FEDERAL UNIVERSITY OF ALAGOAS
INSTITUTE OF COMPUTING
GRADUATE PROGRAM IN INFORMATICS

Master's Dissertation

Code Smells Detection Across Programming Languages

André Moabson da Silva Ramos
amsr@ic.ufal.br

Advisors:

Baldoino Fonseca dos Santos Neto

Rafael Maiani de Mello

MACEIÓ, JUNHO DE 2023

André Moabson da Silva Ramos

Code Smells Detection Across Programming Languages

Dissertation presented as a partial requirement for obtaining a Master's degree from the Graduate Program in Informatics at the Institute of Computing at the Federal University of Alagoas.

Advisors:

Baldoino Fonseca dos Santos Neto

Rafael Maiani de Mello

Maceió, junho de 2023

Catalogação na Fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico

Bibliotecário: Marcelino de Carvalho Freitas Neto – CRB-4 - 1767

R175c Ramos, André Moabson da Silva.
Code smells detection across programming languages / André Moabson da Silva Ramos. – 2023.
39 f. : il.

Orientador: Baldoino Fonseca dos Santos Neto.
Co-orientador: Rafael Maiani de Mello.
Dissertação (mestrado em informática) - Universidade Federal de Alagoas. Instituto de Computação. Maceió, 2023.
Texto em inglês.

Bibliografia: f. 36-39.

1. Engenharia de software. 2. Aprendizado do computador. 3. Aprendizagem profunda. 4. Transferência de aprendizagem. 5. Code smells. 6. Linguagem de programação (Computadores). I. Título.

CDU: 004.43

Resumo

A incidência de *code smells* é frequentemente associada à degradação da qualidade do software. Vários estudos apresentam a importância de técnicas para detectar e combater a incidência deles no código-fonte. No entanto, as técnicas existentes para detectar *code smells* dependem da linguagem de programação. Consequentemente, várias linguagens de programação são amplamente empregadas pela comunidade de software sem técnicas adequadas de detecção. Nosso estudo aborda a ampliação da disponibilidade de técnicas de detecção de *code smells* para diferentes linguagens de programação por meio do aprendizado de transferência. Selecionamos cinco linguagens de programação entre as dez linguagens mais utilizadas de acordo com *StackOverflow*: `Java`, `C++`, `Python`, `C#`, e `JavaScript`. Além disso, algumas dessas linguagens possuem características semelhantes entre si, como Java e C#, o oposto pode-se dizer de Java e Python. Extraímos os conjuntos de dados para treinamento e teste de modelos de aprendizado profundo de `150` projetos de código aberto. Os resultados indicam que as técnicas de aprendizagem por transferência detectam de forma eficaz e eficiente os *code smells*, independentemente da linguagem de programação e da quantidade de camadas da arquitetura de aprendizagem profunda usada na aprendizagem por transferência. Essas descobertas podem ajudar desenvolvedores e pesquisadores a aplicar as mesmas técnicas de detecção de *code smells* em diferentes linguagens de programação.

Palavras-chave: Engenharia de Software, Machine Learning, Deep Learning, Transfer Learning, Code Smells, Linguagens de Programação.

Abstract

The incidence of code smells is often associated with software quality degradation. Several studies present the importance of techniques to detect and tackle the incidence of smells in the source code. However, existing techniques to detect code smells depend on the programming language. Consequently, several programming languages are largely employed by the software community without proper techniques of code smell detection. Our study addresses amplifying the availability of code smell detection techniques to different programming languages through transfer learning. We select five programming languages among the ten most used languages according to *StackOverflow*: `Java`, `C++`, `Python`, `C#`, and `JavaScript`. Also, some of these languages have similar characteristics to each other, such as Java and C# as opposed to Java and Python. We extract the datasets for training and testing of deep learning models from `150` open sources projects. Results indicate that transfer learning techniques effectively and efficiently detect code smells regardless of the programming language and number of layers of the deep learning architecture used in transfer learning. These findings can help developers and researchers to apply the same code smell detection techniques in different programming languages.

Keywords: Software Engineering, Machine Learning, Deep Learning, Transfer Learning, Code Smells, Programming Languages.

List of Figures

List of Tables

Contents

1 Introduction

This chapter presents the motivation and objectives (general and specifics) and the structure of this work.

1.1 Motivation

Code smells are frequently associated with software quality degradation (LIMA et al., 2020; UCHÔA et al., 2020). Developers should properly identify and combat smells in the source code (BIBIANO et al., 2021; BIBIANO et al., 2019), avoiding worst design problems (OIZUMI et al., 2016) and the increasing of technical debt (ZAZWORKA et al., 2014; IAMMARINO et al., 2019). Despite the still need for manual validation (MELLO; OLIVEIRA; GARCIA, 2017; HOZANO et al., 2018; HOZANO et al., 2017; MELLO et al., 2019), techniques for supporting the detection of code smells have considerably evolved in the last decade (PAIVA et al., 2017; AZEEM et al., 2019). However, these techniques have common limitations that hamper their reuse in different contexts. One of the limitations addresses the design of detection algorithms and rules tailored to specific programming languages. In this sense, one may see a clear prevailing of techniques for detecting code smells in Java software projects (FERNANDES et al., 2016; PAIVA et al., 2017).

The support for code smell detection in a single programming language is a limitation, even among the more recent deep learning-based techniques for detecting code smells. Currently, several empirically evaluated deep learning models support the detection of code smells (OLIVEIRA et al., 2020; PECORELLI et al., 2019; AZEEM et al., 2019; SHARMA; SPINELLIS, 2018; LIU et al., 2019; JEBNOUN et al., 2020). These models can be dynamically calibrated for specific contexts based on the training datasets employed, which includes applying a standard programming language. In general, the effectiveness of deep learning techniques for code smells detection is promising, frequently reaching higher effectiveness than traditional detection approaches grounded on metrics and/or detection rules (PECORELLI et al., 2019; AZEEM et al., 2019). The use of deep learning for supporting Software Engineering activities follows a trend observed in different domains. Deep learning models have been largely employed in developing software solutions for several purposes. Examples include applications for supporting medical diagnosis (KOUROU et al., 2015), building autonomous cars (Hussain; Zeadally, 2019), and detection of fraudulent credit card transactions (Dhankhad; Mohammed; Far, 2018).

The development of conventional deep learning techniques is grounded on training models over reliable training datasets. The effectiveness of the resulting model is measured through testing datasets. If the effectiveness of the deep learning model is considered satisfactory, the model may be employed in real settings. The typical training process commonly requires applying training datasets as similar as possible to real settings. For instance, if we want to build deep

learning models to detect different types of code smell in `Python` web applications, we should compose large training datasets for each code smell type according to these settings.

However, composing training datasets at a large scale may be a tedious and costly task. Besides, several popular programming languages lack automated support for detecting even the most well-known code smells. In such cases, we may consider applying transfer learning strategies (SHARMA; SPINELLIS, 2018). Transfer learning employs pre-trained deep learning models tailored to solve a specific task to solve another related task. Consequently, transfer learning contributes to significantly reducing the consumption of computational resources and the consumption of human resources on labeling/relabeling (PAN; YANG, 2010).

1.2 Objectives

1.2.1 General Objective

In this Master's dissertation, we report an empirical study aiming to investigate the feasibility of transfer learning for detecting the incidence of six smell types `Complex Method`, `Long Method`, `Shotgun Surgery`, `Feature Envy`, `Divergent Change`, and `God Class` in open source projects from `five` popular programming languages. In particular, we investigate the effectiveness and efficiency of transfer learning for detecting code smells in the projects analyzed in our study. Then, we analyze how many layers a deep learning architecture must have for the resulting model to effectively and efficiently detect code smells in projects from different programming languages. We expect that our research may contribute to expanding the applicability of code smell detection techniques and reducing efforts on identifying code smells in software projects from different programming languages. For our study, we collect code snippets from `150` open source projects written in five popular programming languages (`C++`, `C#`, `Python`, `Java`, and `JavaScript`) and build models through two deep learning architectures (*Perceptron* and *Convolutional Neural Networks - CNN*).

1.2.2 Specific Objectives

- Develop a tool to detect code smells through metrics;
- Build datasets for each pair programming language and smell;
- Investigate the use of transfer learning in the context of detecting code smells for different programming languages;
- Evaluate the potential benefits of transfer learning compared to traditional machine learning.

1.3 Work Structure

The work was organized into eight chapters, consisting of the current and the following:

- Chapter 2 presents the main subjects of our study;
- Chapter 3 presents the study design and settings of our study;
- Chapter 4 presents the results of our study, reporting its main findings by research question;
- Chapter 5 discusses the findings of our study and their impact on research and practice;
- Chapter 6 discusses related work on support for code smell detection;
- Chapter 7 discusses the main threats to validity identified in our study;
- Finally, Chapter 8 concludes and indicates future work.

2 Study Background

This chapter presents the main subjects of this work, presenting the code smells, deep learning architectures and transfer learning.

2.1 Code Smells

The term *code smells* refers to a couple of characteristics or patterns in source code that indicate potential problems, bad choices of design and therefore areas where code quality could be improved (FOWLER et al., 1999).

Table 1 presents the description of each code smells that we will address.

Table 1 – Code Smells

Name	**Description**
Complex Method	Occurs when a method has a high cyclomatic complexity.
Long Method	Occurs when a method has a high number of lines.
Shotgun Surgery	Occurs when one change leads to batch changes in other classes.
Feature Envy	Occurs when a method accesses more data from another class than the class to which it belongs.
Divergent Change	Occurs when a class is changed in different ways for different reasons.
God Class	Occurs when a class has a concentration of responsibilities having a high number of data members, methods and low cohesion.

The motivations behind the choice of these code smells are due to the fact that they are widely discussed in previous studies and have a significant impact on software quality (CEDRIM et al., 2017; SHARMA; SPINELLIS, 2018; SHARMA et al., 2021; BIBIANO et al., 2021). For instance, a method with a high number of lines and containing multiples paths in control-flow graph, i.e., `Long Method` and `Complex Method`, can lead to misunderstand of how method works and make it hard to maintain. Besides, `Feature Envy`, `God Class` and `Shotgun Surgery` are associated with the presence software bugs (CAIRO; CARNEIRO; MONTEIRO, 2018). For comparison purposes, we will address the `Divergent Change` due to its similarity to `Shotgun Surgery`.

