fontana Arcelli and Zanoni, 2017) to enhance robustness.

Our methodology faced limitations, including constraints on conducting a full ablation study due to limited labeled data. Instead, we performed sample-size-related ablation in

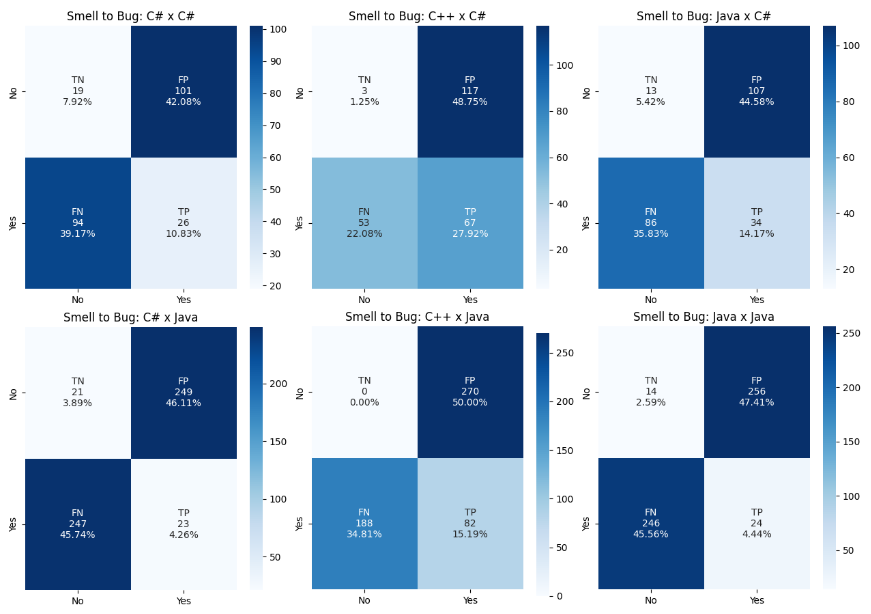


Figure 3. Confusion Matrix of the models trained with smells and tested with bugs

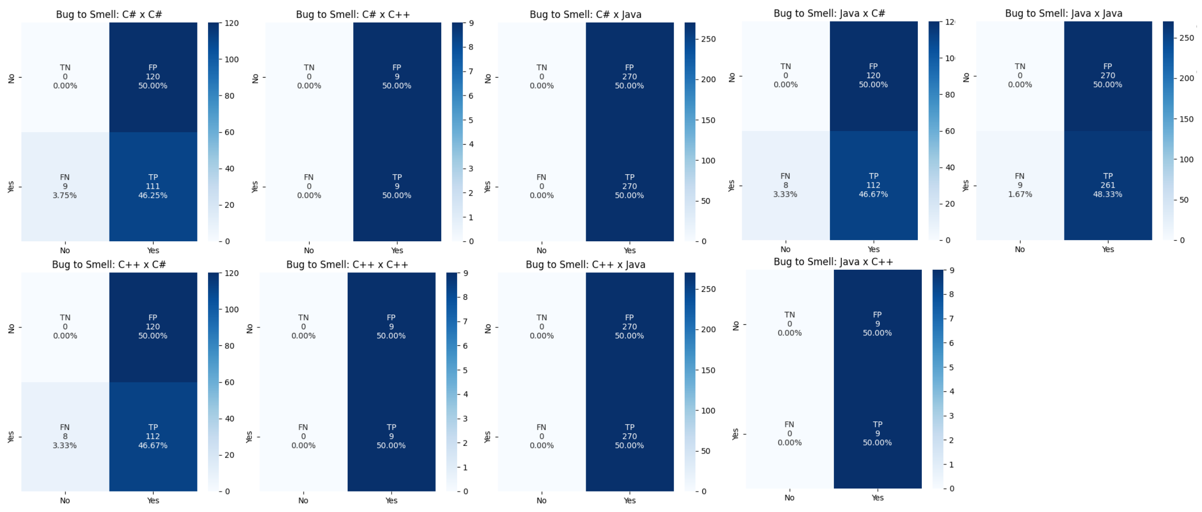


Figure 4. Confusion Matrix of the models trained with bugs and tested with code smells

RQ2, acknowledging this as a limitation. Transfer learning models showed limited effectiveness in predicting bugs from code smells, likely requiring more sophisticated models or additional features.

6.2 External Validity.

Our research also lacked a baseline comparison, which would have been valuable for evaluating the effectiveness of our transfer learning approach. The domain of harmful code detection and the application of transfer learning are relatively novel, which explains the absence of similar studies. We addressed this by comparing models within the same language (e.g., C++ vs C++) for a more accurate cross-language comparison.

It is essential to ensure that performance differences between languages and code smells are not due to random variations. We used established methods to collect buggy and smelly data (Oizumi et al., 2019; Oizumi et al., 2018; Sousa et al., 2017; Falcão et al., 2020; Rodrigo Lima et al., 2020).

Our exclusive use of open-source projects may limit the generalizability to proprietary software environments. However, open-source projects offer diversity, transparency, and replicability, enabling a wide analysis of code smells across various contexts. They often reflect industry standards and real-world practices, despite potential differences from proprietary software. Open-source projects also provide long development histories and diverse coding styles, making them valuable for studying code smells.

