EnseSmells: Deep ensemble and programming language models for automated code smells detection

Anh Ho[a], Anh M. T. Bui[a,*], Phuong T. Nguyen[b], Amleto Di Salle[c]

[a]*Hanoi University of Science and Technology, Vietnam*
anh.h190037@sis.hust.edu.vn, anhbtm@soict.hust.edu.vn
[b]*Università degli studi dell'Aquila, 67100 L'Aquila, Italy*
phuong.nguyen@univaq.it
[c]*Gran Sasso Science Institute, Italy*
amleto.disalle@gssi.it

Abstract

A smell in software source code denotes an indication of sub-optimal design and implementation decisions, potentially hindering the code understanding and, in turn, raising the likelihood of being prone to changes and faults. Identifying these code issues at an early stage in the software development process can mitigate these problems and enhance the overall quality of the software. Current research primarily focuses on the utilization of deep learning-based models to investigate the semantic information concealed within source code instructions to detect code smells, with limited attention given to the importance of structural and design-related features. This paper proposes a novel approach to code smell detection based on a deep learning model that places importance on the fusion of design-related features and source code embedding techniques. We further provide a thorough analysis of how different source code embedding models affect the detection performance with respect to different code smell types. We conducted an empirical investigation employing four widely-used code smell benchmark datasets. Our findings demonstrate that the inclusion of design-related features significantly enhances the detection accuracy for various code smell types, comparing to the state-of-the art works.

1. Introduction

Code smell (Fowler, 1999) is the term used to discernible traits or recurring patterns in software source code, indicating probable deficiencies or regions warranting enhancement. Code smells differ

*Corresponding author

from conventional software bugs or errors in that they do not manifest as malfunctions; instead, they function as indications of latent design or implementation quandaries with the potential to precipitate issues in subsequent phases of development and maintenance. Identifying and addressing code smells is an important part of code review and maintenance, as it can help minimize technical debt and make code easier to understand, modify and extend.

Various studies have been undertaken to investigate the smell metaphor in software engineering. These studies includes comprehensive elucidations of different kinds of code smells (Zhang et al., 2011; Singh and Kaur, 2018), methodologies for detecting such smells (Zhang et al., 2022; Sharma et al., 2021) as well as quantifiable metrics that may serve as indicators for discerning bad smells (Marinescu, 2005; Sharma et al., 2016). Software metrics (Lanza and Marinescu, 2007) have been widely studied to identify design defects in the source code (Marinescu, 2004; Munro, 2005; Van Rompaey et al., 2007). Approaches to code smell detection, whether based on metrics or heuristics, often depended on manually setting thresholds for various measurements. While these methods obtain promising outcomes, crafting optimal rules or heuristics manually proves to be highly challenging (Liu et al., 2015). Alternatively, relying solely on software metrics could result in a potential classification bias, as code snippets exhibiting distinct behaviors may share identical metrics value while exhibiting entirely different code smells.

In order to bypass the need for manually designed heuristics and rules based on thresholds, machine learning techniques have been adopted to establish a sophisticated relationship between code metrics and smell labels. A significant amount of research has focused on the application of traditional classification algorithms, such as SVM (Maiga et al., 2012a), Naive Bayes (Khomh et al., 2011), regression (Fontana and Zanoni, 2017), for the detection of code smells. Despite their promising potential, these approaches have often overlooked the examination of code representation features. Deep learning models have subsequently been adopted for the extraction of textual representation from source code. Numerous studies have been dedicated to the exploration of source code mining using deep neural networks (DNNs) (Hadj-Kacem and Bouassida, 2018; Liu et al., 2019a), convolutional neural networks (CNNs) (Liu et al., 2019a; Das et al., 2019), long short-term memory networks (LSTMs) (Ho et al., 2023), recurrent neural networks (RNNs) (Sharma et al., 2021), to name but few. Prior studies have primarily regarded source code as textual data and applied natural language processing techniques for text mining. However, it is evident that there exists syntactical distinctions between source code and natural language. Approaching source code as text for mining purposes and overlooking the semantic intricacies within its underlying structures (e.g., nesting control), poses the potential of failing to preserve the intended meaning. Moreover, code smells are frequently linked to distinct symptoms that manifest in control and data flows.

In this paper, we propose a holistic approach to code smell detection, focusing on assessing the effectiveness of incorporating both semantic structures and design-related features to capture the relationship between various types of smells and source code. In particular, we conceptualize EnseSmells for code smell detection on top of a two-tier deep learning architecture. The initial tier focuses on acquiring intricate semantic representations of code by integrating deep convolutional networks (CNNs) and long short-term memory networks (LSTMs). Utilizing a 1D-CNN model, we capture structured patterns from tokens in the source code. This information is then fed into a sequence of LSTMs to enhance context encoding and retain semantic details. Given that code smell symptoms are often related to specific design metrics, the second tier employs a deep neural network (DNN) with a single adaptive layer, emphasizing crucial metrics. This aims to learn from manually crafted features, maintaining the connection between code smell symptoms and these metrics.

This paper extends our previous work (Ho et al., 2023) with the following enhancements:

- We enhance the approach presented in the conference version by introducing a module dedicated to understanding the relationship between different code smells and software metrics.
- We extend the scope of experiments with a three-fold objective:
    1. Investigating the impact of adjusting various code embedding techniques on prediction performance, considering four types of code smells (i.e., Long Method, Feature Envy, Data Class, God Class). This paper broadens the scope of code representation by exploring multiple code embedding approaches such as code2vec, CodeBERT, and CuBERT.
    2. Examining whether the combination of code representation vectors and hand-crafted metrics can effectively contribute to the detection of code smells.
    3. Demonstrating the effectiveness of EnseSmells compared to state-of-the-art studies.
- We publish the packages developed alongside the metadata process in this paper to promote future research.[1]

[1] https://github.com/brojackvn/JSS-EnseSmells

Section 2 introduces the background related to the overview of code smells, the survey of software metrics for code smell detection, as well as pre-trained programming language models. It also provides an overview of related work in our research. Section 3 presents the proposed EnseSmells model for code smell detection. Empirical settings, including the benchmark datasets, the employed evaluation metrics, and our evaluation plan, are specified in Section 4. Section 5 discusses the results and address potential threats to the validity of our proposed approach in Section 6. Finally, Section 7 concludes the paper with insights into future directions.

2. Background and Related Work

This section offers a brief overview of the key elements in code smell detection. Initially, we explore different code smells that have garnered attention in recent times. We also provide an overview of software metrics, considering their relevance in the study of code smells. Following this, our focus shifts to deep learning architectures, emphasizing their role in code representation and their importance in the identification of code smells.

2.1. Categorization of Code Smells

The concept of *bad smells*, also known as *code smells*, was introduced by Fowler (Fowler, 1999) as a manner to describe possible issues in code might need refactoring. Software developers have identified a broad spectrum of code smells which can be classified into categories such as *implementation*, *design* and *architecture* depending on their level of granularity and their overall impact on the code (Fowler, 1999; Suryanarayana et al., 2014). Implementation smells emerge at a detailed level including *long method, long parameter list, complex conditional*, among others (Fowler, 1999). Design smells pertain to the principles of object-oriented design, like abstraction and coupling, and are exemplified by issues such as *god class, data class, multifaceted abstraction*, among others (Suryanarayana et al., 2014). Architectural smells, including instances like *god component and scattered functionality*, manifest at a broader granularity, spanning across multiple components and affecting the entire system (Garcia et al., 2009). In our research, we specifically examine four well-known types of code smells: *feature envy*, *long method*, *data class* and *god class*. These smells are well-documented and extensively researched in which the first two smells are associated with method-level issues whereas the remaining relates to class-level problems in the source code. A significant body of work has focused on the exploration and identification of these particular code smells (Sharma et al., 2021; Liu et al., 2019a; Das et al., 2019; Zhang et al., 2022).

Feature Envy. The concept *feature envy*, coined by Beck and Fowler (Fowler and Beck, 1997), describes a scenario where methods access from an object other than the one they belong to. This kind of code smell is indicative of a misplaced method, often marked by frequent interactions with attributes and methods of other classes. The identification of *feature envy* in source code can be performed by measuring the strength of coupling between classes or the lack of cohesion within a class (Barbez et al., 2020).

Long Method. The term *long method* is used to describe cases where a method accumulates too many responsibilities, effectively centralizing a classs functionality by blending various concerns. This smell is linked not only to methods that are excessively lengthy but also to those with intricate control structures, which complicates testing and debugging. Metrics that gauge the complexity of source code, such as the number of lines, the length of vocabulary used, and the McCabe Cyclomatic Complexity (McCabe, 1976), are useful in identifying issues related to *long method*.

God Class. A *god class*, also known as a Blob, is identified as a class that monopolizes the functions of a system by taking on too many responsibilities. This is evident from its extensive array of attributes, methods, and its interactions with data classes. Such centralization usually leads to difficulties in reusing and comprehending the source code. The complexity of source code, often reflected in a high number of lines and extensive vocabulary, is a clear indicator of this smell.

Data Class. The code smell *data class* describes a class in a software system that mainly functions as a repository for data, offering limited functionality. These classes act merely as holders of data, while the logic pertaining to the data tends to be dispersed throughout the code-base. This scattering issue can lead to maintenance challenges and error-proneness. Two primary indicators of this code smell are the absence of significant behavior in the class and the extensive accessibility of its data (Fowler and Beck, 1997).

2.2. Software Metrics for Code Smell Detection

In the realm of software engineering, software measurements serve as crucial quantitative indicators for assessing various aspects of software development and its outcome. These metrics are essential in gaining insights, conducting evaluations and enhancing both the process of software development and the final software product's quality (Vesra, 2015; Sharma and Singh, 2011). Fenton and Bieman have categorized software measurements into three distinct types: (i) *Processes* which include software-related activities like development and support; (ii) *Products* covering deliverables such as requirements, designs and code; (iii) *Resources* referring to people, equipment and tools (Fenton and Bieman, 2014). These measurements can be derived by assessing the attributes of various software artifacts within these specified categories. When it comes to boosting the quality of software products, metrics concerning size and design are particularly significant. These measurements are crucial for pinpointing the potential presence of faults or defects in the source code (Marinescu, 2004; Moha et al., 2009; Fontana and Zanoni, 2017; Ho et al., 2022).

A substantial amount of research has utilized software metrics for detecting code smells. These studies are commonly known as *metric-based approaches* and/or *rule/heuristic-based approaches* (Marinescu, 2004, 2005; Macia et al., 2010; Vidal et al., 2015; Lanza and Marinescu, 2006). Marinescu et al. introduced a detection strategy based on design metrics including *Weighted Method Count* (WMC), *Tight Class Cohesion* (TCC) and *Access To Foreign Data* (ATFD) to identify the *God Class* code smell (Marinescu, 2004). This approach was further expanded to incorporate various formulas based on source code complexity and design metrics, enabling the detection of ten different code smells (Marinescu, 2005). For instance, metrics like the number of public attributes (NOPA) and the number of accessor methods (NOAM) are employed together to identify the *Data Class* code smell. Lanza and Marinescu combined metric-based formulas with thresholds to detect 11 anti-patterns (Lanza and Marinescu, 2007). They proposed heuristics by logically combined metric-threshold pairs using logical operators to define detection rules. For example, they detected *Long Methods* by introducing an AND-expression of four metric-threshold pairs including *Lines of Code* (LoC), *McCabe's Cyclomatic Complexity*, *Maximal Nesting Level* and *Number of Accessed Variables*. These heuristics have been adopted inside the *InCode* (Marinescu et al., 2010) package as an Eclipse-plugin. Sales et al. have proposed a dependency based approach, called *JMove* to identify *Feature Envy* (Sales et al., 2013). This approach involves defining metrics to quantify the similarity between the dependencies created by the considering method and those of all methods in dependent classes. In a similar vein, numerous tools and methods have been proposed to detect different kinds of code smells including CCFinder (Kamiya et al., 2002), SpIRIT (Vidal et al., 2015), DECOR (Moha et al., 2009), among others. While these studies have yielded encouraging outcomes in detecting common code smells in source code, further extensive research is required to reach a level of maturity. A significant challenge in rule/heuristic-based approaches lies in establishing metric thresholds. Indeed, for software engineers, manually setting these thresholds in smell detection algorithms presents a considerable challenge (Liu et al., 2015). Additionally, the ambiguity in detecting code smells from a given code snippet arises because various code smells may be associated with the same metrics. This necessitates distinct characteristics to effectively differentiate one type of code smell from others.