Table 2 presents the descriptions of the software metrics according to the Understand tool [1] documentation. These metrics and the way they are computed will be used to detect the mentioned code smells.

Table 2 – Metrics

Name	**Understand**	**Description**
Cyclomatic Complexity (CC)	Cyclomatic	McCabe Cyclomatic complexity as per the original NIST paper on the subject. The cyclomatic complexity of any structured program with only one entrance point and one exit point is equal to the number of decision points contained in that program plus one. Understand counts the keywords for decision points (*for*, *while*, etc) and then adds 1. For a *switch* statement, each *case* is counted as 1. For languages with Macros, the expanded Macro text is also included in the calculation.
Number of Lines (LOC)	CountLine	Number of physical lines.
Number of Methods (NOM)	CountDeclMethod	Number of local (not inherited) methods.
FanIn	CountInput	The number of inputs a function uses plus the number of unique subprograms calling the function. Inputs include parameters and global variables that are used in the function, so Functions calledby + Parameters read + Global Variables read. Of the two general approachs to calculating FANIN (informational versus structural) ours is the informational approach.

[1] https://scitools.com

FanOut	CountOutput	The number of outputs that are SET. This can be parameters or global variables. So Functions calls + Parameters set/modify + Global Varibales set/modify. Of the two general approachs to calculating FANOUT (informational versus structural) ours is the informational approach.
Coupling Between Objects (CBO)	CountClassCoupled	The Coupling Between Object Classes (CBO) measure for a class is a count of the number of other classes to which it is coupled. Class A is coupled to class B if class A uses a type, data, or member from class B. This metric is also referred to as Efferent Coupling (Ce). Any number of couplings to a given class counts as 1 towards the metric total Chidamber & Kemerer suggest that: 1) Excessive coupling between object classes is detrimental to modular design and prevents reuse. 2) Inter-object class couples should be kept to a minimum. 3) The higher the inter-object class coupling, the more rigorous testing needs to be.
Lack of Cohesion in Methods (LCOM)	PercentLackOfCohesion	100% minus average cohesion for class data members. Calculates what percentage of class methods use a given class instance variable. To calculate, average percentages for all of that class'es instance variables and subtract from 100%. A lower percentage means higher cohesion between class data and methods.

2.2 Deep Learning

2.2.1 Perceptron

Perceptron is a single-layer neural network composed by one neuron this idea was initially proposed by McCulloch and Pitts in 1943 (MCCULLOCH; PITTS, 1943). This type of neural network is able to solve linearly separable problems.

The perceptron can be divided into four parts as shown in Figure 1. The first part is composed of the input values, the second is weights, the third is weighted sum between these values and the four is activation function that is calculated based on this sum producing the output. The learning process consists of determining the weights so that the error is as small as possible.

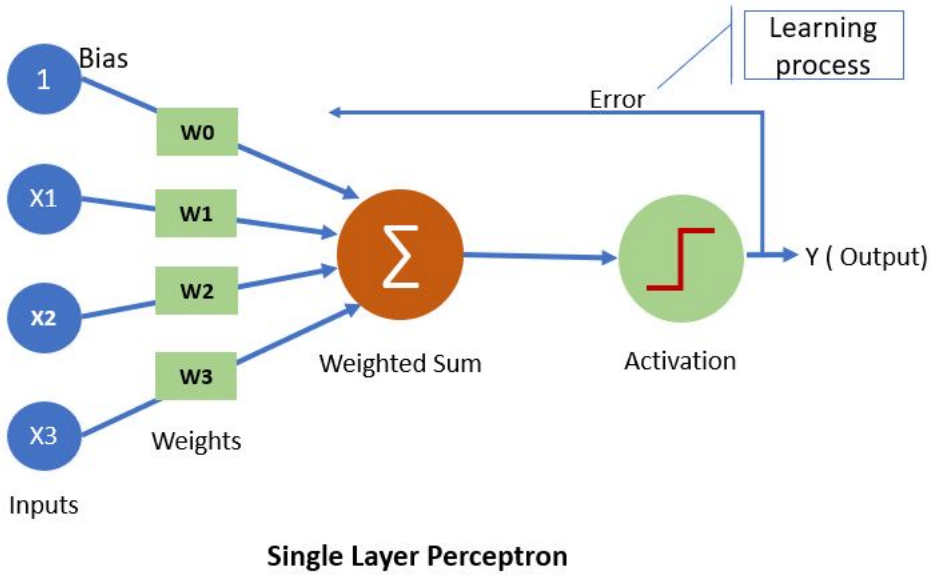

Figure 1 – Perceptron. Source: (Sumanth Bathula, 2018).

2.2.2 Convolutional Neural Network (CNN)

Convolutional Neural Network (CNN) is a feedforward neural network architecture that has at least one convolutional layer and performs learning through of optimization algorithms such as gradient descent and backpropagation (SHRESTHA; MAHMOOD, 2019). This type of neural network is widely used in the classification of images, audio and object detection (HUSSAIN; BIRD; FARIA, 2019; MACCAGNO et al., 2021; GALVEZ et al., 2018).

Automatic feature extraction is one of the main advantages of this architecture, the layer responsible for this activity is called the convolutional layer. Figure 2 shows the process to compute a feature map as output of the convolutional layer. In order to compute one value of feature map a sub matrix of the input matrix is selected, then their values are multiplied by the kernel matrix and the sum of these values is the final result, this process is called convolutional operation and is repeated moving the sub matrix along of the input. The filter applied to input in

order to obtain the feature map is composed by multiples kernels and their values are randomly initialized and updated as the network is trained.

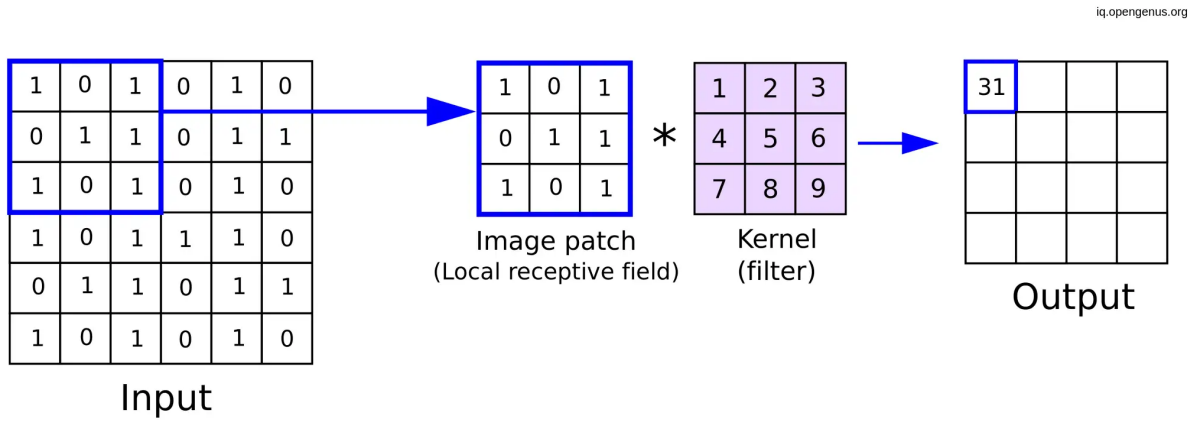

Figure 2 – Convolutional Layer. Source: (SuperAnnotate, 2023).

The first well-known application of this type of architecture was the recognition of handwritten digits using the LeNet architecture (LECUN et al., 1998). The proposed architecture (figure 3) is composed by 7 layers, combining the use of convolutional layers, subsampling layers (pooling layers) and fully connected layers. The model was trained and evaluated using the MNIST dataset containing images of handwritten digits. The model achieved low error rates when trained with 60,000 samples and evaluated with 10,000 samples, demonstrating the effectiveness of the LeNet architecture in achieving high accuracy on this task.

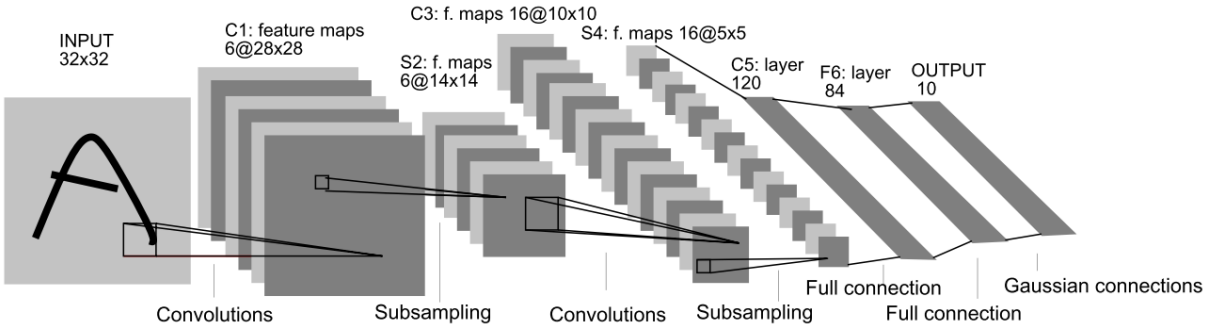

Figure 3 – Architecture of LeNet-5. Source: (LECUN et al., 1998).

Other architectures of this type also became known, such as AlexNet (KRIZHEVSKY; SUTSKEVER; HINTON, 2012) and GoogLeNet (SZEGEDY et al., 2015). The AlexNet architecture using 5 convolutional layers combined with max pooling layers and fully connected layers became notorius for winning the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) competition in 2012 and for improvements on use of the GPU, boosting the use of CNNs in computer vision tasks. In 2014, the GoogLeNet architecture became notorious, winning this year's ILSVRC competition by introducing the idea of convolution modules called "Inception modules".

2.3 Transfer Learning

Transfer learning technique consists of using a pre-trained model on a given set of data to solve a task, to build another model that aims to solve another task, taking advantage of the knowledge obtained in the first and training the second with few examples, as a way of to reduce the efforts needed to solve the second task (PAN; YANG, 2010; WEISS; KHOSHGOFTAAR; WANG, 2016; ZHUANG et al., 2019).

Figure 4 shows the difference between the two processes, where in the transfer learning process, more than one pre-trained model for different tasks can be leveraged to obtain a model that solves the final task.

