e-Informatica Software Engineering Journal
Volume 19, Issue 1, pages: 1–32
DOI: 10.37190/e-Inf250106

BibTeX

A Novel Approach for Enhancing Code Smell Detection Using Random Convolutional Kernel Transform

Mostefai Abdelkader[*], Mekour Mansour
[*]Corresponding author: abdelkader.mostefai@univ-saida.dz

Article info

Dataset link: https://zenodo.org/records/15602620

Keywords:
Code smell
Maintenance
Classification
Random convolutional transform
Deep learning

Submitted: 05 Sep. 2024
Revised: 20 Jun. 2025
Accepted: 7 Jul. 2025
Available online: 8 Jul. 2025

Abstract

Context: In software engineering, the presence of code smells is closely associated with increased maintenance costs and complexities, making their detection and remediation an important concern.
Objective: Despite numerous deep learning approaches for code smell detection, many still heavily rely on feature engineering processes (metrics) and exhibit limited performance. To address these shortcomings, this paper introduces CSDXR, a novel approach for enhancing code smell detection based on Random Convolutional Kernel Transform – a state-of-the-art technique for time series classification. The proposed approach does not rely on a manual feature engineering process and follows a three-step process: first, it converts code snippets into numerical sequences through tokenization; second, it applies Random Convolutional Kernel Transform to generate pooled models from these sequences; and third, it constructs a classifier from the pooled models to identify code smells.
Method: The proposed approach was evaluated on four real-world datasets and compared against four state-of-the-art methods – DeepSmells, AE-Dense, AE-CNN, and AE-LSTM – in detecting Complex Method, Multifaceted Abstraction, Feature Envy, and Complex Conditional smells.
Results: Empirical results demonstrate that CSDXR outperformed the four state-of-the-art methods – DeepSmells, AE-Dense, AE-CNN, and AE-LSTM – in detecting Complex Method and Multifaceted Abstraction smells. Specifically, the enhancement rates in terms of F_1-score were 1.99% and 6.09% for Complex Method and Multifaceted Abstraction smells, respectively. In terms of MCC, the improvement rates were 0.82% and 35.64% for these two smells, respectively. The results also show that while DeepSmells achieves superior overall performance on Feature Envy and Complex Conditional smells, CSDXR surpasses AE-Dense, AE-CNN, and AE-LSTM in detecting these two types of smells.
Conclusions: The paper concludes that the proposed approach, CSDXR, demonstrates significant potential for effectively detecting various types of code smells.

1. Introduction

Software plays an increasingly pivotal role in many aspects of modern life. As these systems become more complex and reliance on them continues to grow, maintaining them becomes ever more critical. To maintain their expected value, software systems require regular upkeep. In a highly competitive environment, developers often employ design and implementation strategies aimed at speeding up time-to-market. However, such practices can exacerbate technical debt [1], which refers to the long-term costs associated with suboptimal design and implementation decisions. While these decisions may offer immediate benefits, such as faster product releases or enhanced client satisfaction, they often undermine the software's quality and lead to costly future maintenance.

Code smells are indicators of poor code design that contribute to technical debt. They manifest in various parts of the code, such as classes or method statements, and arise from inadequate design or implementation choices. These decisions can be intentional, where developers are aware of the trade-offs, or unintentional [2, 3]. Extensive research has highlighted the detrimental impact of code smells on software quality, identifying them as a significant manifestation of technical debt [2] and emphasizing the need for effective detection, filtration and prioritization approaches[4, 5].

Manual detection of code smells is challenging [6], prompting the development of various automatic detection techniques. These methods are generally categorized into deep learning, machine learning, heuristics, and metrics-based approaches [4, 7, 8]. Metric-based and heuristic-based approaches are popular but often rely on costly manual processes involving designed heuristics and selected features.

These processes require significant manual intervention, including configuring and customizing analysis tools to suit specific needs, determining which code aspects to measure, selecting the appropriate metrics, setting thresholds, and fine-tuning these thresholds for specific contexts and projects. Additionally, interpreting the results of these metrics to classify a piece of code as a "smell" requires domain expertise, making the task labor-intensive.

Consequently, these manual interventions make the processes time-consuming and costly, highlighting the inherent limitations of such approaches.