2.3. Pre-trained Programming Language Models for Code Smell Detection

Considering the significant potential of text representation models in forming the cornerstone for linking human communication with machine comprehension, applying Natural Language Processing (NLP) across various Software Engineering (SE) tasks promises substantial accomplishments. Early foundational word embedding models like *word2vec* (Mikolov et al., 2013) and *GloVe* (Pennington et al., 2014) have been explored in various Software Engineering (SE) tasks, such as code search (Sachdev et al., 2018), bug localization (Xiao et al., 2019), and code summarization (Zhang et al., 2020), among others. These studies typically utilized pre-trained models to generate embedding for specific purposes. However, the use of these models in SE is limited by their treatment of technical terms as if they were ordinary text, overlooking their specific meanings in the software engineering context. For instance, while bug typically means an insect in general usage, in the SE field, it denotes a software defect. This limitation has steered research towards exploring code embedding models that are more attuned to the nuances of software engineering terminology.

Various methods for creating embeddings from code have emerged, marking significant advancements in the use of NLP for Software Engineering (SE) (Von der Mosel et al., 2022). These techniques are mainly categorized into two groups based on the training context for code embeddings (Ding et al., 2022). The first category approaches source code as simple text, using standard word embedding models on the tokenized sequence of tokens from the source code. The second category involves transforming source code into structural representations using Abstract Syntax Trees (AST), thereby capturing both lexical and syntactic elements of the code. Recently, there has been a shift towards Transformer-based models (e.g., BERT, RoBERTa, Code2Vec, CodeBERT) (Devlin et al., 2018; Liu et al., 2019b; Alon et al., 2019; Feng et al., 2020), focusing on contextual embeddings that comprehend the contextual meanings of ambiguous terms. Here, each word occurrence is assigned a dense vector, with an emphasis on its specific context. The subsequent part of this section provides a comprehensive overview of the latest advancements in code embedding methods.

Code2Vec. The *code2vec* model (Alon et al., 2019) accounts for the structural context present in the source code. It converts the Abstract Syntax Tree (AST) of a code fragment, typically a single method or function, into a set of paths, known as path-contexts. Utilizing a neural attention-based feed-forward network, the model determines the appropriate level of attention to allocate to each path-context. This process yields an embedding vector for each individual method, derived from its path-contexts.

CodeBERT. The *CodeBERT* model (Feng et al., 2020), based on a multi-layer bidirectional transformer structure similar to BERT (Devlin et al., 2018) and its variant RoBERTa (Liu et al., 2019b), is designed for bi-modal processing of both programming (PL) and natural languages (NL). It effectively manages source code as well as human language. This model is pre-trained using the CodeSearchNet corpus (Husain et al., 2019), which includes over 2 million bi-modal code-documentation pairs and 6.4 million uni-modal code fragments in diverse programming languages like Python, Java, JavaScript, PHP, Ruby, and Go. *CodeBERT* has shown remarkable capabilities in essential tasks such as code search and generating code documentation (Feng et al., 2020).

CuBERT. The CuBERT model (Kanade et al., 2020) is a specialized adaptation of the BERT model for deriving contextual embeddings from source code. While it shares the architectural design and pre-training approach of BERT, CuBERT differs in its pre-training dataset and tokenization process. It uses a vast collection of 7.4 million GitHub files for pre-training. Unlike traditional BERT, which is tailored for natural languages and struggles to capture the intricate syntax of programming languages, CuBERT addresses this gap. The standard BERT tokenization overlooks key programming elements like indentations, parentheses, semicolons, and arithmetic operators CuBERT, however, overcomes this by ensuring these critical structural components are maintained during the training process.

With the significant progress in code representation models, a diverse range of research has been conducted, utilizing these models for efficient code smell detection. These studies can be generally categorized into two streams: (i) *Machine learning-based approaches* for code smell detection, and (ii) *Deep learning-based methods* for detecting code smells.

2.3.1. Machine learning-based approaches

Machine learning-based methods for detecting code smells have attracted the interest of software engineering researchers. These techniques involve training models through data-driven inference, enabling them to learn from examples and apply this knowledge to new cases. This method boosts the effectiveness and efficiency of identifying code smells, thereby enhancing the quality and maintainability of software. Maiga et al. proposed SVMDetect based on the Support Vector Machine for code smell detection (Maiga et al., 2012b). They employed an input vector encompassing 60 structural metrics for each class to detect four well known code smells including God Class, Functional Decomposition, Spagheti Code and Swiss Army Knife. In an extensive study by Fontana et al. (Arcelli Fontana et al., 2016), 16 different machine learning algorithms, including their boosting variants, were used to detect four specific code smells: data class, large class, feature envy, and long method. The experiments were carried out using a range of independent metrics at various levels of granularity, including class, method, package, and project, showcasing the potential of the machine learning-based approach comparing to the rule-based ones. Along the same line of research direction, Azadi et al. developed WekaNose, a semi-automated tool aimed at detecting code smells (Azadi et al., 2018). This approach involves learning the relationship between code smell instances and pertinent metrics, leading to the automatic generation of rules that improve the effectiveness of code smell detection. Generally, initial research in machine learning algorithms for smell detection predominantly depends on software metrics. Recent research by Aleksandar et al. (Kovačević et al., 2022) demonstrated enhanced performance with a machine learning model that combines CuBERT embeddings and manually crafted code metrics, outperforming traditional heuristic metric-based methods. Such progress is aiding the continuous enhancement of using code representation models for effective code smell detection.

2.3.2. Deep learning-based approaches

Deep learning-based methods for detecting code smells focus on uncovering complex features in the ever-growing data volumes. Das et al. utilized a CNN to identify patterns in software metrics extracted using IPlasma, achieving notable accuracy in pinpointing Brain Class and Brain Method code smells (Das et al., 2019). Liu et al. adopted word2vec for textual representation, combining class names, enclosing package names, and target package names into a word sequence, which, along with additional features, was fed into a deep learning model for classification (Liu et al., 2019a). Sharma et al. evaluated various deep learning techniques, including CNN, RNN, and autoencoder models, for the detection of different code smells, and also investigated the transferability of models between programming languages, showing promising results (Sharma et al., 2021). Ho et al. introduced DeepSmells as an effective tool for code smell detection, employing diverse deep learning methods to identify patterns and semantic features in code snippets, establishing its prowess as a state-of-the-art technique in the field (Ho et al., 2023).

3. The Proposed Approach

Building upon our prior research, namely DeepSmells (Ho et al., 2023), where we concentrated on extracting unique features and patterns from source code, our current focus is on seamlessly integrating the strengths of metrics-based and deep learning approaches. EnseSmells combines the automated generation of both *semantic* and *structural* features from source code. The ultimate aim is to improve the accuracy of code smell detection and to address the specific challenges encountered in previous methods.

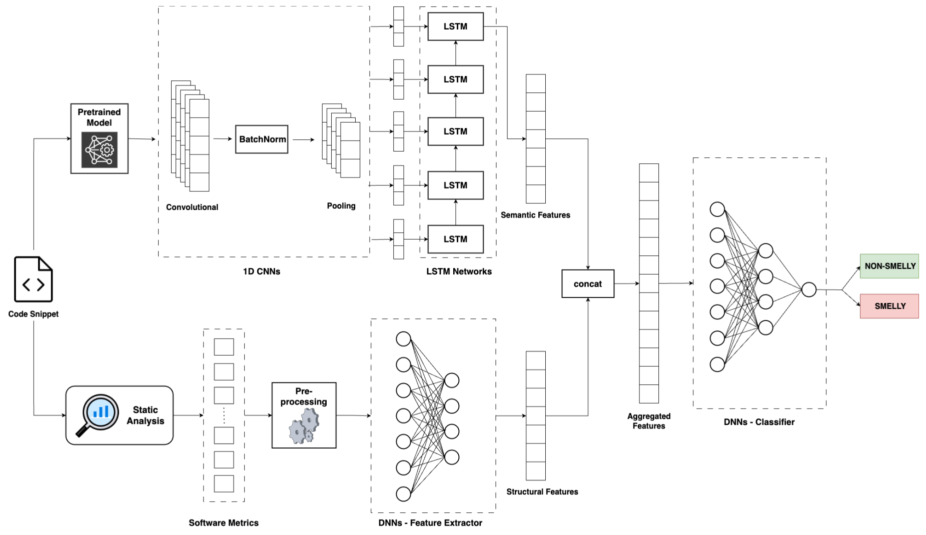


Figure 1: The overall architecture of EnseSmells.

As illustrated in Figure 1, our approach comprises two distinct phases. In the initial phase, our model focuses on maximizing the extraction of valuable features from code snippets. We emphasize two types of features: *(i)* Extracting pattern and semantic features from code snippets, building on our previous work (Ho et al., 2023); *(ii)* Extracting structural features from code snippets through automatic learning, a crucial component in code smell detection. Finally, these features are then integrated and input into a deep neural network classifier. Once the classifier model has been constructed, involving the determination of weights and biases, it can provide a probability for each code snippets, indicating whether it exhibits code smell or is clean.

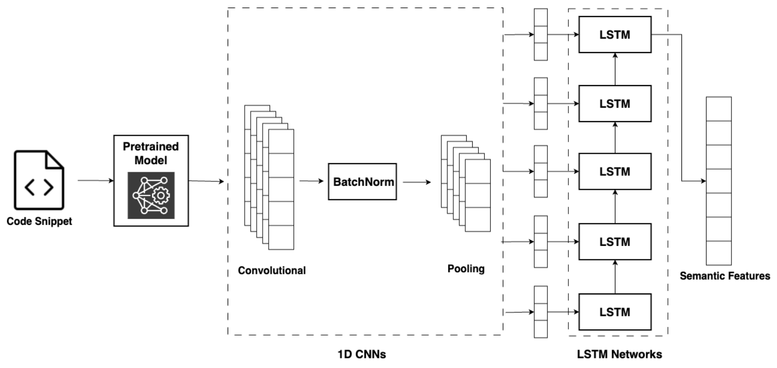


Figure 2: The module for extracting *semantic* features.

3.1. Extracting semantic features for code smell detection

This module comprises four distinct phases, illustrated in Figure 2. During the pre-processing phase, we perform several critical steps on the dataset, including: *(i)* embedding source code by indexing code tokens using Tokenizer;[2] *(ii)* computing statistical information about the samples' length and excluding samples that deviate more than one standard deviation from the mean; *(iii)* padding samples with zero values to match the length of the longest array input. This phase is referred to as the *token-indexing* method.