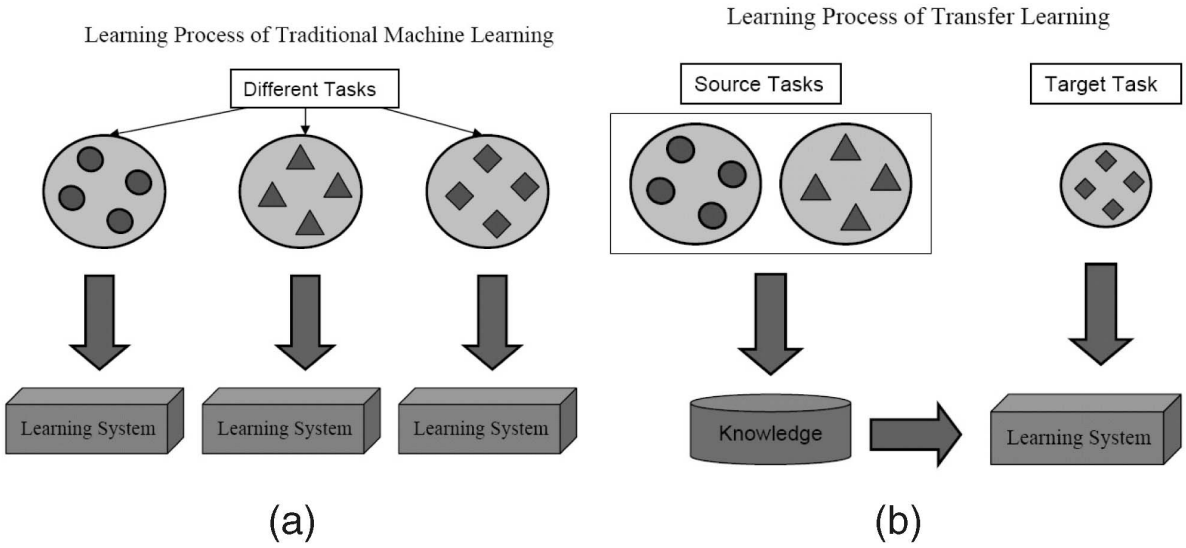

Figure 4 – Difference between learning processes, (a) traditional machine learning and (b) transfer machine learning. Source: (PAN; YANG, 2010).

In traditional machine learning (Figure 4. a) models are trained from scratch for each task without observing any relationships between the different tasks. In transfer learning (Figure 4. b), the main idea is to reuse the pre-trained models according to the similarity between the tasks. One of the motivations for using transfer learning is due to the fact that the dataset of a task can be more difficult to build and validate than another one, in this case, can be feasible to reuse a dataset that has already been validated.

The study presented by Sharma et al. (2021) explores the feasibility of applying transfer learning for code smells detection in Java and C# projects. Besides, this study defined transfer learning as the process of using a pre-trained model with code snippets containing smelly and non-smelly samples of a code smell to detect the same smell in other language different from the one used in training. The results indicate the feasible of use deep learning models based on Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN) to detect code smells in a programming language other than the one he was trained, opening up the possibilities of exploring different types of code smells and programming languages.

3 Study Design

The study presented in this Master's dissertation aims to *evaluate the use of transfer learning to detect six different smell types in five programming languages*. As transfer learning, we will consider the process of using a pre-trained model for a code smell for detect the same smell in different languages, also we will evaluate in test dataset of same language used for training aiming to compare the results. This evaluation addresses three particular aspects. The first one is *effectiveness* of transfer learning, i.e., to which extent transfer learning effectively detects code smells in different programming languages. The second aspect is the *efficiency* of transfer learning, i.e., the effort (in terms of sample size) to transfer learning be effective in code smells detection. The third aspect is the *complexity*, i.e., how many layers the deep learning architecture must have for transfer learning effectively and efficiently detects code smells. In this sense, we aim to answer the following research questions with this study:

RQ1: ***How effective is transfer learning to detect code smells?***

To answer this RQ: We investigate the effectiveness (in terms of *f-measure*) of transfer learning to detect six smell types in open source projects from five programming languages. As a result, we expect to identify which programming languages tend to produce deep learning models that would be reused for effectively identifying code smells in projects from other programming languages.

RQ2: ***How efficient is transfer learning to detect code smells?***

To answer this RQ: We analyze the effort for transfer learning effectively detecting code smells. By effort, we mean the number of instances (sample size) that compose the training dataset used to train a deep learning model. As a result, we expect to identify a specific sample size in which the transfer learning effectively detects code smells in different programming languages.

RQ3: ***How many layers must a deep learning architecture have to effectively detect code smells?***

To answer this RQ: We compare the effectiveness and efficiency of transfer learning using models resulting from two deep learning architectures: *Perceptron* and *CNN*. As a result, we expect to reveal whether the number of layers (complexity) in the architecture influences the detection of code smells by comparing a single-layer architecture versus multi-layer architecture.

In the following sections, we describe the study settings.

3.1 Programming Languages

We select five programming languages for our study: `Java`, `C#`, `C++`, `JavaScript`, and `Python`. These programming languages are among the ten most used ones according to the

2021 StackOverflow survey[1]. To observe the effect of transfer learning over diverse contexts, we intentionally selected popular script-based programming languages and compiler-based ones. Besides, these languages present certain similarities and differences valuable for the study. For instance, C++ and C# are the only programming languages offering the resource of *preprocessor directives*, a resource commonly employed for introducing variability at compilation time. Python is the only programming language that didn't offer a mechanism to control flow similar to *switch statements* until a more recent version (3.10). This version introduced a feature called *Structural Pattern Matching* with support for pattern matching. Consequently, it is expected that most of the source code implemented in Python at the time of this study does not employ this feature. Another relevant difference between Python and the other programming languages analyzed in our study is the peculiar syntactical representation of control structures such as conditional expressions.

3.2 Smell Types and Detection Rules

We select six smell types: `Complex Method`, `Long Method`, `Shotgun Surgery`, `Feature Envy`, `Divergent Change`, and `God Class`. As mentioned in 2.1 theses smells are among the most relevants in terms of software quality.

Table 3 presents the metrics employed to classify code snippets and their correspondent in Understand tool. The first column shows the metric name. The second column shows the correspondent name in the Understand tool. The third column shows the abbreviation adopted.

Table 3 – Metrics used in rules and naming used in the SciTools Understand tool.

Name	Understand	Abbreviation
Cyclomatic Complexity	Cyclomatic	CC
Number of Lines	CountLine	LOC
Number of Methods	CountDeclMethod	NOM
FanIn	CountInput	FanIn
FanOut	CountOutput	FanOut
Coupling Between Objects	CountClassCoupled	CBO
Lack of Cohesion in Methods	PercentLackOfCohesion	LCOM

Table 4 describes the programming languages and smell types analyzed in our study as well as the detection rules used to detect each smell type. We consider rules and thresholds defined in the tool DesigniteJava[2] for `Complex Method` and from previous work (CEDRIM et al., 2017) for `Long Method`, `Shotgun Surgery`, `Feature Envy`, `Divergent Change` and `God Class`.

Some programming languages were not included for certain types of code smells due to being script-based and we did not find an appropriate way to detect them to build the datasets. For instance, JavaScript does not have a class structure like Java, although there is a reserved

[1] https://insights.stackoverflow.com/survey/2021#most-popular-technologies-language
[2] https://github.com/tushartushar/DesigniteJava

Table 4 – Rules and Thresholds for each Code Smell

Code Smell	Programming Languages	Rule
Complex Method	Java, C#, C++, JavaScript and Python	CC >= 8
Long Method	Java, C#, C++ and Python	(LOC > 50) and (CC > 5)
Shotgun Surgery	Java, C# and C++	(CC > 4) and (FanOut > 7)
Feature Envy	Java, C# and C++	(CC > 4) and (FanOut > 4) and (LCOM < 30%)
Divergent Change	Java, C# and C++	(FanIn > 10) and (LCOM < 50%) and (CC > 4)
God Class	Java, C#, C++ and Python	[(LOC >150) and (CBO > 6)] or [(NOM > 15) and (CBO > 6)]

word called *class* and syntactically similar structure, it's just a convenient way to work with the prototype system. Therefore, JavaScript and Python were not included for `Shotgun Surgery`, `Feature Envy` and `Divergent Change`. We can easily include the JavaScript in the list of `Long Method` for future works.

3.3 Projects

We manually select 30 open-source projects from GitHub[3] for each programming language investigated in our study, i.e., 150 projects in total. From these projects, we collect the metrics used by the rules for code smells detection. Table 5 describes the total number of classes and methods of the projects analyzed. `Java` is the programming language with the highest number of classes collected, and `C++` presents the highest number of methods.

Table 5 – Number of code snippets

Programming Languages	Number of Classes	Number of Methods
Java	51,326	324,943
C++	32,425	414,923
C#	26,719	176,531
Python	24,017	129,623
JavaScript	-	61,997

Table 6 describes the number of code smells detected for each programming language. Besides, code snippets not classified as smelly are used as negative samples.

Table 6 – Number of smelly snippets

Programming Languages	CM	LM	SS	FE	DC	GC
Java	4,380	2,119	13,297	4,605	1,557	8,863
C++	11,026	4,531	20,977	3,928	611	11,392
C#	2,821	1,201	6,802	1,959	372	5,252
Python	4,325	1,296	-	-	-	2,534
JavaScript	5,950	-	-	-	-	-

[3] https://github.com

3.4 Source Code Metrics

We face difficulties in identifying and applying open-source tools available for gathering the metrics in the five programming languages investigated. The first challenge was identifying tools covering all the programming languages investigated in our study. The second challenge was that the available tools often do not sufficiently cover the desired metrics for our study. For instance, there are few or no tools for script-based languages like JavaScript and Python. Some of the tools found may be considered obsolete, considering the evolution of these programming languages.

To overcome these challenges, we built our own tool based on the Tree Sitter[4] library. With this tool, we could parse the source code of the selected projects for analyzing the *Abstract Syntax Tree* (AST), gathering the necessary metrics, and obtaining the desired code snippet for the experiment. Our tool is similar to the Designite tool for capturing but addressing different programming languages, including *Cyclomatic Complexity* (MCCABE, 1976), the number of *nested methods* in a method for script-based languages, the number of *struct* and *enum* declarations in C++ methods, and the number of *lambda expressions*. The Scitools Understand was used to complement the data in our study, collecting metrics such *FanIn*, *FanOut*, *CBO*, *LCOM* and others, this tool also permits capturing the piece of code of interest thought a `Python` API[5].