7 Conclusion

In this study, we applied transfer learning for harmful code detection across Java, C#, C++, and Python. Strong knowledge transfer was observed between Java and C#, while C++ posed greater challenges. A sample size of 32 was effective for most code smells, though larger datasets were needed for complex cases like Multifaceted Abstraction.

Our findings highlight transfer learning's potential in cross-language harmful code detection and the importance of sample size and feature selection. While code smells were not ideal for predicting bugs, bugs effectively predicted code smells.

Acknowledgements

This work is partially supported by FACEPE (APQ-0399-1.03/17), CNPq (423125/2021-4, 315532/2021-1, 465614/2014-0), CAPES (88887.136410/2017-00), PRONEX (APQ/0388-1.03/14), the Alexander Von Humboldt Foundation.[5], and INES (www.ines.org.br). The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions.

References

L. Amorim, Costa, et al. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. *ISSRE 2015*, 2016.

P. Ardimento and A. et al. Transfer learning for just-in-time design smells prediction using temporal convolutional networks. *ICSOFT 2021*.

C. Barbosa, Uchoa, et al. Revealing the social aspects of design decay: A retrospective study of pull requests. In *SBES*, pages 364–373, 2020.

C. Barbosa, Uchoa, et al. Beyond the code: Investigating the effects of pull request conversations on design decay. 2023.

M. De Stefano, Pecorelli, et al. Comparing within- and cross-project machine learning algorithms for code smell detection. *MaLTESQuE '21*.

F. Falcão, Barbosa, et al. On relating technical, social factors, and the introduction of bugs. In *2020 (SANER)*. IEEE, 2020.

F. Fontana Arcelli and M. Zanoni. Code smell severity classification using machine learning techniques. *Knowledge-Based Systems*, 2017.

F. Fontana Arcelli, Mäntylä, et al. Comparing and experimenting machine learning techniques for code smell detection. *ESEM 21*.

M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

E. Kalliamvakou, Gousios, et al. An in-depth study of the promises and perils of mining github. *ESEM*, 2016.

L. N. Kanal. Perceptron. In *Encyclopedia of Computer Science*, pages 1383–1385. 2003.

A. Kovacevic, Slivka, et al. Automatic detection of long method and god class code smells through neural source code embeddings. *Expert Systems With Applications 204*, 2022.

R. Krishna and T. Menzies. Bellwethers: A baseline method for transfer learning. *IEEE Transactions on Software Engineering*, 45:1081–1105, 2019.

W. Oizumi, A. Garcia, T. Colanzi, M. Ferreira, and A. Staa. When code-anomaly agglomerations represent architectural problems? An exploratory study. In *SBES; Maceio, Brazil*, pages 91–100, 2014.

W. Oizumi, A. Garcia, T. Colanzi, A. Staa, and M. Ferreira. On the relationship of code-anomaly agglomerations and architectural problems. *JSERD*, 2015.

W. Oizumi, A. Garcia, L. Sousa, B. Cafeo, and Y. Zhao. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In *The 38th ICSE; USA*, 2016.

W. Oizumi, L. Sousa, A. Oliveira, A. Garcia, O. Agbachi, R. Oliveira, and C. Lucena. On the identification of design problems in stinky code: experiences and tool support. *J. Braz. Comp. Soc.*, 24(1):13:1–13:30, 2018.

W. Oizumi, L. Sousa, et al. On the density and diversity of degradation symptoms in refactored classes: A multi-case study. In *IEEE 30th ISSRE*, 2019.

F. Palomba, Bavota, et al. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *EMSE 2018*, 2018.

F. Palomba, M. Zanoni, F. Arcelli Fontana, A. De Lucia, and R. Oliveto. Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering*, 45(2):194–218, 2019.

D. Pereira Cesar, C. Barbosa Vieira da Silva, et al. Is your code harmful too? understanding harmful code through transfer learning. In *SBQS*, SBQS '23, 2023. ISBN 9798400707865. . URL `https://doi.org/10.1145/3629479.3629512`.

I. M. Roberta Arcoverde, A. G. Christina Chavez, , and A. V. Staa. On the relevance of code anomalies for identifying architecture degradation symptoms. *CSMR*, pages 277–286, 2012.

J. S. Rodrigo Lima, B. F. Leopoldo Teixeira, R. G. Márcio Ribeiro, and A. G. Rafael de Mello. Understanding and detecting harmful code. *SBES '20*, 2020.

T. Sharma and D. Spinellis. A survey on software smells. *J. Syst. Softw. (JSS)*, 138:158–173, 2018.

T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis. On the feasibility of transfer-learning code smells using deep learning. *ESEM*, 1, 2019.

L. Sousa, R. Oliveira, et al. How do software developers identify design problems?: A qualitative analysis. In *SBES*, SBES'17, 2017.

A. Takahasi, N. Sae-Lim, S. Hayashi, and M. Saeki. A preliminary study on using code smells to improve bug localization. *ICPC '18*, 2018.

L. Torrey and J. Shavlik. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pages 242–264. IGI global, 2010.

J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes*, 30:1–5, 2005.

[5]https://www.humboldt-foundation.de/