Subsequently, in the model-building phase, the pre-processed data is passed through convolution blocks to extract unique features from the source code automatically. Typically, many convolution blocks are employed to progressively learn more complex and abstract features from the input data, including high-level characteristics such as component relationships within the source code. However, when deeper networks can start converging, a degradation problem has been exposed: with the network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly (He et al., 2016). In this case, two convolution blocks are utilized as follows.

- The first convolution block consists of a 1D-CNN (`torch.nn.Conv1d`) with 16 filters with a kernel size k which specifies the length of 1D convolution window. We experiment different values of k (i.e., $k = 3, 4, 5, 6, 7$) to evaluate the effectiveness of the convolution layer. This layer employs a `ReLU` activation function, and the remaining parameters are kept as defaults. Following by a 1D-Batch Normalization (`torch.nn.BatchNorm1d`) to accelerate the network

[2] Available at: https://github.com/dspinellis/tokenizer

training and to reduce internal covariate shift (Ioffe and Szegedy, 2015). The obtained feature map is then passed to a MaxPooling layer (`torch.nn.MaxPool1d`) by a factor of size 3 to reduce the spatial dimension.

- The second convolution block is similar to the first one, excepting that we use 32 filters.

Once the output from the convolutional neural networks has been obtained, it is fed into the LSTM network to preserve the meaning and context of the data. The `torch.nn.LSTM` function is used to configure the LSTM network, with the input size of each LSTM unit being based on the initial size of the initial embedding source code vector. The number of LSTM units in the network is also set to 32 to match the 32 filters in the second convolution block. In addition, we implement BiLSTM to capture long-term dependencies in sequential data that span both forward and backward directions. BiLSTM consists of two separate layers of LSTM units, one for processing the input sequence in the forward direction and the other layer for the backward direction. The hidden state at a time step of the network is the concatenation of both the forward and backward hidden states, allowing the hidden states to capture future information. We use the same configuration as before but set the hyperparameter `bidirectional` to `True`. The final hidden state from this network serves as the *semantic features* of this module.

In the encoding phase of our previous research, we employed the *token-indexing* method, involving a mapping between integers and tokens to convert token vectors into integer vectors. It is important to note that a simple index does not convey rich contextual information about code snippets. On the other hand, large language models specifically designed for programming languages, such as CodeBERT, CuBERT, and code2vec (see Section 2), are trained on extensive and intricate datasets. These models have the capacity to learn complex semantic features from code snippets due to their sophisticated architecture and specialized training, providing a more comprehensive understanding of the underlying code semantics. In this study, we conduct experiments involving various techniques in the encoding phase while retaining the rest of the module.

- *CodeBERT*. As done in the original study (Feng et al., 2020), during the pre-training phase, we format the input as the concatenation of two segments separated by a special token: $[CLS], w1, w2, ..wn, [SEP], c1, c2, ..., cm, [EOS]$. One segment represents the natural language text, typically a comment for a function,and the other segment contains code from a specific programming language. The $[CLS]$ token at the beginning is special and provides the aggregated sequence representation for classification (Feng et al., 2020). CodeBERT treats a piece of code as a sequence of tokens. The output of CodeBERT includes: *(i)* Contextual vector representations for each token in both natural language and code, and *(ii)* The representation of $[CLS]$, which serves as the aggregated sequence representation.

- *CuBERT*. To feed Java code snippets into the BERT Large model, we utilized a custom Java tokenizer developed by Kanade *et al.* (Kanade et al., 2020). This preprocessing step enabled us to convert each Java code sequence, or line of code, into a format suitable for the CuBERT model. Subsequently, for each line of code, we extracted a 1024-dimensional embedding from a special token referred to as the start-of-example $[CLS]$ (Devlin et al., 2018).
- *Code2Vec* is specialized in code embeddings, and input comprises vector representations of code snippets, and the output corresponds to specific tags. Alon et al. (2019) exemplified this by using method body as input and method names as output tags. For code smell detection, we employed *code2vec* to represent each method with fixed-length vectors, following these steps: *(i)* Constructing methods' AST paths using a Java path extractor; *(ii)* Feeding the extracted AST paths into a pretrained model, excluding the last softmax layer, and utilizing the resulting 384-dimensional code vector as our chosen representation for each method.

For class embedding, we addressed the challenge that pretrained *code2vec* models cannot directly handle. Following an existing study (Compton et al., 2020), we treated classes as collections of methods. An evaluation of class embedding revealed that the mean aggregation method consistently provided the best results. Thus, to embed a class using the *code2vec* model, we performed the embedding process for all its constituent methods and calculated the mean of the resulting code embedding vectors.

3.2. Extracting structural features for code smell detection

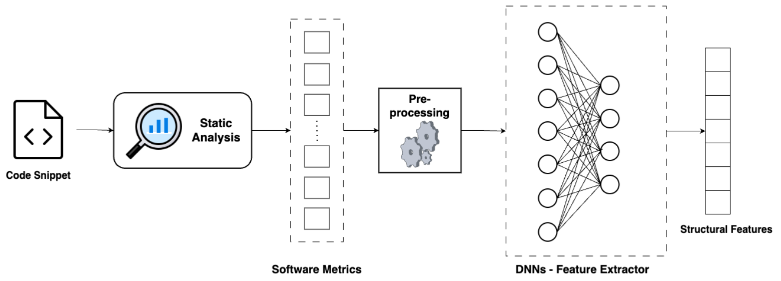


Figure 3: The module for extracting *structural* features.

To detect code smells, numerous efforts have relied on metric-based and rule/heuristic-based methodologies, which employ predefined sets of metrics, rules, and associated thresholds to achieve acceptable results. However, these established approaches are not without their challenges, notably in *(i)* defining appropriate metrics and thresholds, and *(ii)* addressing insufficient feature extraction. In response to these challenges, we conceptualized a novel module by harnessing a comprehensive set of object-oriented metrics as independent variables, enabling deep learning models to autonomously capture essential structural features for code smell detection as shown in Figure 3.

In the analysis phase, we selected object-oriented metrics at both the method and class levels, utilizing the static analysis tool known as CK.[3] These metrics, as categorized by the original study (Arcelli Fontana et al., 2016), span various aspects of the code, including size, complexity, cohesion, coupling, encapsulation, and inheritance. The complete list of these metrics is provided in Table 1, with detailed descriptions available in Table 10 in the Appendix.

Table 1: The selected object-oriented metrics (Arcelli Fontana et al., 2016).

Size	**Complexity**	**Cohesion**	**Coupling**	**Encapsulation**	**Inheritance**
LOC	CYCLO	LCOM5	FANIN	LAA	DIT
NOM	WMC	TCC	FANOUT	NOAM	NOI
NOM	WMCNAMM		ATFD	NOPA	NOC
NOPK	AMWNAMM		FDP		NMO
NOCS	MAXNESTING		RFC		NIM
NOA	WOC		CFNAMM		NOII
	CLNAMM		CINT		
	NOP		CDISP		
	NOAV		MaMCL		
	ATLD		MeMCL		
	NOLV		NMCS		
			CC		
			CM		

Our metric selection process focused on well-established metrics widely acknowledged in the literature (see Section 2.2). Additionally, we defined custom metrics in Tables 2 and 3, including 44 class level metrics and 28 method level metrics for each type of dataset, primarily based on combinations of modifiers (e.g., public, private, protected, static, final, and abstract) on attributes and methods. The comprehensive names of these metrics are in Tables 11 and 12 in the Appendix.

[3] Available at: `https://github.com/mauricioaniche/ck`

Table 2: The custom metrics selected for class level.

anonymousClassesQty	assignmentsQty	totalFieldsQty	logStatementsQty
innerClassesQty	returnQty	visibleFieldsQty	tcc
abstractMethodsQty	loopQty	variablesQty	lcc
privateMethodsQty	uniqueWordsQty	tryCatchQty	wmc
protectedMethodsQty	privateFieldsQty	lambdasQty	loc
publicMethodsQty	protectedFieldsQty	parenthesizedExpsQty	lcom
defaultMethodsQty	publicFieldsQty	numbersQty	rfc
staticMethodsQty	defaultFieldsQty	comparisonsQty	cbo
finalMethodsQty	staticFieldsQty	maxNestedBlocksQty	dit
synchronizedMethodsQty	finalFieldsQty	mathOperationsQty	modifiers
totalMethodsQty	synchronizedFieldsQty	stringLiteralsQty	nosi

Several preprocessing techniques have been applied to prepare the data for analysis as follows: *(i)* We encoded categorical variables using label encoding; *(ii)* We removed metrics with constant values; *(iii)* For cases where a metric value couldn't be calculated for a particular code sample, we used k-Nearest Neighbors from the scikit-learn library[4] to impute the missing value. However, if the amount of missing data for metrics exceeded 5%, those metrics were removed; *(iv)* We normalized the metric values using StandardScaler from the scikit-learn library. StandardScaler is employed when the input dataset characteristics differ significantly in their ranges or units of measure. It removes the mean and scales the data to have unit variance. These preprocessing steps ensure that the data is in a suitable format for the final step, where we utilize a deep neural network with a single adaptive layer. This layer is efficient and convenient as it adaptively sets the number of hidden nodes, unlike traditional neural networks where parameters are tuned through iterations. This adaptive layer assigns higher weights to the importance of each metric and transforms them into a lower-dimensional latent space. This approach effectively captures the essential *structural features* for code smell detection.

3.3. Unifying structural and semantic features in source code through ensemble for enhancing code smell detection

In some classification tasks, multiple output embeddings may be available, with each capturing a different aspect of the input space with varying strengths. These embeddings offer non-redundant information about the output space, and by ensembling them, we can create a more robust joint

[4]Available at: https://scikit-learn.org/

Table 3: The custom metrics selected for method level.

constructor	variablesQty	tryCatchQty	anonymousClassesQty
line	parametersQty	parenthesizedExpsQty	innerClassesQty
cbo	methodsInvokedQty	stringLiteralsQty	lambdasQty
wmc	methodsInvokedLocalQty	numbersQty	uniqueWordsQty
rfc	methodsInvokedIndirectLocalQty	assignmentsQty	modifiers
loc	loopQty	mathOperationsQty	logStatementsQty
returnsQty	comparisonsQty	maxNestedBlocksQty	asJavaDoc

embedding. In our prior work (Ho et al., 2023), we solely considered the semantic features learned from source code using a relatively simple deep learning model. However, the architecture of the model was not sufficiently complex, and the data volume was insufficient to comprehensively represent the characteristics of each code smell. In this study, we aim to address these limitations by providing additional information to enhance the model's capabilities. To incorporate this information, we merge semantic features with structural vectors using the `torch.cat` operator in PyTorch. This ensemble of feature vectors is then fed as input to the Deep Imbalanced Neural Network to classify code snippets as either containing code smells or not. This ensemble approach forms the core of EnseSmells, accentuating the significance of ensemble techniques in our methodology.

The network comprises two hidden layers with ReLU as the activation function. The output layer, crucial for binary classification, features a single node with a Sigmoid activation function. Notably, the optimal number of hidden nodes is determined empirically, ensuring adaptability to different datasets. However, in scenarios where data exhibits a significant class imbalance, a challenge arises. The model's classification can exhibit a bias toward the majority class, even though the minority class may hold greater importance. To address this, we introduce a sensitivity weight into the binary cross-entropy loss function. This weight adjustment allows us to fine-tune the significance of the minority class in the classification process. To discover the optimal weight value, we undertake a series of experiments, comparing the performance of binary cross-entropy with a weight value of 1 to weighted binary cross-entropy with varying weight values, such as 2, 4, 8, 12, 32, and 84. The comparison results guide us in selecting the best hyperparameter for our proposed method. In the training and testing phase, we trained our model using a mini-batch size of 128. The learning rate is set to 0.025. The training is done within 85 epochs using the SGD to optimize the weighted loss function.