3.5 Data Preparation

To train the deep learning models based on the syntactical characteristics of the code snippets, we need to select a single technique to represent the code elements from all programming languages. To do that, we select the technique *tokenization*, which transforms each character into a numerical token. *Tokenization* has been employed in a previous study on transfer learning among two programming languages (SHARMA et al., 2021). For composing the tokens, we employ the tool provided by Spinellis et al. (SPINELLIS, 2019).

Figure 5 shows the application of *Tokenization* for methods in `Python` (see Figure 5a) and `Java` (see Figure 5b). The left side presents the methods and their respective tokens on the right side. The tokens generated for these programming languages have some differences due to their structural differences. For instance, the method declaration in `Java` has access modifiers and type declaration, unlike Python. The tokens differ in conditional expressions because there are some differences in logical operators and identity operators. Also, the `Java` method uses *switch* whereas *if-elif-else* in `Python`.

[4] https://tree-sitter.github.io
[5] https://documentation.scitools.com/html/python/index.html

```python
def do_something(op, condition, *values):
    values = filter(condition, values) if condition is not None else values

    if op == '+':
        return sum(values)
    elif op == '*':
        return functools.reduce(operator.mul, values, 1)
    else:
        raise Exception('invalid operator')
```

```
348 2000 40 2001 44 2002 44 42 2003 41 58
  2003 61 2004 40 2002 44 2003 41 392 2002 407 422 303 360 2003

  392 2001 614 648 58
    450 2005 40 2003 41
  359 2001 614 648 58
    450 2006 46 2007 40 427 46 2008 44 2003 44 1501 41
  360 58
    440 2009 40 648 41
```

(a) Python 3.8.7

```java
public static int doSomething(char op,
                  Predicate<? super Integer> condition,
                  Integer ...values) {

    var stream = condition != null ?
                  Stream.of(values).filter(condition) : Stream.of(values);

    return switch (op) {
        case '+' -> stream.reduce(0, Integer::sum);
        case '*' -> stream.reduce(1, (accumulator, value) -> accumulator * value);
        default -> throw new IllegalArgumentException("invalid operator");
    };
}
```

```
439 457 404 2000 40 330 2001 44
          2002 60 63 463 2003 62 2004 44
          2003 613 2005 41 123

  490 2006 61 2004 631 424 63
          2007 46 2008 40 2005 41 46 2009 40 2004 41 58 2007 46 2008 40 2005 41 59

  450 464 40 2001 41 123
    328 607 45 62 2006 46 2010 40 1500 44 2003 58 58 2011 41 59
    328 607 45 62 2006 46 2010 40 1501 44 40 2012 44 2013 41 45 62 2012 42 2013 41 59
    349 45 62 469 418 2014 40 648 41 59
  125 59
125
```

(b) Java 17

Figure 5 – Source code tokens representation

3.6 Composing the Datasets and Training the Models

Figure 6 presents the process for composing each dataset used in our study to train and test the deep learning models. Each row of the dataset contains the representation of the code snippet transformed into a sequence of numerical tokens. These tokens are the input for the training and testing of the models.

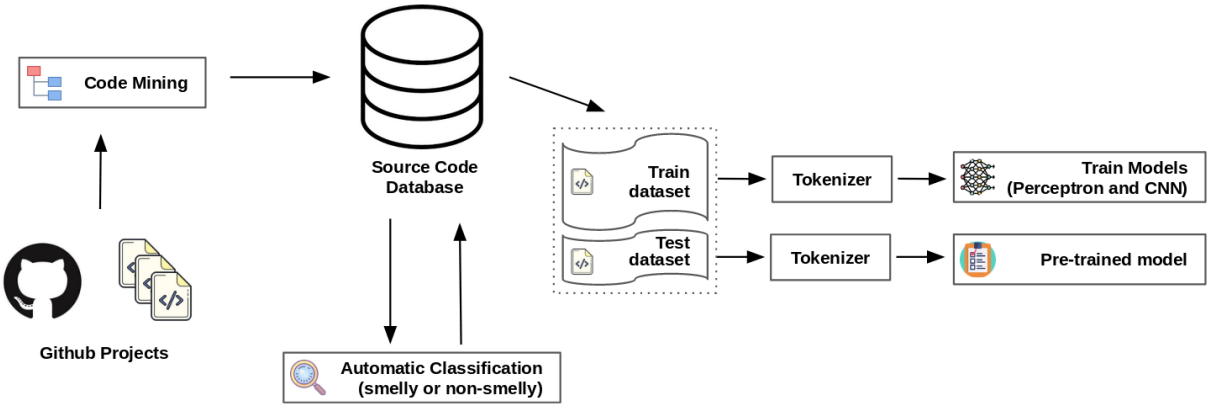

Figure 6 – Datasets Composition

First, we download the open-source projects from GitHub. In the second step, we mine relevant data from the projects storing classes and methods with their respective collected metrics resulting from the AST analysis. In the third step, we perform the automatic classification of code snippets by applying the code smell detection rules, identifying the smelly or non-smelly methods and classes, i.e., any code snippet that does not conform to the defined rule will be considered non-smelly for the specific rule it refers to. In the fourth step, we randomly select the automatic classified samples for training and testing the models, ensuring that the code snippets that will be used for testing are different from those used for training. Finally, we use the tokenizer tool to transform the textual code snippets into a sequence of numbers, and then we train the models.

Table 7 presents the datasets resulting from this process. For each programming language, we have a training dataset (named *Training*) and a testing dataset (named *Testing*). Each training dataset contains 1000 instances, and each testing dataset contains 500 instances of code snippets. In each dataset, half of the code snippets contain smells, and the other half have no smell.

Table 7 – Training and Testing Datasets

Name	Programming Languages	Smell / No Smell
Training	Java	500 / 500
	C#	500 / 500
	C++	500 / 500
	Python	500 / 500
	JavaScript	500 / 500
Testing	Java	250 / 250
	C#	250 / 250
	C++	250 / 250
	Python	250 / 250
	JavaScript	250 / 250

3.7 Data Analysis

To answer **RQ1**, we use the *Perceptron* architecture to train a deep learning model over training datasets containing a sample size of 1000 instances of code snippets, as described in Table 7. We train a deep learning model for each pair of programming languages and smell types analyzed in our study. Then, we test the trained models to detect code smells in the projects analyzed in our study. For each test, we evaluate the effectiveness of the models by calculating the *f-measure*.

In **RQ2**, we also use the *Perceptron* architecture to train a deep learning model for each pair of programming languages and smell types analyzed in our study. Then, we test the trained models to detect code smells in the projects analyzed in our study. But now, we perform the

training and testing of the models over datasets containing different sample sizes. In particular, we use datasets containing 32, 64, 128, 256, 512, and 1000 instances of code snippets. In each dataset, half of the code snippets contain smells, and the other half have no smell. This way, we can analyze the efficiency of the trained models to detect code smells in the projects analyzed in our study. By efficiency, we mean the effort (sample size) needed for a deep learning model effectively detects code smells in different programming languages.

To answer **RQ3**, we use the **CNN** architecture to train deep learning models to detect code smells in the projects analyzed in our study. Then, we evaluate the effectiveness and efficiency of these models following the procedure adopted in the **RQ1** and **RQ2**. Finally, we compare the effectiveness and efficiency between the *Perceptron* and *CNN* architectures.

4 Results and Discussion

In this chapter, we describe the main results of the study.

RQ1. How effective is transfer learning to detect code smells?

In this research question, we analyze the effectiveness of transfer learning using the *Perceptron* model to detect code smells in the programming languages analyzed in our study. For each programming language and smell type, we train a model over a training dataset containing code snippets and smells written in such language. Then, we evaluate the effectiveness of each trained model in detecting smells over the testing datasets containing code snippets of all programming languages analyzed in our study. For example, suppose we train a model in the training dataset containing Python code snippets and the smell type *Complex Method*. In that case, we evaluate this model to detect different smell types (*Complex Method*, *God Class*, *Long Method*, *Shotgun Surgery*, *Feature Envy*, and *Divergent Change*) in the testing datasets containing Python, JavaScript, C++, Java, and C# code snippets. Table 8 presents the main results to support this discussion. The first and second columns describe the smell types and programming languages used to train the model. The smell types and programming languages arranged horizontally correspond to the ones contained in the testing datasets used to evaluate the effectiveness of the model trained.