Machine learning methods, on the other hand, depend on external tools to compute features (e.g., metrics) from the source code, making their effectiveness contingent upon these tools and the expert-defined features. However, different tools can yield varying results for the same metric, even when the metric is conceptually the same (e.g., lines of code or cyclomatic complexity). These discrepancies arise from variations in how each tool defines, computes, or interprets the metric. Since tools may apply slightly different algorithms, rules, or default settings, the values they produce can differ.

For example, tools calculating cyclomatic complexity may handle control flow constructs, such as loops or exception handling, differently, leading to variations in the final complexity score. Similarly, tools measuring lines of code (LOC) might differ in their definitions of what constitutes a line.

Additionally, variations in results can stem from the parsers or lexers used. Each parser may interpret code differently based on how it handles syntax, grammar, or language-specific rules, which in turn affects the features extracted for metric computation.

In the absence of a standardized tool, these differences in the tools used can significantly impact the results, and consequently, the performance of code smell detectors.

Despite the variety of existing techniques, many remain underdeveloped and ineffective [3, 4, 7, 9]. Thus, there is a pressing need for advanced techniques to enhance code smell detection models. Recent studies have explored deep learning models that minimize manual feature engineering by automatically learning features from source code[7, 9]. However, these models face limitations, including specificity to particular code smells and overall limited performance [7, 9, 10]. Xu and Zhang [10] argue that these limitations stem from token-based representations of code, which lose rich semantic and structural information. Nevertheless, we believe that deep learning models can still effectively extract meaningful representations from raw token sequences for code smell detection, as demonstrated by Ho et al. [9]. We propose that advanced time series classification (TSC) techniques could address these shortcomings.

The TSC field has seen significant advancements, with numerous techniques for time series representation and classification achieving success in various domains, including finance, Internet of Things, cloud computing, energy, transportation, code clone detection and social networks [11–16]. We propose that source code can be converted to an ordered sequence of tokens and treated as a time series. By leveraging TSC algorithms, we aim to improve code smell detection, inspired by their success in code clone detection [16].

Our thesis is that in kernel-based approaches such as XRocket (e.g., MiniRocket and Rocket), kernels serve as tools for detecting specific patterns by convolving them over the time series. The result of convolving each kernel is an activation map that indicates the location and strength of this pattern. A pooling operator then is used to summarize this activation map into a single feature (i.e., a single number). With n kernels and m pooling operators, a time series representation consisting of $n \times m$ features is created. The resulting representations is then used to train a classifier to classify new instances. Consequently, for a code smell characterized by identifiable patterns, it becomes feasible to detect such smells using carefully designed kernels, appropriate pooling operators, and a suitable classifier. This detection process works on time series data derived from source code to be checked for “smelliness.”

Drawing inspiration from the success of TSC methods and guided by our thesis, we introduce CSDXR, a novel approach for code smell detection based on MINImally RandOm Convolutional KErnel Transform (MiniRocket) and RandOm Convolutional KErnel Transform (Rocket). Our approach hypothesizes that using MiniRocket or Rocket to pool representations of code snippets will outperform previous methods. CSDXR converts source code snippets into time series, then uses MiniRocket (CSDMR) or Rocket (CSDR) to pool representations. A classifier is then trained on these representations, labeled as either smelly or non-smelly, and used to predict the presence of code smells in new source code. The proposed CSDXR method does not rely on a manual feature engineering process.

Main Contributions:

– Introduction of a novel method does not rely on feature engineering process for code smells detection.
– Introduction of a novel method based on MiniRocket and Rocket for modelling source code.
– Evaluation of the method’s effectiveness in detecting four specific code smells: Complex Method, Complex Conditional, Feature Envy, and Multifaceted Abstraction.

The rest of this paper is organized as follows: Section 2 provides background on code smells, Rocket, and MiniRocket. Section 3 reviews the state-of-the-art code smell detection approaches. Section 4 details our proposed approach. Section 5 presents our empirical study, Section 6 presents and discuss the results of the empirical study. Section 7 addresses threats to validity, and Section 8 concludes the paper and outlines directions for future work.

2. Background

This section provides the background information necessary for understanding the approach.

2.1. Code smell

The term *code smell* as first introduced by Kent Beck in the 1990s [17]. Code smells are indicators of poor code quality in various code elements such as classes, methods, or statements, and they often lead to increased technical debt. These smells typically signal violations of design principles and best practices, arising from suboptimal design and implementation decisions.