4. Empirical Settings

We ran the evaluation with Google Colab[5] using a virtual machine equipped with an Intel Xeon CPU with 2 vCPUs (virtual CPUs) and 13GB of RAM. The virtual machine was equipped with a NVIDIA Tesla T4 GPU with 16GB of VRAM to accelerate the deep learning computations. To evaluate EnseSmells, we re-ran all the baselines using the same benchmark datasets introduced in Section 4.1. To ensure a robust evaluation, we initially split the dataset into 80%:20% for training and testing, respectively. Moreover, to mitigate any potential bias and variance related to the test set resulting from the dataset split, stratified 5-fold cross-validation was applied, resembling traditional k-fold cross-validation but preserving the class distribution within each split.

4.1. Benchmark dataset

The MLCQ dataset–originally made available by an existing study (Madeyski and Lewowski, 2020)–was used in our experiments. The dataset comprises 14,739 reviews related to approximately 5,000 Java code snippets gathered from 26 software developers. The reviews are categorized into four types of code smells, i.e., God Class and Data Class at the class level, and Feature Envy and Long Method at the method level.

Table 4: Statistics of the datasets.

Smell	Smell Alias	# Positive	# Negative
Long Method	LM	1,993	243
Feature Envy	FE	2,126	64
Data Class	DC	1,838	282
God Class	GC	1,857	228

Leveraging this information, we retrieved the code snippets and assigned labels to each of them. In the original dataset, each review categorizes a code snippet into one of four severity levels: *none, minor, major*, and *critical*. This problem is formalized as binary classification, where code samples are labeled as *smelly* if they have more smelly reviews (i.e., *minor, major, critical* severity) than non-smelly reviews (i.e., *none* severity). Conversely, we classify them as *non-smelly* if they have a large number of non-smelly reviews. Code snippets with an equal count of smelly and non-smelly reviews are subsequently removed. Furthermore, this classification problem exhibits a significant

[5]Available at: `https://colab.research.google.com/`

class imbalance, with the positive class (smelly instances) representing the minority class. The average imbalance rate across all datasets is 10.46%, as illustrated in Table 4.

4.2. Evaluation metrics

The evaluation adopts widely accepted evaluation metrics, including Precision (P), Recall (R), F1-Score (F1), and the Matthews Correlation Coefficient (MCC). These metrics hold significance in assessing our approach's effectiveness, as they are commonly utilized not only in the context of code smells detection (Sharma et al., 2021; Kovačević et al., 2022) but also in addressing imbalanced classification challenges (Hossin and Sulaiman, 2015).

Precision, Recall, and F1-Score. Confusion matrix is an effective means to evaluate any classifier with four possible outcomes including *true positive* (TP), *true negative* (TN), *false positive* (FP) and *false negative* (FN). *Precision* measures how many of the positive predictions made by the model are actually correct: $P = \frac{TP}{TP+FP}$; *Recall* counts the number of positive cases in the dataset that model can identify: $R = \frac{TP}{TP+FN}$; *F1-Score* represents the harmonic mean of *precision* and *recall* of the prediction model: $F1 = \frac{2*P*R}{P+R}$.

MCC. The metric is used with imbalanced classification, measuring the correlation between the predicted and actual class, which lies in the range $[-1, 1]$, where 1 represents a perfect prediction, and -1 shows a perfect negative correlation; An MCC equal to 0 corresponds to a random prediction.

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \tag{1}$$

4.3. Research questions

RQ$_1$: *How does EnseSmells compare with DeepSmells?* An extensive experiment was conducted with EnseSmells and DeepSmells, utilizing various embedding techniques within the semantic module. The aim is to identify the optimal configuration under each embedding technique, maximizing prediction performance for code smells detection.

RQ$_2$: *How do software metrics impact the performance enhancement of EnseSmells?* Both EnseSmells and DeepSmells experiments adopt identical structures and embedding techniques for the semantic module. However, EnseSmells utilizes both *semantic* and *structural* modules, while DeepSmells solely relies on the *semantic* module. This comparison explores the impact of software metrics on enhancing the overall performance of the model.

RQ$_3$: *How do various embedding techniques impact each category of code smell?* Our study investigates a variety of embedding techniques, each characterized by unique architectures as detailed in Section 2.3, with the objective of capturing distinctive features associated with individual smells. We compare the performance of various embedding techniques with both EnseSmells and DeepSmells to assess the effectiveness of pre-trained programming language models in addressing different types of code smells.

RQ$_4$: *How eective is EnseSmells compared with classical ML classifiers when utilizing only structural features?* Given that our EnseSmells method combines various features within a complex architecture, this question aims to evaluate its effectiveness in comparison to traditional machine learning classifiers that solely rely on software metrics. We utilize classic classifiers with configurations as outlined in Table 5 to determine the most optimal choice.

Table 5: The parameter settings of the used machine learning classifiers.

Classifier	**Hyperparameter settings** (*scikit-learn* library)
Naive Bayes (NB)	`sklearn.naive_bayes.GaussianNB`:
	$var\_smoothing = [1e^{-9}, 1e^{-8}, 1e^{-7}, 1e^{-6}, 1e^{-5}]$
Nearest Neighbor (NN)	`sklearn.neighbors.KNeighborsClassifier`:
	$n\_neighbors = [1, 3, 5, 7, 9]$
Random Forest (RF)	`sklearn.ensemble.RandomForestClassifier`:
	$n\_estimators = [10, 50, 100, 200]$
	$max\_depth = [None, 10, 20, 30]$
Logistic Regression (LR)	`sklearn.linear_model.LogisticRegression`:
	$C\_values = [0.001, 0.01, 0.1, 1, 10, 100]$
Classification and	`sklearn.tree.DecisionTreeClassifier`:
Regression Tree (CART)	$max\_depth\_values = [None, 10, 20, 30]$
	$min\_samples\_split\_values = [2, 5, 10]$
	$min\_samples\_leaf\_values = [1, 2, 4]$
	$max\_features\_values = [auto, sqrt, log2]$
Support Vector	`sklearn.svm.SVC`:
Machine (SVM)	$kernel\_function = GaussianRBF$
	$gamma\_values = [0.001, 0.01, 0.1, 1, 10, 100]$
	$C\_values = [0.001, 0.01, 0.1, 1, 10, 100]$
Back Propagation	`sklearn.neural_network.MLPClassifier`:
neural networks (BP)	$hidden\_layer\_size = [16, 32, 64, (32, 16)]$
	$learning\_rate\_init = [0.001, 0.01, 0.1]$
	$max\_iter = [100, 200, 300]$
	$tol\_err = [1e^{-4}, 1e^{-3}, 1e^{-2}]$

RQ$_5$: *How does EnseSmells perform compared to the baselines?* This research questions compares EnseSmells with aforementioned baselines. This includes a comparison with the state-of-the-art ML_CuBERT model (Kovačević et al., 2022) and benchmarking against three baseline models (Sharma et al., 2021). This comprehensive evaluation provides an in-depth understanding of EnseSmells performance.

As for the baseline in our previous work, conducted by Sharma et al. (2021), they have investigated the efficiency of different variants of an auto-encoder model to detect code smells. The objective of an auto-encoder is to compress the source code and to learn salient information which is reflected in the reconstructed output (Sharma et al., 2021). Three architectures of auto-encoder have been studied and showed promising results, as follows: *(i) AE-Dense:* The auto-encoder model employed denses for encoder and decoder layers; *(ii) AE-CNN:* The auto-encoder model employed two CNN networks, the first one for encoder and the other for decoder; and *(iii) AE-LSTM:* Similar to AE-CNN but the CNNs are replaced by LSTM networks. The encoder and decoder layers are followed by a fully connected dense layer for classification.

5. Results and Discussion

5.1. ***RQ$_1$****:* How does EnseSmells compare with DeepSmells?

Table 6 presents the optimal performance of the EnseSmells model with different embedding techniques, including *token-indexing, CuBERT, CodeBERT*, and *code2vec*, each under their respective optimal network configurations. To facilitate a comprehensive comparison with DeepSmells, we select the optimal configurations, as shown in Table 7. This table provides a performance evaluation that compares EnseSmells with both DeepSmells (Ho et al., 2023) and its extended variant. The comparison involves the application of various embedding techniques, specifically *CuBERT, CodeBERT*, and *code2vec*, with DeepSmells employing these techniques in their respective optimal network configurations. Table 7 shows that EnseSmells achieves the best performance when compared with DeepSmells and its variants.

In particular, with respect to the LM and FE code smells, the best-performing variant for both types of code smells is DeepSmells$_{CuBERT}$, and the worst-performing variant is DeepSmells$_{code2vec}$ within the variants of DeepSmells. For the LM code smell, EnseSmells demonstrates improvements of 1.28% and 1.07% in F1-Score and MCC, respectively, compared to DeepSmells$_{CuBERT}$.

Table 6: Performance of different embedding techniques in EnseSmells architecture under optimal configurations.

Smell	**Model**	**Evaluation metric**			
		P	**R**	**F1**	**MCC**
LM	**EnseSmells**$_{token-indexing}$	0.8215	0.7873	**0.8016**	**0.7780**
	EnseSmells$_{CuBERT}$	**0.8482**	0.7391	0.7877	0.7616
	EnseSmells$_{CodeBERT}$	0.7938	0.7824	0.7857	0.7611
	EnseSmells$_{code2vec}$	0.7526	**0.8242**	0.7841	0.7618
FE	**EnseSmells**$_{token-indexing}$	0.4821	0.3554	0.3982	0.3849
	EnseSmells$_{CuBERT}$	**0.5215**	**0.3393**	**0.4025**	**0.3868**
	EnseSmells$_{CodeBERT}$	0.5128	0.2718	0.3527	0.3449
	EnseSmells$_{code2vec}$	0.4423	0.3727	0.3538	0.3568
GC	**EnseSmells**$_{token-indexing}$	**0.6962**	0.5077	**0.5801**	**0.5318**
	EnseSmells$_{CuBERT}$	0.5565	0.5105	0.5153	0.4675
	EnseSmells$_{CodeBERT}$	0.5993	0.5607	0.5764	0.5241
	EnseSmells$_{code2vec}$	0.5947	**0.5724**	0.5729	0.5288
DC	**EnseSmells**$_{token-indexing}$	0.7340	**0.5427**	0.6203	0.5638
	EnseSmells$_{CuBERT}$	0.8027	0.4567	0.5772	0.5238
	EnseSmells$_{CodeBERT}$	0.7447	0.5390	0.6199	0.5648
	EnseSmells$_{code2vec}$	**0.8123**	0.5314	**0.6393**	**0.5907**

Although DeepSmells$_{CuBERT}$ boasts a higher precision by 3.45%, this gain is offset by a notable reduction in recall by 3.83%. Moreover, the metrics of EnseSmells significantly surpass those of DeepSmells$_{code2vec}$, with differences of 2.41% (Precision), 58.91% (Recall), 48.64% (F1-Score), and 52.71% (MCC). For the FE code smell, EnseSmells showcases a substantial increase compared to DeepSmells$_{CuBERT}$, with improvements of 8.22% and 6.67% in F1 and MCC, respectively. Notably, EnseSmells achieves a higher recall value compared to DeepSmells$_{CuBERT}$ by 10.71%, while the precision value is lower by only 7.21%. Additionally, when compared to DeepSmells$_{code2vec}$, EnseSmells exhibits a significant increase of 22.1% and 21.85% in F1 and MCC, respectively.