Table 8 – Effectiveness of Transfer Learning using Perceptron model

		Complex Method					God Class				Long Method				Shotgun Surgery			Feature Envy			Divergent Change		
		Java	C#	C++	JavaScript	Python	Java	C#	C++	Python	Java	C#	C++	Python	Java	C#	C++	Java	C#	C++	Java	C#	C++
Complex Method	Java	98%	96%	96%	93%	91%	83%	84%	60%	84%	98%	97%	96%	95%	87%	84%	73%	78%	80%	56%	89%	85%	81%
	C#	98%	98%	96%	93%	92%	82%	83%	64%	84%	97%	97%	96%	94%	88%	87%	77%	81%	85%	59%	92%	88%	83%
	C++	96%	97%	96%	93%	94%	83%	85%	51%	83%	97%	98%	96%	94%	76%	83%	74%	66%	80%	51%	85%	84%	78%
	JavaScript	97%	96%	95%	95%	94%	83%	85%	71%	85%	97%	96%	96%	94%	86%	83%	78%	76%	82%	66%	87%	87%	84%
	Python	82%	84%	90%	81%	97%	84%	82%	58%	84%	96%	95%	96%	95%	67%	63%	68%	50%	57%	51%	72%	64%	74%
God Class	Java	23%	21%	28%	37%	4%	94%	89%	42%	53%	48%	53%	67%	7%	9%	10%	25%	4%	9%	11%	15%	23%	35%
	C#	28%	28%	36%	44%	8%	94%	92%	46%	53%	63%	64%	76%	18%	12%	16%	35%	5%	17%	15%	20%	32%	42%
	C++	9%	7%	36%	12%	3%	51%	40%	81%	24%	16%	12%	57%	5%	10%	4%	31%	1%	12%	19%	10%	6%	14%
	Python	28%	33%	38%	41%	22%	93%	88%	48%	96%	60%	70%	75%	53%	16%	18%	35%	5%	18%	18%	26%	29%	48%
Long Method	Java	73%	74%	81%	78%	54%	88%	90%	60%	91%	100%	98%	96%	96%	51%	59%	57%	42%	53%	40%	65%	65%	70%
	C#	77%	77%	84%	79%	55%	88%	89%	66%	91%	99%	98%	96%	92%	58%	62%	62%	49%	54%	45%	71%	66%	77%
	C++	74%	74%	87%	79%	40%	88%	89%	71%	93%	98%	98%	97%	73%	49%	58%	62%	43%	52%	46%	65%	65%	77%
	Python	75%	76%	84%	74%	74%	88%	89%	62%	89%	98%	97%	96%	96%	60%	65%	61%	48%	57%	47%	71%	68%	75%
Shotgun Surgery	Java	94%	94%	93%	90%	90%	77%	81%	63%	80%	95%	93%	95%	93%	95%	93%	83%	92%	88%	65%	92%	90%	82%
	C#	92%	91%	92%	87%	94%	77%	79%	69%	82%	95%	94%	93%	93%	93%	95%	86%	91%	91%	71%	91%	92%	83%
	C++	87%	85%	87%	87%	84%	74%	76%	70%	79%	87%	85%	91%	90%	85%	86%	88%	85%	82%	78%	84%	84%	87%
Feature Envy	Java	94%	94%	93%	89%	84%	72%	78%	56%	79%	94%	94%	93%	82%	92%	93%	83%	91%	92%	68%	90%	93%	84%
	C#	95%	94%	94%	92%	94%	74%	80%	63%	79%	95%	94%	93%	92%	94%	93%	83%	91%	96%	67%	93%	93%	87%
	C++	78%	69%	83%	69%	65%	74%	76%	69%	73%	85%	82%	87%	72%	73%	61%	83%	66%	63%	83%	77%	70%	80%
Divergent Change	Java	95%	94%	92%	90%	94%	79%	81%	70%	82%	95%	93%	94%	94%	87%	91%	83%	86%	88%	65%	93%	92%	87%
	C#	93%	92%	92%	88%	78%	79%	83%	66%	84%	97%	96%	94%	91%	90%	90%	79%	88%	89%	61%	93%	92%	85%
	C++	91%	91%	91%	89%	87%	78%	80%	71%	78%	95%	93%	93%	93%	87%	85%	79%	82%	82%	68%	89%	88%	87%

`Complex Method`. When both training and testing datasets contain only the *Complex Method* smell, the trained model reaches the minimum effectiveness of 81%, considering all the programming languages analyzed. The effectiveness is even better when we train the model with *Complex Method* and use the trained model to detect *Long Method*. In such cases, the model effectiveness is at least 94%, regardless of the programming language. When we use this trained

model to detect *God Class*, *Shotgun Surgery*, *Feature Envy*, and *Divergent Change*, the model reaches effectiveness above 70% in most of the cases analyzed. **Such results indicate that models trained over datasets containing only the smell *Complex Method* tend to be effective to detect different smell types regardless of the programming language.**

`God Class`. Different from the *Complex Method*, the model trained over a dataset containing only the smell *God Class* reaches effectiveness below 80% in most of the cases analyzed. Indeed, the trained model reaches effectiveness above 75% only when we evaluate this model in testing datasets containing *God Class* or *Long Method*. However, the effectiveness tends to be low for `Long Method`. When we evaluate the trained model in testing datasets containing *Complex Method*, *Shotgun Surgery*, or *Divergent Change*, the effectiveness reaches only a maximum of 44%. **Such results indicate that the model trained with *God Class* tends to be effective only for detecting God Classes.**

`Long Method`. In the case of the model trained over a dataset containing only `Long Method`, the effectiveness of this model is above 60% when we evaluate it in testing datasets containing `God Class`, `Long Method`, or `Divergent Change`. On the other hand, when we evaluate this model in testing datasets containing only `Feature Envy`, the effectiveness is at a maximum of 57%. Notice also that the evaluation of this model in the testing dataset containing `Complex Method` presents effectiveness above 70% in most of the cases, except when the testing dataset contains only Python code snippets. In such cases, the effectiveness varies between 40% and 74%. **Such results indicate that the model trained with *Long Method* tends to be effective for detecting other smell types, except for *Feature Envy*.**

`Shotgun Surgery`. Regarding the training of the model in a dataset containing only the smell `Shotgun Surgery`, the effectiveness of the trained model is above 80% in most of the cases analyzed, except when we evaluate the model in testing datasets containing the smells `God Class`. Even in this case, the trained model presents effectiveness between 63% and 82%. **Such results indicate that the models trained with *Shotgun Surgery* tend to be effective for detecting other code smell types regardless of the programming language.**

`Feature Envy`. In the case of the model trained in a dataset containing only the smell `Feature Envy`, the model reaches a minimum effectiveness of 60% in all the cases analyzed. The model reaches effectiveness above 90% in most cases analyzed, except when we apply it in a testing dataset containing only God Class. In such a case, the trained model reaches a maximum effectiveness of 80%. **Such results indicate that the model trained with *Feature Envy* tend to be effective in detecting other code smell types regardless of the programming language.**

`Divergent Change`. The mode trained in datasets containing only `Divergent Change` reaches effectiveness at least 71% in all the cases analyzed, except when we evaluate the model in a dataset containing only `Feature Envy`. However, in such a case, the trained model reaches the minimum effectiveness of 61%. **Such results indicate that the model trained with *Divergent Change* tends to be effective in detecting other code smell types regardless of the programming language.**

Summary of RQ1: Transfer learning tends to effectively detect different smell types analyzed regardless of the programming language and the smell type of the training dataset, except when we train the model with `God Class`.

RQ2: How efficient is transfer learning to detect code smells?

So far, we evidenced that transfer learning detects code smells effectively (RQ1). However, we still do not know the actual effort needed to research this effectiveness. The smaller the sample size is, the lower the effort to build the training dataset. Thus, to answer RQ2, we analyze the minimum sample size required by transfer learning to reach models with similar effectiveness to those found for a sample size=1000 (RQ1). Table 9 presents the effectiveness (in terms of *f-measure*) of Perceptron models as we increase the number of instances (sample size) from 32 to 1000 composing the training datasets.

Table 9 – F-measure of Perceptron models trained with increasing number of samples

		Samples = 32					**Samples = 64**					**Samples = 128**				
		Java	C#	C++	JavaScript	Python	Java	C#	C++	JavaScript	Python	Java	C#	C++	JavaScript	Python
Complex Method	Java	93%	93%	93%	90%	91%	95%	95%	93%	91%	94%	95%	95%	94%	91%	94%
	C#	91%	89%	92%	87%	84%	92%	91%	92%	88%	84%	94%	94%	93%	89%	87%
	C++	91%	90%	91%	89%	89%	94%	92%	93%	90%	91%	95%	94%	95%	92%	94%
	JavaScript	89%	88%	91%	89%	86%	93%	91%	93%	91%	93%	93%	92%	92%	93%	94%
	Python	92%	90%	92%	89%	91%	92%	90%	92%	89%	91%	92%	91%	91%	88%	95%
God Class	Java	90%	86%	37%	-	85%	93%	88%	38%	-	91%	94%	89%	43%	-	88%
	C#	93%	90%	50%	-	94%	91%	90%	58%	-	92%	94%	92%	58%	-	68%
	C++	78%	83%	78%	-	81%	80%	86%	77%	-	86%	84%	87%	76%	-	88%
	Python	94%	89%	43%	-	94%	93%	90%	46%	-	94%	94%	90%	48%	-	94%
Long Method	Java	97%	96%	95%	-	87%	98%	98%	96%	-	97%	99%	98%	96%	-	96%
	C#	97%	98%	96%	-	95%	98%	98%	97%	-	97%	98%	98%	97%	-	91%
	C++	96%	96%	96%	-	91%	97%	97%	96%	-	77%	97%	97%	96%	-	87%
	Python	98%	98%	96%	-	96%	98%	98%	96%	-	96%	98%	98%	96%	-	97%
Shotgun Surgery	Java	92%	91%	87%	-	-	92%	91%	87%	-	-	93%	92%	87%	-	-
	C#	90%	88%	82%	-	-	92%	92%	84%	-	-	93%	94%	85%	-	-
	C++	92%	91%	86%	-	-	89%	88%	89%	-	-	88%	87%	87%	-	-
Feature Envy	Java	88%	87%	72%	-	-	91%	89%	68%	-	-	92%	91%	69%	-	-
	C#	87%	87%	70%	-	-	88%	87%	69%	-	-	90%	90%	71%	-	-
	C++	89%	87%	77%	-	-	86%	85%	78%	-	-	81%	80%	79%	-	-
		Samples = 256					**Samples = 512**					**Samples = 1000**				
		Java	C#	C++	JavaScript	Python	Java	C#	C++	JavaScript	Python	Java	C#	C++	JavaScript	Python
Complex Method	Java	96%	95%	95%	92%	94%	97%	96%	96%	92%	90%	98%	96%	96%	93%	91%
	C#	97%	97%	95%	91%	91%	96%	97%	96%	92%	86%	98%	98%	96%	93%	92%
	C++	97%	95%	95%	94%	94%	95%	94%	96%	94%	89%	96%	97%	96%	93%	94%
	JavaScript	94%	93%	92%	95%	92%	95%	95%	93%	96%	87%	97%	96%	95%	95%	94%
	Python	85%	85%	87%	82%	96%	85%	88%	90%	84%	97%	82%	84%	90%	81%	97%
God Class	Java	94%	90%	42%	-	91%	94%	88%	42%	-	82%	94%	89%	42%	-	53%
	C#	95%	93%	51%	-	67%	93%	92%	44%	-	57%	94%	92%	46%	-	53%
	C++	87%	85%	80%	-	76%	83%	80%	82%	-	54%	51%	40%	81%	-	24%
	Python	93%	89%	43%	-	94%	94%	89%	46%	-	95%	93%	88%	48%	-	96%
Long Method	Java	99%	98%	96%	-	95%	99%	98%	96%	-	97%	100%	98%	96%	-	96%
	C#	99%	98%	96%	-	94%	99%	98%	96%	-	94%	99%	98%	96%	-	92%
	C++	99%	98%	98%	-	88%	99%	98%	97%	-	85%	98%	98%	97%	-	73%
	Python	98%	97%	97%	-	97%	97%	97%	96%	-	97%	98%	97%	96%	-	96%
Shotgun Surgery	Java	93%	92%	86%	-	-	95%	94%	86%	-	-	95%	93%	83%	-	-
	C#	93%	94%	85%	-	-	93%	93%	86%	-	-	93%	95%	86%	-	-
	C++	87%	85%	87%	-	-	87%	87%	88%	-	-	85%	86%	88%	-	-
Feature Envy	Java	92%	89%	69%	-	-	92%	90%	69%	-	-	91%	92%	68%	-	-
	C#	91%	92%	71%	-	-	92%	95%	72%	-	-	91%	96%	67%	-	-
	C++	73%	72%	82%	-	-	54%	51%	84%	-	-	66%	63%	83%	-	-