By the remaining code smells, we observe that DeepSmells$_{token-indexing}$ is the best variant of DeepSmells for GC code smell, yet it still falls short of EnseSmells by 3.09% and 3.80% in F1-Score and MCC, respectively. Notably, DeepSmells$_{token-indexing}$ boasts a higher precision value by 9.73%, but its recall value is lower by 13.09% compared to EnseSmells. For DC code smell, EnseSmells outperforms DeepSmells$_{code2vec}$, the best variant of DeepSmells in this type of smell, in all four evaluation metrics by 12.40%, 1.82%, 5.54%, and 7.22%. It's worth highlighting that while DeepSmells$_{token-indexing}$ has the highest precision at 95.79%, its recall value is only

Table 7: Comparing the performance of DeepSmells model employing various embedding techniques under optimal configurations with EnseSmells model.

Smell	Model	Metric			
		P	**R**	**F1**	**MCC**
LM	**DeepSmells**$_{token-indexing}$ (Ho et al., 2023)	0.8327	0.7470	0.7865	0.7607
	DeepSmells$_{CuBERT}$	**0.8560**	0.7490	0.7888	0.7673
	DeepSmells$_{CodeBERT}$	0.7903	0.7448	0.7661	0.7373
	DeepSmells$_{code2vec}$	0.7974	0.1982	0.3152	0.2509
	EnseSmells	0.8215	**0.7873**	**0.8016**	**0.7780**
FE	**DeepSmells**$_{token-indexing}$ (Ho et al., 2023)	**0.5936**	0.2322	0.3198	0.3318
	DeepSmells$_{CuBERT}$	0.5143	0.2439	0.3203	0.3201
	DeepSmells$_{CodeBERT}$	0.5590	0.2048	0.2833	0.2947
	DeepSmells$_{code2vec}$	0.3603	0.1422	0.1815	0.1683
	EnseSmells	0.5215	**0.3393**	**0.4025**	**0.3868**
GC	**DeepSmells**$_{token-indexing}$ (Ho et al., 2023)	**0.7193**	0.4559	0.5542	0.5049
	DeepSmells$_{CuBERT}$	0.6191	0.4809	0.5277	0.4747
	DeepSmells$_{CodeBERT}$	0.7007	0.2798	0.3845	0.3270
	DeepSmells$_{code2vec}$	0.4647	0.5293	0.4780	0.4272
	EnseSmells	0.6220	**0.5867**	**0.5851**	**0.5429**
DC	**DeepSmells**$_{token-indexing}$ (Ho et al., 2023)	**0.9579**	0.1476	0.2531	0.0439
	DeepSmells$_{CuBERT}$	0.9158	0.1389	0.2382	0.0281
	DeepSmells$_{CodeBERT}$	0.6916	0.2868	0.3802	0.2573
	DeepSmells$_{code2vec}$	0.6883	0.5132	0.5839	0.5185
	EnseSmells	0.8123	**0.5314**	**0.6393**	**0.5907**

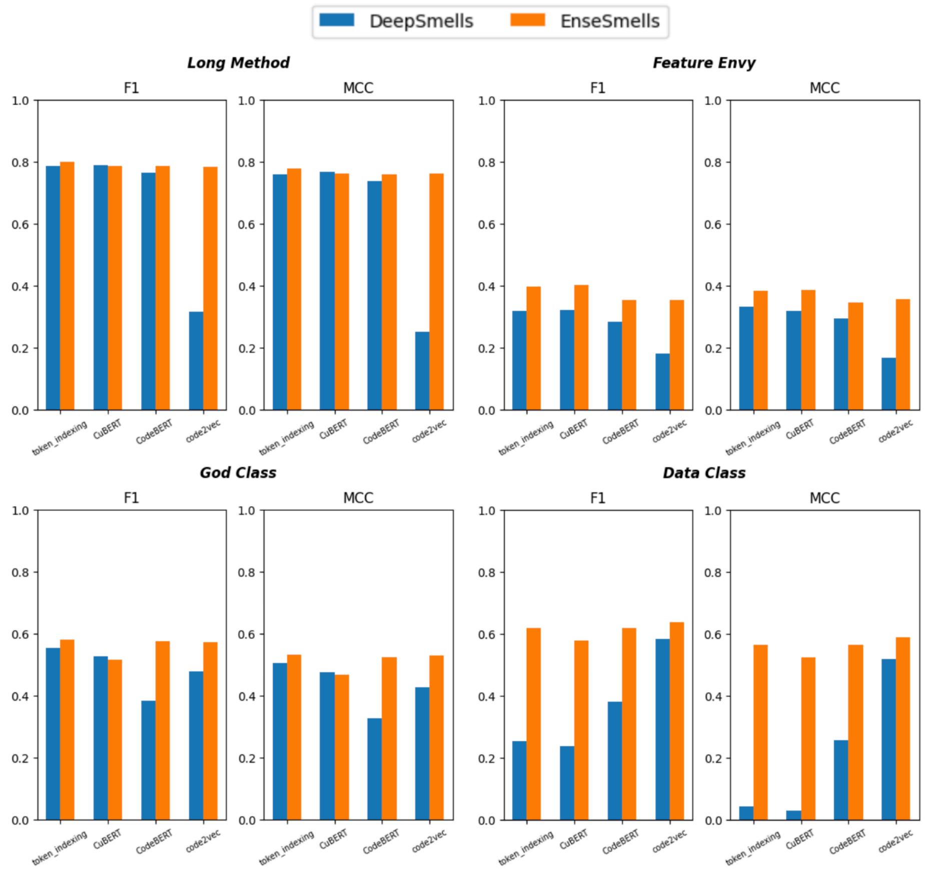
495 14.75% due to the high imbalanced dataset, resulting in lower F1-Score and MCC compared to EnseSmells, by 38.62% and 54.68%, respectively.

Answer to RQ$_1$. The method refinement in EnseSmells, achieved through the integration of both *structural* and *semantic* features, alongside the adoption of suitable embedding techniques, results in a superior performance compared to the original architecture, DeepSmells.

5.2. ***RQ$_2$****:* How do software metrics impact the performance enhancement of EnseSmells?

Figure 4 compares the performance of EnseSmells and DeepSmells under identical struc-
500 tures and embedding techniques for the semantic module. Notably, EnseSmells incorporates an additional module, the structural model. Analyzing each embedding technique, starting with the *token indexing* method, we observe that EnseSmells outperforms DeepSmells across all four types of smells. Particularly, in the case of DC smell, EnseSmells exhibits significantly higher

Figure 4: Performance of EnseSmells and DeepSmells across different embedding techniques.



performance compared to DeepSmells. This trend is similarly observed for *CodeBERT*, with a substantial increase in performance for EnseSmells in GC and DC smells. As for *code2vec*, the same pattern persists, but the considerable outperformance is clearly evident across all smells.

Notably, *CuBERT* presents an exception in LM and GC smell, where EnseSmells slightly trails DeepSmells by less than 1% in both F1 and MCC metrics. However, in FE smell, EnseSmells outperforms DeepSmells by nearly 10% in both F1-Score and MCC. The most significant disparity is observed in DC smell, where EnseSmells showcases a substantial performance improvement of approximately 40% in both F1-Score and MCC compared to DeepSmells.

Answer to RQ$_2$. This finding highlights the crucial role of software metrics in enhancing the performance of EnseSmells, underscoring their significant impact on overall effectiveness.

5.3. $\boldsymbol{RQ_3}$: How do various embedding techniques impact each category of code smell?

In Figure 5, we compare the performance of embedding techniques within the same architecture 515 for both EnseSmells and DeepSmells. This study aims to assess the performance of each embedding technique across various smells, examining the performance patterns for each method in relation to each type of smell.

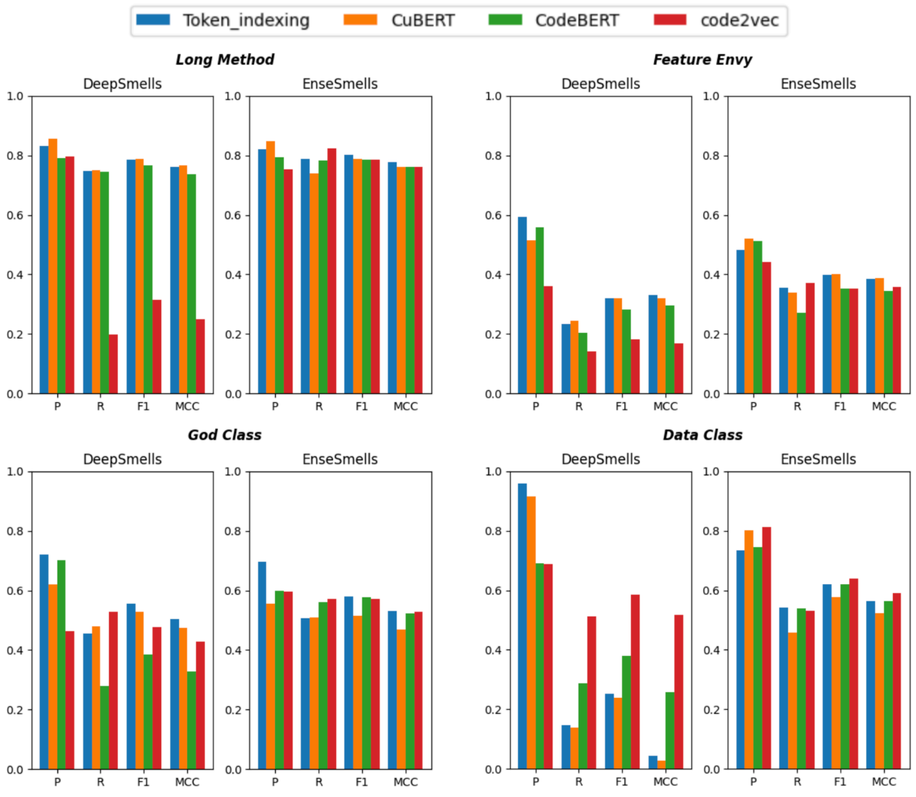


Figure 5: Performance of EnseSmells and DeepSmells in various embedding techniques.

Starting with method-level smells, the use of *code2vec* in both EnseSmells and DeepSmells exhibits the lowest performance compared to other techniques. In contrast, *token indexing* emerges 520 as a highly recommended choice due to its consistently superior performance. Notably, in this context, the consideration of *CuBERT* may be considered, given its potential impact, as it demonstrates only a slightly lower performance than *token indexing*.

Shifting focus to class-level smells, in the case of GC smell, both EnseSmells and DeepSmells employing the *token indexing* embedding technique show the best results compared to other techniques. Transitioning to DC smell, the use of *code2vec* delivers the highest performance, especially in DeepSmells, where it exhibits significant outperformance.

Answer to RQ$_3$. Each pre-trained programming language model possesses a distinct architecture, tailored to address specific code smells. In particular, *token-indexing* is effective for *Long Method*, *CuBERT* for *Feature Envy*, *token-indexing* for *God Class*, and *code2vec* for *Data Class*, showcasing the suitability of each model for different types of code smells.

5.4. ***RQ$_4$:*** How eective is EnseSmells compared with classical ML classifiers when utilizing only structural features?

Table 8 illustrates the performance of EnseSmells in comparison to seven classical machine learning models, which include *Naive Nayes (NB), Nearest Neighbor (NN), Random Forest (RF), Logistic Regression (LR), Classification and Regression Tree (CART), Back Propagation neural networks (BP),* and *Support Vector Machine (SVM)*. These machine learning models exclusively utilize software metrics. The results indicate that EnseSmells outperforms all other models, demonstrating the highest performance across all evaluated models.

Table 8: Comparing the performance of seven classifiers on software metrics with EnseSmells model.

Smell	**Metric**	**Machine learning classifier**							**EnseSmells**
		NB	**NN**	**RF**	**LR**	**CART**	**SVM**	**BP**	
LM	P	0.5698	0.8048	0.8137	0.8168	0.7595	0.7560	**0.8477**	0.8215
	R	**0.8196**	0.5310	0.6821	0.6535	0.6429	0.6901	0.6857	0.7873
	F1	0.6715	0.6360	0.7416	0.7248	0.6943	0.7199	0.7566	**0.8016**
	MCC	0.6339	0.6173	0.7150	0.6994	0.6625	0.6874	0.7347	**0.7780**
FE	P	0.1162	0.1265	0.3000	0.2000	0.4611	0.2246	0.2126	**0.5215**
	R	**0.7440**	0.1220	0.0462	0.0736	0.2099	0.1220	0.2451	0.3393
	F1	0.1993	0.1226	0.0800	0.1076	0.2784	0.1545	0.2181	**0.4025**
	MCC	0.2453	0.0987	0.1084	0.1068	0.2911	0.1441	0.1978	**0.3868**
GC	P	0.1620	0.5415	**0.7442**	0.6231	0.5296	0.6109	0.5836	0.6220
	R	**0.8308**	0.3943	0.3662	0.2744	0.4505	0.4011	0.5494	0.5867
	F1	0.2709	0.4552	0.4895	0.3760	0.4845	0.4809	0.5656	**0.5851**
	MCC	0.1280	0.3944	0.4761	0.3570	0.4166	0.4339	0.5028	**0.5429**
DC	P	0.1644	0.5415	0.6849	0.5980	0.5525	0.6343	0.5839	**0.8123**
	R	**0.8205**	0.3943	0.3979	0.2816	0.4263	0.3838	0.5393	0.5314
	F1	0.2734	0.4552	0.5019	0.3803	0.4808	0.4772	0.5573	**0.6393**
	MCC	0.1291	0.3944	0.4703	0.3541	0.4184	0.4366	0.4971	**0.5907**

Specifically, for the LM code smell, the F1-Score and MCC of EnseSmells are 80.16% and 77.80% higher than those of BP, with improvements of 4.50% and 4.33%, respectively. Notably, the precision value of BP is 2.62% higher than EnseSmells but comes at the cost of a significant reduction in recall by 10.16%. Additionally, while the recall value of NB achieves the highest at 81.96%, it is higher than EnseSmells; however, the precision, F1-Score, and MCC values are significantly lower by 25.17%, 13.01%, and 14.41%, respectively.

Regarding to FE code smell, when comparing EnseSmells to CART, which is considered the best machine learning classifier for this type of code smell, EnseSmells outperforms in all four evaluation metrics, showing improvements of 6.04%, 12.94%, 12.41%, and 9.57% in precision, recall, F1-Score, and MCC, respectively. The scenario is significantly different when comparing NB and EnseSmells. While NB achieves the highest precision value among these models at 74.40%, its recall value is only 11.62%, highlighting its weakness in handling the high imbalanced dataset.

Turning to GC code smell, while BP exhibits the best overall performance among these ML classifiers, when compared to EnseSmells, our model demonstrates outperformance in all four metrics. Furthermore, NB and RF are two models with the highest values in recall and precision, but they also exhibit significant reductions in precision and recall, respectively. Consequently, this leads to a significant decrease in both F1-Score and MCC, ranging from 6.68% to 41.49%.

As for the remaining code smell, DC, when comparing EnseSmells to BP, which demonstrates good performance across almost all types of code smells including this one, our model reveals significant differences in precision, F1-Score, and MCC, corresponding to 22.84%, 8.2%, and 9.36%. Additionally, when comparing EnseSmells to NB, which boasts the highest recall value at 82.05%, the precision value of only 16.44% highlights its weakness in handling highly imbalanced datasets leading to a remarkable decrease in both F1-Score and MCC, with a deviation of 36.59% and 46.16% lower than EnseSmells, respectively.

Answer to RQ$_4$. EnseSmells outperforms the baseline methods including all the traditional ML classifiers that build prediction models with the *structural* features.

5.5. ***RQ$_5$****:* How does EnseSmells perform compared to the baselines?

Table 9 presents a comparison between EnseSmells and the state-of-the-art approach, *ML_CuBERT* (Kovačević et al., 2022), and existing approaches that include various auto-encoder variants (Sharma

Table 9: Comparison with baseline models.

Smell	Model	Metric			
		P	**R**	**F1**	**MCC**
LM	**ML_CuBERT** (Kovačević et al., 2022)	0.6933	0.8142	0.7481	0.7182
	AE-Dense (Sharma et al., 2021)	0.7432	0.8274	0.7804	0.7548
	AE-CNN (Sharma et al., 2021)	0.7363	0.8315	0.7728	0.7500
	AE-LSTM (Sharma et al., 2021)	0.6680	**0.8480**	0.7471	0.7187
	EnseSmells	**0.8215**	0.7873	**0.8016**	**0.7780**
FE	**ML_CuBERT** (Kovačević et al., 2022)	0.1223	**0.8033**	0.2121	0.2683
	AE-Dense (Sharma et al., 2021)	0.2325	0.6295	0.3356	0.3509
	AE-CNN (Sharma et al., 2021)	0.2704	0.5333	0.3500	0.3478
	AE-LSTM (Sharma et al., 2021)	0.1364	0.7987	0.2329	0.2888
	EnseSmells	**0.5215**	0.3393	**0.4025**	**0.3868**
GC	**ML_CuBERT** (Kovačević et al., 2022)	0.4580	0.5763	0.5092	0.4492
	AE-Dense (Sharma et al., 2021)	0.5495	0.6444	0.5751	0.5306
	AE-CNN (Sharma et al., 2021)	0.5334	**0.6622**	0.5775	0.5311
	AE-LSTM (Sharma et al., 2021)	0.5227	0.6490	0.5706	0.5200
	EnseSmells	**0.6220**	0.5867	**0.5851**	**0.5429**
DC	**ML_CuBERT** (Kovačević et al., 2022)	0.4953	0.2853	0.3615	0.3081
	AE-Dense (Sharma et al., 2021)	0.1559	0.8576	0.2613	0.1068
	AE-CNN (Sharma et al., 2021)	0.1489	**0.9184**	0.2561	0.0977
	AE-LSTM (Sharma et al., 2021)	0.1532	0.8328	0.2567	0.0922
	EnseSmells	**0.8123**	0.5314	**0.6393**	**0.5907**

et al., 2021). The table demonstrates that our proposed model outperforms the other models by all evaluation metrics. Especially for LM, EnseSmells exhibits the lowest recall value at 78.73%, which is lower than the others by 2.69% to 6.07%. However, it boasts the highest precision value, surpassing the others by 7.83% to 15.35%. As a result, the overall performance demonstrates the highest F1-Score and MCC, with significant differences compared to the other models. Regarding the FE code smell, following a similar pattern LM, EnseSmells showcases an overall outperformance, with substantial differences compared to ML_CuBERT and the variants of autoencoder. The F1-Score varies from 5.25% to 19.04%, while the MCC differs from 3.59% to 11.85%.

In relation to the GC code smell, compared to AE-CNN, which shows the best performance among the baseline models with values of 53.34% (precision), 66.22% (recall), 57.75% (F1-Score), and 53.133% (MCC), EnseSmells exhibits a slightly higher in F1-Score and MCC by 0.76% and 1.18%, respectively. Notably, the precision value of EnseSmells is 8.86% higher than AE-CNN, while the recall value is only 7.55% lower.

The remaining code smell pertains to DC. The autoencoder variants exhibit a weakness in dealing with highly imbalanced datasets, showcasing very high recall values of up to 91.84%, but the precision values hover around 15%, resulting in the lowest overall performance. Moreover, EnseSmells outperforms ML_CuBERT in all four evaluation metrics by 31.7% (precision), 24.61% (recall), 27.78% (F1-Score), and 28.26% (MCC).

Answer to RQ$_5$. In real-world projects, EnseSmells achieves better prediction performance for all four types of code smells compared to the considered state-of-the-art baselines.

6. Threat to validity

- **Internal validity.** This pertains to the extent to which our evaluation mirrors real-world scenarios. In our evaluation, we utilized existing datasets (Madeyski and Lewowski, 2020) that were manually curated and classified by human experts. The quality of the curated data heavily relies on the expertise of the evaluators. Additionally, we assigned labels to each code snippet based on reviews related to four different levels of code smell severity. However, it is important to note that some of these labels may not be universally accepted, potentially impacting the overall accuracy of our predictions.
- **External validity.** This concerns the generalizability of the findings beyond the scope of this study. We attempted to mitigate threats by evaluating with different experimental configurations to simulate real-world scenarios. The findings of our work might apply only to the considered datasets. For other datasets, additional empirical evidence is required before reaching a final conclusion.
- **Construct validity.** This dimension relates to how we set up our experiments to compare EnseSmells with the baseline models. To ensure a fair comparison, we used the original implementations provided by the authors of the baseline models, maintaining their internal structures. Furthermore, our method involved measuring various software metrics using well-accepted open-source tools in our field. It is crucial to note that our study focused on the extensive MLCQ dataset, comprising a substantial number of projects (792). In such a vast dataset, there is a possibility of errors in metric calculations due to parsing issues. During our error analysis, our domain expert identified a few instances where unique word counts were miscalculated. Nevertheless, we have confidence that the use of established tools widely recognized in the research community helps minimize the impact of these potential errors.

7. Conclusion

This paper introduces EnseSmells, an innovative approach for detecting four types of code smells - two at the method level (*Long Method* and *Feature Envy*) and two at the class level (*God Class* and *Data Class*). Our methodology leverages a unique ensemble of two feature types: *semantic features* derived from novel pre-trained programming language models and *structural features* obtained through object-oriented metrics extracted by a static analysis tool. Building upon the foundation of our sophisticated model in a previous work, DeepSmells, we conduct extensive experiments to address five research questions and assess the effectiveness of EnseSmells. The results demonstrate that EnseSmells outperforms existing methods, establishing itself as the state-of-the-art for MLCQ dataset. Our future work involves expanding the application of this approach to additional code smells and identifying areas for improvement to further enhance its capabilities.

References

U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21:1143–1191, 2016.

U. Azadi, F. A. Fontana, and M. Zanoni. Poster: machine learning based code smell detection through wekanose. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 288–289. IEEE, 2018.

A. Barbez, F. Khomh, and Y.-G. Guéhéneuc. A machine-learning based ensemble method for anti-patterns detection. *Journal of Systems and Software*, 161:110486, 2020.

R. Compton, E. Frank, P. Patros, and A. Koay. Embedding java classes with code2vec: Improvements from variable obfuscation. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 243–253, 2020.

A. K. Das, S. Yadav, and S. Dhal. Detecting code smells using deep learning. In *TENCON 2019-2019 IEEE Region 10 Conference (TENCON)*, pages 2081–2086. IEEE, 2019.

J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Z. Ding, H. Li, W. Shang, and T.-H. P. Chen. Can pre-trained code embeddings improve model performance? revisiting the use of code embeddings in software engineering tasks. *Empirical Software Engineering*, 27(3):63, 2022.

Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

N. Fenton and J. Bieman. *Software metrics: a rigorous and practical approach.* CRC press, 2014.

F. A. Fontana and M. Zanoni. Code smell severity classification using machine learning techniques. *Knowledge-Based Systems*, 128:43–58, 2017.

M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., USA, 1999. ISBN 0201485672.

M. Fowler and K. Beck. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*, 1997.

J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 255–258. IEEE, 2009.

M. Hadj-Kacem and N. Bouassida. A hybrid approach to detect code smells using deep learning. In *ENASE*, pages 137–146, 2018.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

A. Ho, N. Nhat Hai, and B. Thi-Mai-Anh. Combining deep learning and kernel pca for software defect prediction. In *Proceedings of the 11th International Symposium on Information and Communication Technology*, pages 360–367, 2022.

A. Ho, A. M. Bui, P. T. Nguyen, and A. Di Salle. Fusion of deep convolutional and lstm recurrent neural networks for automated detection of code smells. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 229–234, 2023.

M. Hossin and M. N. Sulaiman. A review on evaluation metrics for data classification evaluations. *International journal of data mining & knowledge management process*, 5(2):1, 2015.

H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE transactions on software engineering*, 28(7): 654–670, 2002.

A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi. Learning and evaluating contextual embedding of source code. In *International conference on machine learning*, pages 5110–5121. PMLR, 2020.

F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui. Bdtex: A gqm-based bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, 84(4):559–572, 2011.

A. Kovačević, J. Slivka, D. Vidaković, K.-G. Grujić, N. Luburić, S. Prokić, and G. Sladić. Automatic detection of long method and god class code smells through neural source code embeddings. *Expert Systems with Applications*, 204:117607, 2022.

M. Lanza and R. Marinescu. Characterizing the design. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*, pages 23–44, 2006.

M. Lanza and R. Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems.* Springer Science & Business Media, 2007.

H. Liu, Q. Liu, Z. Niu, and Y. Liu. Dynamic and automatic feedback-based threshold adaptation for code smell detection. *IEEE Transactions on Software Engineering*, 42(6):544–558, 2015.

H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu, and L. Zhang. Deep learning based code smell detection. *IEEE transactions on Software Engineering*, 47(9):1811–1837, 2019a.

Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019b.

I. Macia, A. Garcia, and A. von Staa. Defining and applying detection strategies for aspect-oriented code smells. In *2010 Brazilian Symposium on Software Engineering*, pages 60–69. IEEE, 2010.

L. Madeyski and T. Lewowski. Mlcq: Industry-relevant code smell data set. In *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, pages 342–347, 2020.

A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, and E. Aimeur. Smurf: A svm-based incremental anti-pattern detection approach. In *2012 19th Working Conference on Reverse Engineering*, pages 466–475. IEEE, 2012a.

A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aimeur. Support vector machines for anti-pattern detection. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 278–281, 2012b.

R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance. Proceedings.*, pages 350–359. IEEE, 2004.

R. Marinescu. Measurement and quality in object-oriented design. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 701–704. IEEE, 2005.

R. Marinescu, G. Ganea, and I. Verebi. Incode: Continuous quality assessment and improvement. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 274–275. IEEE, 2010.

T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1): 20–36, 2009.

M. J. Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In *11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 15–15. IEEE, 2005.

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

S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 31–41, 2018.

V. Sales, R. Terra, L. F. Miranda, and M. T. Valente. Recommending move method refactorings using dependency sets. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 232–241. IEEE, 2013.

M. Sharma and G. Singh. Analysis of static and dynamic metrics for productivity and time complexity. *International Journal of Computer Applications*, 30(1):7–13, 2011.

T. Sharma, P. Mishra, and R. Tiwari. Designite: A software design quality assessment tool. In *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*, pages 1–4, 2016.

T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis. Code smell detection by deep direct-learning and transfer-learning. *Journal of Systems and Software*, 176:110936, 2021.

S. Singh and S. Kaur. A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Engineering Journal*, 9(4):2129–2151, 2018.

G. Suryanarayana, G. Samarthyam, and T. Sharma. *Refactoring for software design smells: managing technical debt.* Morgan Kaufmann, 2014.

B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on software engineering*, 33(12):800–817, 2007.

A. Vesra. A study of various static and dynamic metrics for open source software. *International Journal of Computer Applications*, 122(10), 2015.

S. Vidal, H. Vazquez, J. A. Diaz-Pace, C. Marcos, A. Garcia, and W. Oizumi. Jspirit: a flexible tool for the analysis of code smells. In *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–6. IEEE, 2015.

J. Von der Mosel, A. Trautsch, and S. Herbold. On the validity of pre-trained transformers for natural language processing in the software engineering domain. *IEEE Transactions on Software Engineering*, 49(4):1487–1507, 2022.

Y. Xiao, J. Keung, K. E. Bennin, and Q. Mi. Improving bug localization with word embedding and enhanced convolutional neural networks. *Information and Software Technology*, 105:17–29, 2019.

J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu. Retrieval-based neural source code summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1385–1397, 2020.

M. Zhang, T. Hall, and N. Baddoo. Code bad smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: research and practice*, 23(3):179–202, 2011.

Y. Zhang, C. Ge, S. Hong, R. Tian, C. Dong, and J. Liu. Delesmell: Code smell detection based on deep learning and latent semantic analysis. *Knowledge-Based Systems*, 255:109737, 2022.

Appendix

Table 10: The feature description and abbreviation for all object-oriented metrics (Arcelli Fontana et al., 2016).

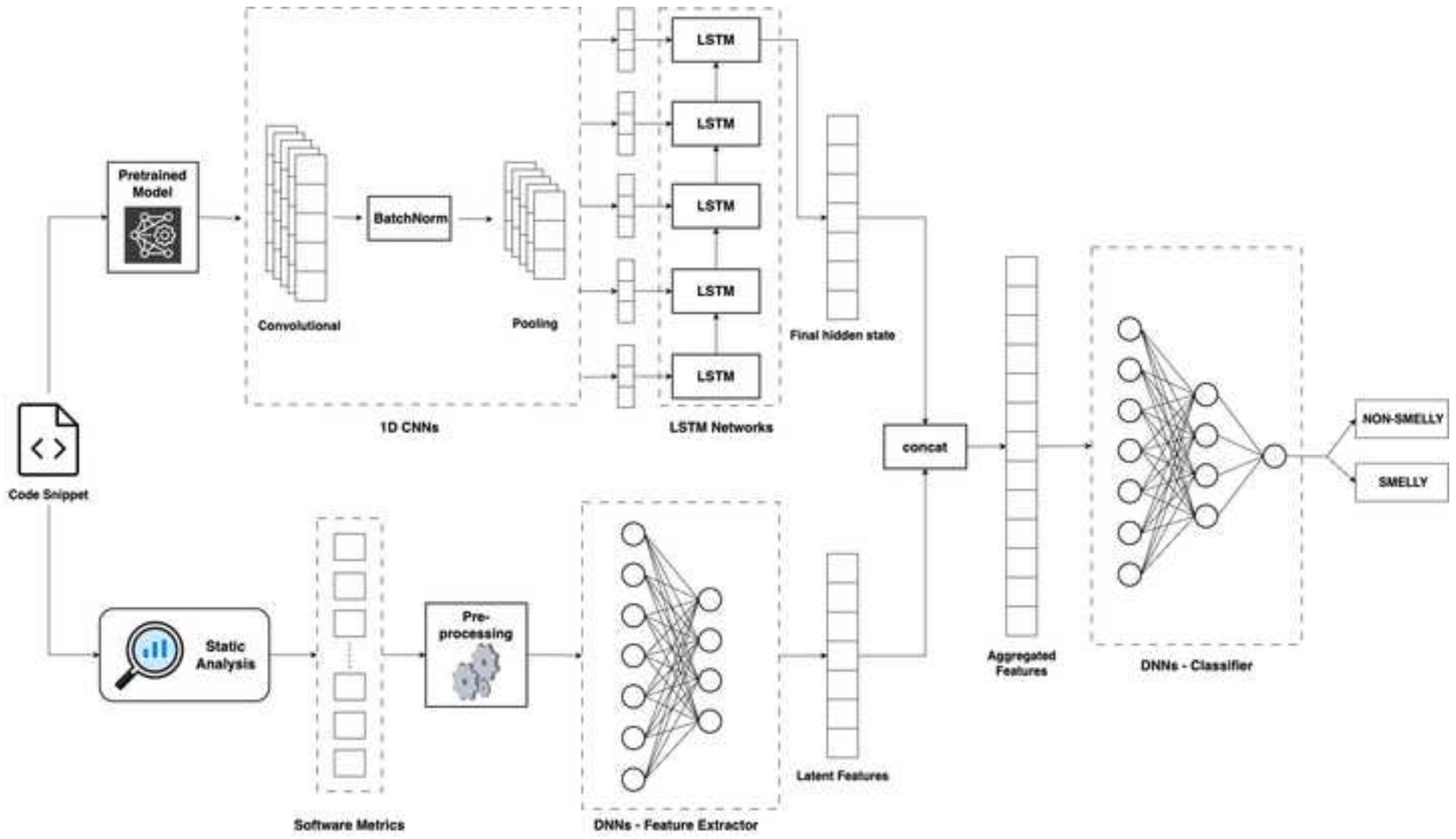
Quality Dimension	**Metric label**	**Metric name**	**Granularity**
Size	LOC	Lines of Code	Project, Package, Class Method
	LOCNAMM	Lines of Code Withour Accessor or Mutator Methods	Class
	NOPK	Number of Packages	Project
	NOCS	Number of classes	Project, Package
	NOM	Number of Methods	Project, Package, Class
	NOMNAMM	Number of Not Accessor or Mutator Methods	Project, Package, Class
	NOA	Number of Attributes	Class
Complexity	CYCLO	Cyclomatic Complexity	Method
	WMC	Weighted Methods Count	Class
	WMCNAMM	Weighted Methods Count of Not Accessor or Mutator Methods	Class
	AMW	Average Methods Weight	Class
	AMWNAMM	Average Methods Weight of Not Accessor or Mutator Methods	Class
	MAXNESTING	Maximum Nesting Level	Method
	CLNAMM	Called Local Not Accessor or Mutator Methods	Method
	NOP	Number of Parameters	Method
	NOAV	Number of Accessed Variables	Method
	ATLD	Access to Local Data	Method
	NOLV	Number of Local Variable	Method
Coupling	FANOUT	-	Class, Method
	FANIN	-	Class
	ATFD	Access to Foreign Data	Method
	FDP	Foreign Data Providers	Method
	RFC	Response for a Class	Class
	CBO	Coupling Between Objects Classes	Class
	CFNAMM	Called Foreign Not Accessor or Mutator Methods	Class, Method
	CINT	Coupling Intensity	Method
	MaMCL	Maximum Message Chain Length	Method
	MeMCL	Mean Message Chain Length	Method
	NMCS	Number of Message Chain Statements	Method
	CC	Changing Classes	Method
	CM	Changing Methods	Method
Encapsulation	NOAM	Number of Accessor Methods	Class
	NOPA (NOAP)	Number of Public Attribute	Class
	LAA	Locality of Attribute Accesses	Method
Inheritance	DIT	Depth of Inhertitance Tree	Class
	NOI	Number of Interfaces	Project, Package
	NOC	Number of Children	Class
	NMO	Number of Methods Overridden	Class
	NIM	Number of Inherited Methods	Class
	NOII	Number of Implemented Interface	Class