`Complex Method`. When we train the model over training datasets containing only `Complex Methods`, we observe that the trained models reach high effectiveness (from 81% to 98%) regardless of the programming language and the sample size. **This result suggests that a trained dataset with 32 instances is sufficient to build models that can effectively detect Complex Methods with transfer learning.**

`God Class`. For God Class, we observe a high variation in the models' effectiveness not influenced by the sample size but by the programming language. Indeed, the lowest effectiveness scores are obtained by the Python and C++ models (24% to 96%), while all the scores for Java and C# models are high (80% to 95%) **These results indicate that the sample size does not affect the effectiveness in detecting God Classes with transfer learning.**

`Long Method`. We observe that all the trained models reached effectiveness higher than 95%, except for the C++ model on detecting long methods in Python (73% to 91%). However, even in this case, we cannot observe an improvement in the sample size over the effectiveness. **This result suggests that a trained dataset with 32 instances is sufficient to build transfer learning models that can effectively detect Long Methods in different programming languages.**

`Shotgun Surgery`. When we train the model over training datasets containing only `Shotgun Surgery`, we observe that the trained models reach high effectiveness (from 82% to 95%) regardless of the programming language and the sample size. **This result suggests that a trained dataset with 32 instances is sufficient to build transfer learning models that can effectively detect `Shotgun Surgery`. in different programming languages**

`Feature Envy`. The model reaches effectiveness above 70% in most cases, except in a few cases where we train or evaluate models in C++. **Although we could not identify a positive influence of the sample size over the models' effectiveness, it is important to note that the model in C++ reaches lower effectiveness with higher sample sizes (256 or higher).**

Summary of RQ2: Transfer learning is an efficient way to detect different types of code smells in different programming languages. Considering our experiment settings, sample sizes=32 are sufficient to compose the training datasets.

RQ3: How many layers must a deep learning architecture have to effectively detect code smells?

In this research question, we evaluate how complex, i.e., how many layers the neural network architecture should be for reaching higher levels of effectiveness in detecting code smells. For this purpose, we evaluate the effectiveness and efficiency of transfer learning supported by CNN in detecting code smells. Then, we compare the effectiveness and efficiency of transfer learning between CNN and Perceptron.

`Effectiveness.` As discussed in the **RQ1** results, the Perceptron model effectively detects code smells regardless of the programming language used to train or evaluate the model. Table 10 describes the effectiveness of the CNN model in detecting code smells in different programming languages. We observe that the CNN model reaches effectiveness values close to the Perceptron. In `Complex Method`, both CNN and Perceptron models reach effectiveness above 70% in most of the cases analyzed. Regarding the `God Class`, these models reach effectiveness below 80% in most of the cases analyzed. For `Long Method`, the CNN model presents an effectiveness slightly greater than the Perceptron model. While the CNN model presents maximum effectiveness above 60% in all the smell types analyzed, the Perceptron model presents effectiveness above 60% only when evaluated in testing datasets containing `God Class`, `Long Method`, and `Divergent Change`. On the other hand, the Perceptron model presents a minimum effectiveness slightly greater than CNN model in these cases. While the CNN model presents a minimum effectiveness varying between 59% and 67%, the Perceptron model presents a minimum effectiveness varying between 63% and 85%. Concerning the `Feature Envy`, both the CNN and Perceptron models present a minimum effectiveness of 60% in most cases analyzed. The only exception is when we evaluate the CNN model to detect `Feature Envy` in Python. In such a case, the model presents a minimum effectiveness of 37%. Regarding the `Divergent Change`, both the CNN and Perceptron models present an effectiveness of at least 66%. **The results indicate that the complexity of the model does not lead to a significant difference in the models' effectiveness for detecting different types of code smells with transfer learning, regardless of the programming language.**

Table 10 – Effectiveness of Transfer Learning using CNN model

		Complex Method					God Class				Long Method				Shotgun Surgery			Feature Envy			Divergent Change		
		Java	C#	C++	JavaScript	Python	Java	C#	C++	Python	Java	C#	C++	Python	Java	C#	C++	Java	C#	C++	Java	C#	C++
Complex Method	Java	94%	96%	95%	94%	95%	85%	85%	72%	84%	97%	98%	96%	94%	80%	81%	73%	68%	79%	64%	89%	84%	84%
	C#	96%	98%	96%	96%	93%	86%	85%	66%	85%	97%	98%	95%	94%	83%	82%	74%	75%	83%	58%	88%	89%	83%
	C++	92%	98%	97%	92%	94%	81%	85%	43%	84%	94%	97%	95%	93%	72%	85%	69%	61%	82%	53%	82%	89%	79%
	JavaScript	97%	97%	93%	97%	94%	85%	84%	74%	82%	97%	97%	95%	95%	86%	87%	82%	76%	84%	69%	91%	91%	85%
	Python	83%	85%	91%	83%	97%	86%	85%	69%	84%	98%	96%	96%	95%	69%	66%	70%	53%	60%	57%	75%	71%	77%
God Class	Java	21%	22%	27%	33%	6%	93%	87%	42%	71%	42%	48%	67%	12%	8%	9%	25%	3%	10%	15%	14%	26%	34%
	C#	31%	36%	49%	55%	14%	93%	93%	64%	87%	65%	70%	83%	41%	13%	21%	36%	7%	23%	20%	23%	33%	51%
	C++	15%	32%	28%	50%	43%	83%	73%	81%	86%	28%	41%	48%	46%	5%	10%	28%	0%	15%	21%	15%	15%	39%
	Python	39%	44%	55%	58%	23%	95%	91%	59%	95%	78%	85%	89%	54%	19%	29%	41%	11%	26%	27%	31%	39%	57%
Long Method	Java	84%	82%	86%	84%	74%	87%	89%	72%	89%	98%	99%	96%	97%	57%	60%	62%	45%	58%	50%	70%	70%	77%
	C#	81%	76%	85%	81%	59%	90%	88%	72%	92%	98%	98%	96%	92%	52%	58%	62%	41%	53%	50%	67%	63%	79%
	C++	81%	76%	86%	81%	45%	89%	89%	69%	92%	97%	97%	97%	74%	51%	60%	60%	44%	57%	43%	68%	70%	74%
	Python	81%	81%	88%	81%	77%	86%	88%	67%	89%	99%	98%	97%	97%	66%	67%	66%	53%	61%	51%	77%	70%	78%
Shotgun Surgery	Java	90%	94%	95%	90%	95%	81%	81%	59%	82%	95%	94%	94%	92%	96%	94%	85%	93%	92%	67%	94%	94%	82%
	C#	82%	91%	90%	82%	94%	66%	68%	64%	70%	93%	93%	91%	91%	93%	94%	84%	91%	92%	69%	91%	93%	80%
	C++	66%	67%	67%	66%	67%	67%	67%	67%	67%	67%	67%	67%	67%	67%	67%	67%	67%	67%	67%	67%	67%	67%
Feature Envy	Java	95%	94%	92%	89%	95%	78%	81%	66%	81%	96%	94%	94%	93%	93%	95%	85%	92%	95%	72%	93%	93%	86%
	C#	67%	67%	67%	66%	67%	67%	66%	67%	67%	67%	67%	66%	67%	67%	67%	67%	67%	67%	67%	67%	67%	67%
	C++	78%	77%	82%	73%	37%	73%	78%	69%	77%	88%	88%	87%	60%	68%	63%	81%	68%	70%	85%	74%	80%	84%
Divergent Change	Java	89%	94%	89%	89%	90%	76%	80%	68%	80%	96%	93%	91%	90%	88%	89%	86%	84%	89%	75%	94%	93%	87%
	C#	94%	95%	93%	88%	93%	79%	83%	69%	79%	96%	96%	94%	94%	90%	88%	81%	85%	88%	66%	94%	91%	85%
	C++	90%	88%	90%	86%	87%	75%	76%	69%	77%	95%	89%	93%	86%	86%	86%	82%	85%	87%	73%	93%	89%	85%

`Efficiency.` As discussed in the **RQ2**, the Perceptron model efficiently detects code smells regardless of the programming language and sample size. Table 11 presents the efficiency of the CNN models to detect code smells in different programming languages as we increase the sample size.

For `Complex Methods`, we observe that the trained models reach high effectiveness (from 81% to 97%) regardless of the programming language and the sample size. For `God Class`,

Table 11 – F-measure of CNN models trained with increasing number of samples

		Samples = 32					**Samples = 64**					**Samples = 128**				
		Java	C#	C++	JavaScript	Python	Java	C#	C++	JavaScript	Python	Java	C#	C++	JavaScript	Python
Complex Method	Java	93%	93%	94%	91%	93%	96%	96%	95%	92%	91%	96%	95%	94%	92%	93%
	C#	67%	67%	66%	66%	67%	93%	92%	92%	89%	82%	97%	96%	95%	92%	91%
	C++	91%	89%	91%	89%	91%	91%	89%	90%	87%	90%	94%	93%	92%	88%	91%
	JavaScript	85%	87%	90%	89%	86%	92%	89%	92%	91%	93%	95%	93%	93%	93%	95%
	Python	67%	67%	67%	67%	67%	92%	91%	92%	88%	92%	93%	92%	92%	90%	95%
God Class	Java	92%	91%	55%	-	94%	93%	89%	45%	-	87%	94%	90%	45%	-	93%
	C#	93%	91%	53%	-	94%	92%	91%	63%	-	90%	93%	92%	59%	-	69%
	C++	70%	70%	72%	-	67%	83%	86%	73%	-	86%	88%	88%	78%	-	89%
	Python	92%	90%	50%	-	94%	93%	90%	50%	-	94%	91%	90%	62%	-	93%
Long Method	Java	97%	97%	95%	-	91%	98%	97%	96%	-	97%	99%	98%	96%	-	91%
	C#	96%	94%	95%	-	95%	96%	95%	95%	-	95%	98%	98%	96%	-	96%
	C++	97%	95%	96%	-	73%	96%	95%	96%	-	86%	96%	96%	96%	-	86%
	Python	98%	97%	96%	-	97%	98%	98%	96%	-	97%	98%	97%	97%	-	97%
Shotgun Surgery	Java	88%	87%	89%	-	-	0%	0%	3%	-	-	94%	93%	87%	-	-
	C#	67%	67%	66%	-	-	92%	90%	89%	-	-	0%	0%	0%	-	-
	C++	90%	88%	88%	-	-	85%	84%	88%	-	-	88%	87%	87%	-	-
Feature Envy	Java	89%	89%	70%	-	-	91%	89%	61%	-	-	67%	67%	67%	-	-
	C#	88%	87%	72%	-	-	89%	87%	70%	-	-	91%	92%	73%	-	-
	C++	84%	81%	80%	-	-	83%	80%	75%	-	-	78%	75%	78%	-	-
		Samples = 256					**Samples = 512**					**Samples = 1000**				
		Java	C#	C++	JavaScript	Python	Java	C#	C++	JavaScript	Python	Java	C#	C++	JavaScript	Python
Complex Method	Java	0%	0%	0%	1%	0%	98%	97%	95%	93%	95%	98%	96%	95%	94%	95%
	C#	97%	96%	94%	89%	93%	97%	97%	96%	93%	93%	98%	98%	96%	96%	93%
	C++	98%	96%	96%	92%	94%	95%	96%	96%	94%	95%	97%	98%	97%	92%	94%
	JavaScript	96%	95%	94%	95%	83%	96%	95%	94%	96%	90%	97%	97%	93%	97%	94%
	Python	80%	83%	88%	81%	97%	87%	92%	91%	88%	97%	84%	85%	91%	83%	97%
God Class	Java	93%	87%	43%	-	91%	92%	89%	61%	-	94%	93%	87%	42%	-	71%
	C#	93%	94%	50%	-	42%	94%	94%	48%	-	25%	93%	93%	64%	-	87%
	C++	91%	88%	81%	-	78%	80%	76%	81%	-	55%	83%	73%	81%	-	86%
	Python	94%	90%	49%	-	95%	93%	91%	53%	-	95%	95%	91%	59%	-	95%
Long Method	Java	99%	98%	96%	-	88%	99%	98%	96%	-	97%	98%	99%	96%	-	97%
	C#	99%	99%	96%	-	91%	99%	98%	96%	-	94%	98%	98%	96%	-	92%
	C++	99%	97%	97%	-	97%	98%	98%	97%	-	97%	97%	97%	97%	-	74%
	Python	98%	98%	96%	-	97%	98%	97%	96%	-	97%	99%	98%	97%	-	97%
Shotgun Surgery	Java	94%	92%	86%	-	-	0%	0%	1%	-	-	96%	94%	85%	-	-
	C#	94%	94%	84%	-	-	67%	67%	67%	-	-	93%	94%	84%	-	-
	C++	85%	80%	85%	-	-	0%	0%	0%	-	-	67%	67%	67%	-	-
Feature Envy	Java	93%	92%	72%	-	-	92%	89%	62%	-	-	92%	95%	72%	-	-
	C#	92%	93%	73%	-	-	91%	95%	72%	-	-	67%	67%	67%	-	-
	C++	70%	64%	79%	-	-	45%	37%	83%	-	-	68%	70%	85%	-	-

we observe a high variation in the models' effectiveness (42% to 94%) not influenced by the sample size but by the programming language. For `Long Method`, all the trained models reached effectiveness higher than 94%, reaching high effectiveness regardless of the sample size and the programming language. For `Shotgun Surgery`, we observed worst results for CNN than for Perceptron, ranging from 0% to 96% without a clear influence of the programming language. However, one may see that better results are reached with the sample size=256. For `Feature Envy`, the model reaches effectiveness above 70% in most cases, except in a few cases where we train or evaluate models trained in C++. **Although we could not identify a positive influence of the sample size over the models' effectiveness, it is important to note that the model in C++ reaches lower effectiveness with higher sample sizes (256 or higher).** Therefore, similar to Perceptron, the results show that a small sample size of instances (32) is sufficient for training CNN models for detecting Complex Method, God Class, Long Method, and Feature Envy. However, different from Perceptron, CNN requires higher sample sizes (256) to reach better effectiveness in detecting Shotgun Surgery.

Summary of RQ3: The Perceptron and CNN models effectively detect code smells with small samples, but the Perceptron presents an efficiency slightly better than CNN.

5 Discussion

The findings of our study reveal that transfer learning is an effective and efficient approach for leveraging the software development community to expand their possibilities of detecting code smells in different programming languages through the reuse of already existing resources. The study findings also reveal interesting aspects and trends in using transfer learning that may contribute to the software engineering community optimizing future research efforts, discussed in the following paragraphs.

Genetic matters. Despite programming languages such as C++, Java, C#, and JavaScript may be frequently applied for different purposes, they share basic flow and conditional structures. These characteristics probably facilitate deep learning algorithms to identify the same patterns after the tokenization process. Future investigations in transfer learning may explore these similarities, for instance, to investigate the effectiveness of transfer learning in detecting more complex smell types.

Small sample size. One of the main results of our study is that large training datasets are not necessary for effectively detecting code smells in different programming languages. Indeed, the models trained from the Perceptron and CNN architectures present high effectiveness even if we consider a sample size of 32 instances. We believe that the technique of tokenization used to represent the code snippets has a significant contribution to obtaining these results. This technique enables us to extract information from the source code that is not easy to do the same by representing the code only through metrics.

Keep it as simple (as possible). The experience of our study reveals that it is worthwhile to start the evaluation of transfer learning on code smell detection from the simplest deep learning architectures. From the perspective of deep learning, code smell detection is a relatively simple problem. Our study suggests that Perceptron is an effective architecture for detecting several smell types. However, we should also keep in mind that detecting more complex types of code smells may require different settings.

6 Related Work

Different techniques have been proposed to detect code smells automatically by using a variety of methods: (i) metrics; (ii) rules/heuristics; (iii) history; (iv) optimization; and most recently, (v) machine learning. The abstract syntax tree (AST) is commonly used in metric-based methods for computing relevant metrics of the source code. Once the thresholds of these metrics are established, they are employed for detecting code smells. However, some code smells can not be detected only using metrics, such as rebellious hierarchy, missing abstraction, cyclic hierarchy, and empty catch block. For this purpose, heuristic-based methods are proposed. In such methods, simple rules or complex heuristics are defined according to the code smell type. History-based methods analyze the evolution of source code to spot code smells. The optimization-based methods are supported by optimization algorithms such as genetic algorithms.

Most recently, machine learning-based models emerged as promising method for code smell detection. For training these models, machine learning algorithms such as Support Vector Machines, Bayesian Belief Networks, and Logistic Regression have been employed. (PECORELLI et al., 2019) present comparison between the use of the *DECOR* heuristic and a machine learning model based on the *Naive Bayes* algorithm. Both methods performed poorly. This study mentions the fact that the dataset has a wide variety of code smells and has real examples (i.e., manually validated) as a possible cause of low performance, indicating that the success of these methods can be related to the dataset and the results may be unsuitable for use in practice. On the other hand, the study by (OLIVEIRA et al., 2020) evaluated the detection of six code smells by seven machine learning algorithms. The results indicate that overall the models reach their best accuracy with few samples, indicating the importance of investigating sample size in depth with transfer learning.

Besides the more recent dissemination of machine learning models, there are several code smell detection tools available. These tools are predominantly based on metrics and rules/heuristics. (PAIVA et al., 2017) compare four tools for code smells detection and present supported programming languages, code smells types coverage, advantages, and disadvantages. For instance, the *InFunsion* is a commercial tool supporting the detection of 22 different types of code smells in Java, C, and C++ programs. *JDeodorant* is an open-source *plugin* for *Eclipse*[1], supporting the detection of four types of code smells in Java programs. Most of these tools support code smell detection in Java programs. Among them, two support C/C++ programming languages, while only *PMD* can detect code smells in other programming languages.

The feasibility of applying transfer learning based on deep learning models for code smells detection was investigated in previous work (SHARMA et al., 2021). In this work, the authors focused on four types of code smells (complex method, empty catch block, magic number, and multifaceted abstraction), evaluating the effectiveness of transfer learning between datasets

[1] www.eclipse.org

composed of source code written in C# and Java. The source code used in the study was obtained from open-source projects available at GitHub. The authors conducted the study in two steps: (i) evaluating the use of deep learning models for code smells detection in C# and Java, and (ii) evaluating transfer learning using pre-trained models between C# and Java. The study involved two architectures of neural networks, Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN). The study findings indicate the feasibility of deep learning for code smells detection and transfer learning for detecting code smells in Java through models trained with C# datasets. However, both models obtained considerably lower results for detecting the multifaceted abstraction smell.

Beyond the context of code smell detection, there are other works investigating the use of transfer learning in the field. For instance, transfer learning has been used for code autocompletion (ZHOU et al., 2021) and automatic repair of software vulnerabilities (CHEN; KOMMRUSCH; MONPERRUS, 2022).

7 Threats to validity

Internal validity. We use more than 4,000 code snippets automatically analyzed through heuristics without following manual validation, which can pose a threat to internal validity. This is a common challenge in large-scale studies addressing deep learning solutions in code smell detection. However, one may see that our research goal is evaluating the feasibility of transfer learning models replicating the detection behavior observed in the training datasets, which aligns with our internal validity goals. Besides, it is important to note that we used common types of code smell, which detection is addressed by several detection tools through similar detection rules. In this way, we made an effort to build a standard and optimized detection by combining these rules. Moreover, it is questionable whether performing manual validation over large datasets without a proper analysis of the projects' characteristics effectively contributes to composing oracles (HOZANO et al., 2017; HOZANO et al., 2018; JUNIONELLO; MELLO, 2021; MELLO et al., 2022).

Another threat to validity addresses the limitations of the parsing tools employed. We need to perform our own parsing solution for Python and JavaScript. For this purpose, we made efforts to identify and reuse reliable and stable tools as background for implementing these solutions. We also tested our solution over several code elements from both programming languages before running the study.

External validity. One common threat to the validity, both internal and external, of studies based on mining software repositories addresses the representativeness of the software projects and the code snippets analyzed. To mitigate this threat to external validity, we apply rigorous criteria for composing our datasets. We select relevant releases from popular open-source projects addressing different technologies and domains for each programming language, aiming to enhance the generalizability of our findings. Additionally, we reuse the datasets from previous work (SHARMA et al., 2021) for the programming languages available to ensure comparability and external validity.