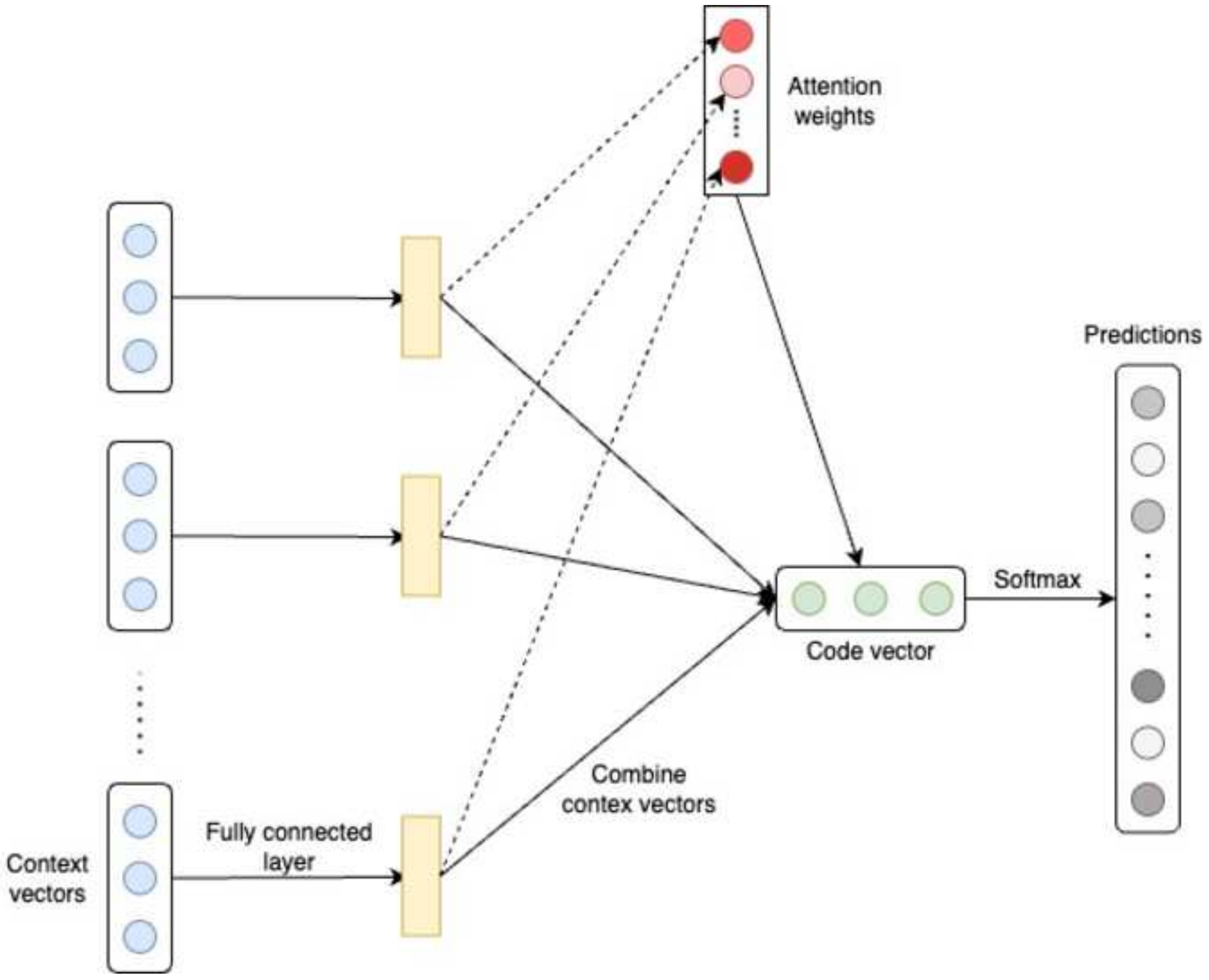
Table 11: The feature description and abbreviation for class level.

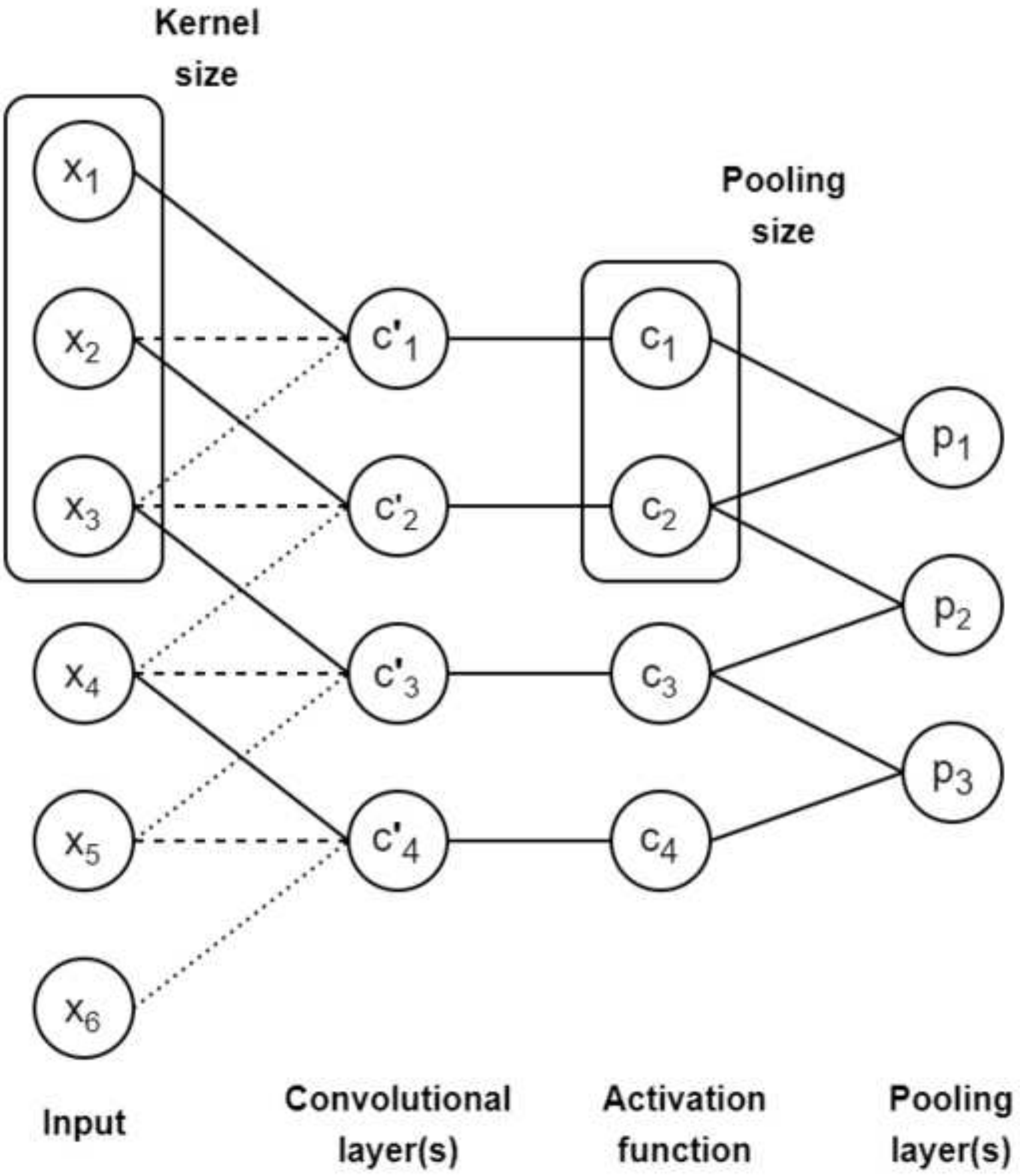
Software metric	Abbreviation	Software metric	Abbreviation
Number of abstract methods	abstractMethodsQty	Number of private fields	privateFieldsQty
Quantity of anonymous classes	anonymousClassesQty	Number of private methods	privateMethodsQty
Quantity of assignments	assignmentsQty	Number of protected fields	protectedFieldsQty
Coupling between objects	cbo	Number of protected methods	protectedMethodsQty
Quantity of comparisons	comparisonsQty	Number of public fields	publicFieldsQty
Number of defualt fields	defaultFieldsQty	Number of public methods	publicMethodsQty
Number of default methods	defaultMethodsQty	Quantity of returns	returnQty
Depth Inheritance Tree	dit	Response for a Class	rfc
Number of final fields	finalFieldsQty	Number of static fields	staticFieldsQty
Number of final methods	finalMethodsQty	Number of static methods	staticMethodsQty
Weight Method Class	wmc	The number of string literals	stringLiteralsQty
Quantity of inner classes	innerClassesQty	Number of synchronized fields	synchronizedFieldsQty
Quantity of lambda expressions	lambdasQty	Number of synchronized methods	synchronizedMethodsQty
Loose Class Cohesion	lcc	Tight Class Cohesion	tcc
Lack of Cohesion of Methods	lcom	Total number of fields	totalFieldsQty
Lines of code	loc	Total number of methods	totalMethodsQty
Number of Log Statements	logStatementsQty	Quantity of try/catches	tryCatchQty
Quantity of loops	loopQty	Number of visible fields	visibleFieldsQty
Quantity of math operations	mathOperationsQty	Quantity of numbers	numbersQty
Max nested blocks	maxNestedBlocksQty	Quantity of parenthesized expressions	parenthesizedExpsQty
Modifiers	modifiers	Number of unique words	uniqueWordsQty
Number of static invocations	nosi	Quantity of Variables	variablesQty

Table 12: The feature description and abbreviation for method level.

Software metric	Abbreviation	Software metric	Abbreviation
Is constructor	constructor	Quantity of try/catches	tryCatchQty
Total numbers of line in method	line	Quantity of parenthesized expressions	parenthesizedExpsQty
Coupling between objects	cbo	The number of string literals	stringLiteralsQty
Weight Method Class	wmc	Quantity of numbers	numbersQty
Response for a Class	rfc	Quantity of assignments	assignmentsQty
Lines of code	loc	Quantity of Math Operations	mathOperationsQty
Quantity of returns	returnsQty	Max nested blocks	maxNestedBlocksQty
Quantity of variables	variablesQty	Quantity of anonymous classes	anonymousClassesQty
Quantity of parameters	parametersQty	Quantity of inner classes	innerClassesQty
Method invocations	methodsInvokedQty	Quantity of lambda expressions	lambdasQty
Quantity of local method invocations	methodsInvokedLocalQty	Number of unique words	uniqueWordsQty
Quantity of indirect local method invocations	methodsInvokedIndirectLocalQty	Modifiers	modifiers
Quantity of loops	loopQty	Number of Log Statements	logStatementsQty
Quantity of comparisons	comparisonsQty	Has Javadoc	hasJavaDoc


Pretrained Model
Convolutional
BatchNorm
Pooling
1D CNNs
LSTM
LSTM
LSTM
LSTM
LSTM
LSTM Networks
Final hidden state
Code Snippet
concat
Aggregated Features
DNNs - Classifier
NON-SMELLY
SMELLY
Static Analysis
Software Metrics
Pre-processing
DNNs - Feature Extractor
Latent Features


Attention
weights
Predictions
Softmax
Code vector
Combine
contex vectors
Fully connected
layer
Context
vectors


Kernel size
Pooling size
x_1
x_2
x_3
x_4
x_5
x_6
c'_1
c'_2
c'_3
c'_4
c_1
c_2
c_3
c_4
p_1
p_2
p_3
Input
Convolutional layer(s)
Activation function
Pooling layer(s)