8 Conclusions

The purpose of our research was to identify a low-cost and effective alternative based on transfer learning for supporting the detection of code smells in open source projects from different programming languages. We investigated the effectiveness and efficiency of transfer learning for detecting code smells. Also, we analyzed how complex should be the deep learning architecture to effectively and efficiently detect code smells.

For this purpose, we obtained pre-trained models based on Perceptron and Convolutional Neural Network (CNN) for detecting six smell types existing in projects from the programming languages Java, C#, C++, Python, and JavaScript. The results indicated that transfer learning is effective and efficient in detecting code smells in different programming languages. Also, the results suggest that it is not necessary to use complex deep learning architectures and large sample sizes to train models for effectively and efficiently detecting code smells.

8.1 Future Work

In future work, we intend to:

- Extend the investigation with other programming languages;
- Investigate other code smells with different levels of complexity;
- Explore different approaches to represent code snippets.

Bibliography

AZEEM, M. I. et al. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, Elsevier, v. 108, p. 115–138, 2019. Citado na página 9.

BIBIANO, A. C. et al. Look ahead! revealing complete composite refactorings and their smelliness effects. In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2021. p. 298–308. Citado 2 vezes nas páginas 9 and 12.

BIBIANO, A. C. et al. A quantitative study on characteristics and effect of batch refactoring on code smells. In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.: s.n.], 2019. p. 1–11. Citado na página 9.

CAIRO, A. S.; CARNEIRO, G. d. F.; MONTEIRO, M. P. The impact of code smells on software bugs: A systematic literature review. *Information (Switzerland)*, v. 9, n. 11, p. 1–22, 2018. ISSN 20782489. Citado na página 12.

CEDRIM, D. et al. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2017. (ESEC/FSE 2017), p. 465–475. ISBN 9781450351058. Disponível em: <https://doi.org/10.1145/3106237.3106259>. Citado 2 vezes nas páginas 12 and 19.

CHEN, Z.; KOMMRUSCH, S. J.; MONPERRUS, M. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. *IEEE Transactions on Software Engineering*, p. 1–20, 2022. ISSN 19393520. Citado na página 34.

Dhankhad, S.; Mohammed, E.; Far, B. Supervised machine learning algorithms for credit card fraudulent transaction detection: A comparative study. In: *2018 IEEE International Conference on Information Reuse and Integration (IRI)*. [S.l.: s.n.], 2018. p. 122–125. ISSN null. Citado na página 9.

FERNANDES, E. et al. A review-based comparative study of bad smell detection tools. In: *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. [S.l.: s.n.], 2016. p. 1–12. Citado na página 9.

FOWLER, M. et al. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. (Addison-Wesley object technology series). ISBN 9780201485677. Disponível em: <https://books.google.com.br/books?id=1MsETFPD3I0C>. Citado na página 12.

GALVEZ, R. L. et al. Object detection using convolutional neural networks. In: *TENCON 2018 - 2018 IEEE Region 10 Conference*. [S.l.: s.n.], 2018. p. 2023–2027. Citado na página 15.

HOZANO, M. et al. Smells are sensitive to developers! on the efficiency of (un)guided customized detection. In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. [S.l.: s.n.], 2017. p. 110–120. Citado 2 vezes nas páginas 9 and 35.

HOZANO, M. et al. Are you smelling it? investigating how similar developers detect code smells. *Information and Software Technology*, v. 93, p. 130–146, 2018. ISSN 0950-5849.

Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584916303901>. Citado 2 vezes nas páginas 9 and 35.

HUSSAIN, M.; BIRD, J. J.; FARIA, D. R. A study on cnn transfer learning for image classification. In: LOTFI, A. et al. (Ed.). *Advances in Computational Intelligence Systems*. Cham: Springer International Publishing, 2019. p. 191–202. ISBN 978-3-319-97982-3. Citado na página 15.

Hussain, R.; Zeadally, S. Autonomous cars: Research results, issues, and future challenges. *IEEE Communications Surveys Tutorials*, v. 21, n. 2, p. 1275–1313, Secondquarter 2019. ISSN 2373-745X. Citado na página 9.

IAMMARINO, M. et al. Self-admitted technical debt removal and refactoring actions: Co-occurrence or more? In: IEEE. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2019. p. 186–190. Citado na página 9.

JEBNOUN, H. et al. The Scent of Deep Learning Code: An Empirical Study. *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020*, p. 420–430, 2020. Citado na página 9.

JUNIONELLO, L. F.; MELLO, R. de. Towards heuristics for supporting the validation of code smells. *Proceedings of the XXIV Iberoamerican Conference on Software Engineering*, p. 276–289, 2021. Citado na página 35.

KOUROU, K. et al. Machine learning applications in cancer prognosis and prediction. *Computational and Structural Biotechnology Journal*, v. 13, p. 8 – 17, 2015. ISSN 2001-0370. Disponível em: <http://www.sciencedirect.com/science/article/pii/S2001037014000464>. Citado na página 9.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. In: PEREIRA, F. et al. (Ed.). *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2012. v. 25. Disponível em: <https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf>. Citado na página 16.

LECUN, Y. et al. LeNet. *Proceedings of the IEEE*, n. November, p. 1–46, 1998. ISSN 00189219. Citado 2 vezes nas páginas 6 and 16.

LIMA, R. et al. Understanding and detecting harmful code. In: *Proceedings of the 34th Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2020. p. 223–232. Citado na página 9.

LIU, H. et al. Deep Learning Based Code Smell Detection. *IEEE Transactions on Software Engineering*, v. 5589, n. c, p. 1–28, 2019. ISSN 19393520. Citado na página 9.

MACCAGNO, A. et al. A cnn approach for audio classification in construction sites. In: ____. *Progresses in Artificial Intelligence and Neural Systems*. Singapore: Springer Singapore, 2021. p. 371–381. ISBN 978-981-15-5093-5. Disponível em: <https://doi.org/10.1007/978-981-15-5093-5_33>. Citado na página 15.

MCCABE, T. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2, n. 4, p. 308–320, 1976. Citado na página 21.

MCCULLOCH, W.; PITTS, W. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, v. 5, p. 127–147, 1943. Citado na página 15.

MELLO, R. de et al. Recommendations for developers identifying code smells. *IEEE Software*, IEEE Computer Society, n. 01, p. 2–10, 2022. Citado na página 35.

MELLO, R. de et al. Do research and practice of code smell identification walk together? a social representations analysis. In: IEEE. *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.], 2019. p. 1–6. Citado na página 9.

MELLO, R. M. de; OLIVEIRA, R.; GARCIA, A. On the influence of human factors for identifying code smells: A multi-trial empirical study. In: IEEE. *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.], 2017. p. 68–77. Citado na página 9.

OIZUMI, W. et al. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In: IEEE. *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. [S.l.], 2016. p. 440–451. Citado na página 9.

OLIVEIRA, D. et al. Applying machine learning to customized smell detection: A multi-project study. In: *Proceedings of the 34th Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2020. p. 233–242. Citado 2 vezes nas páginas 9 and 33.

PAIVA, T. et al. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development*, Journal of Software Engineering Research and Development, v. 5, n. 1, p. 1–28, 2017. ISSN 2195-1721. Citado 2 vezes nas páginas 9 and 33.

PAN, S. J.; YANG, Q. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, IEEE, v. 22, n. 10, p. 1345–1359, 2010. ISSN 10414347. Citado 3 vezes nas páginas 6, 10, and 17.

PECORELLI, F. et al. Comparing heuristic and machine learning approaches for metric-based code smell detection. *IEEE International Conference on Program Comprehension*, IEEE, v. 2019-May, p. 93–104, 2019. Citado 2 vezes nas páginas 9 and 33.

SHARMA, T. et al. *Code smell detection by deep direct-learning and transfer-learning*. 2021. Citado 5 vezes nas páginas 12, 17, 21, 33, and 35.

SHARMA, T.; SPINELLIS, D. A survey on software smells. *Journal of Systems and Software*, Elsevier Inc., v. 138, p. 158–173, 2018. ISSN 01641212. Disponível em: <https://doi.org/10.1016/j.jss.2017.12.034>. Citado 3 vezes nas páginas 9, 10, and 12.

SHRESTHA, A.; MAHMOOD, A. Review of deep learning algorithms and architectures. *IEEE Access*, IEEE, v. 7, n. c, p. 53040–53065, 2019. ISSN 21693536. Citado na página 15.

SPINELLIS, D. *dspinellis/tokenizer: Version 1.1*. Zenodo, 2019. Https://github.com/dspinellis/tokenizer/. Disponível em: <https://doi.org/10.5281/zenodo.2558420>. Citado na página 21.

Sumanth Bathula. *Single Layer Perceptron*. 2018. [Online; accessed June 12, 2023]. Disponível em: <http://sumanthrb.com/wp-content/uploads/2019/05/SLP.jpg>. Citado 2 vezes nas páginas 6 and 15.

SuperAnnotate. *Convolutional Neural Networks: 1998-2023 Overview*. 2023. [Online; accessed June 12, 2023]. Disponível em: <https://uploads-ssl.webflow.com/614c82ed388d53640613982e/6462188f86bb3724db70a0c5_convolutional%20layer.webp>. Citado 2 vezes nas páginas 6 and 16.

SZEGEDY, C. et al. Going deeper with convolutions. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, v. 07-12-June-2015, p. 1–9, 2015. ISSN 10636919. Citado na página 16.

UCHÔA, A. et al. How does modern code review impact software design degradation? an in-depth empirical study. In: IEEE. *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2020. p. 511–522. Citado na página 9.

WEISS, K.; KHOSHGOFTAAR, T. M.; WANG, D. D. *A survey of transfer learning*. [S.l.]: Springer International Publishing, 2016. v. 3. ISSN 21961115. ISBN 4053701600. Citado na página 17.

ZAZWORKA, N. et al. Comparing four approaches for technical debt identification. *Software Quality Journal*, Springer, v. 22, n. 3, p. 403–426, 2014. Citado na página 9.

ZHOU, W. et al. Improving code autocompletion with transfer learning. *arXiv preprint arXiv:2105.05991*, 2021. Citado na página 34.

ZHUANG, F. et al. A comprehensive survey on transfer learning. *arXiv*, 2019. Citado na página 17.