The concept of code smells gained wider recognition through Martin Fowler's book, which detailed 22 types of code smells and their corresponding refactoring solutions [17]. Examples of common code smells include Feature Envy, God Class, Duplicated Code, Long Method, Long Switch, and Long Parameter List. For more comprehensive information about code smells, refer to [17, 18].

In this paper, we evaluate the proposed approach on four specific code smells [18]:
- **Complex Method (CM):** A method characterized by high cyclomatic complexity.
- **Complex Conditional (CC):** A conditional statement with a complex condition expression (e.g., an intricate if statement).
- **Feature Envy (FE):** A method that is more interested in the details of a different class than the class it is in.
- **Multifaceted Abstraction (MA):** A class that has multiple, unrelated responsibilities.

Complex Method and Complex Conditional are implementation-level smells, while Feature Envy and Multifaceted Abstraction are design-level smells.

2.2. Rocket and MiniRocket

Time series data consists of ordered sequences, such as temporal data, where each data point is associated with a specific time. Time Series Classification (TSC) involves predicting the class of a time series based on previously classified series. According to Bagnall et al. [11], the order of attributes in time series data differentiates TSC from traditional classification problems, necessitating that the representation process creates discriminative and meaningful features by accounting for this temporal structure.

Two state-of-the-art techniques for time series classification are the RandOm Convolutional KErnel Transform (ROCKET) [19] and its variant, MINImally RandOm Convolutional KErnel Transform (MiniROCKET) [20]. Both methods are inspired by convolutional neural networks (CNNs) but differ in their approach to kernel generation and application.

The MiniRocket and Rocket methods compute a representation of a time series by first convolving it with a set of k kernels. In the case of Rocket, these kernels are randomly generated, whereas in MiniRocket, they are designed based on predefined rules.

Second, the activation map, which results from the convolution of each kernel with the time series, is summarized using pooling operators. Rocket utilizes two pooling operators, Proportion of Positive Values (PPV) and Max, to generate two features per kernel, resulting in a feature vector with $2k$ features. In contrast, MiniRocket uses only the PPV operator, producing a representation with k features.

ROCKET uses a large set of randomly generated convolutional kernels, typically 10,000 by default. These kernels vary in length and dilation, and are employed to transform the input time series into a feature vector.

The kernel initialization in ROCKET follows a random process with the following parameters:
1. **Kernel Length (l)**: Randomly selected from $\{7, 9, 11\}$.
2. **Kernel Weights (w)**: Randomly initialized from a normal distribution.
3. **Bias Term (b)**: Added to the result of the convolution operation.
4. **Dilation (d)**: Determines the spread of the kernel weights over the input time series. Dilation allows similar kernels with different dilation values to detect patterns at various frequencies and scales. For example, a kernel $[2\,{-1}\,1]$ with $d = 1$ becomes $[2\,0\,{-1}\,0\,1]$, and with $d = 3$, it becomes $[2\,0\,0\,0\,{-1}\,0\,0\,0\,1]$.
5. **Padding (p)**: Adds zeros to the start and end of the input series to ensure the activation map and the input series are of the same length.

The result of applying a kernel ω with dilation d to a time series T at offset i is defined as:

$$T_{i:(i+l)} * w = \sum_{j=0}^{l-1} T_{i-\left(\lfloor \frac{m}{2} \rfloor \times d\right)+(j\times d)} \times w_j \tag{1}$$

MiniROCKET is a variant of ROCKET that retains the core principles but is designed to be more computationally efficient. It uses a smaller number of kernels, reducing the computational burden while maintaining performance [20].

MiniROCKET utilizes a set of predefined kernels with a fixed length of 9 and two possible weight values $\{-1, 2\}$, applying 84 fixed convolutions. Unlike ROCKET, which computes two features per kernel, MiniROCKET computes only one feature per kernel. These design choices make MiniROCKET significantly faster – up to 75 times – compared to ROCKET, while maintaining performance comparable to other models.

3. State of the art

Many approaches have been proposed to detect smells in software systems, classified into deep learning [3,21], machine learning[8], heuristics, and metrics-based methods [4,7–10,22].

3.1. Metrics-Based Smell Detection Methods [21,23]

Metrics-based approaches, in particular, are widely used for code smell detection. Software metrics are a common method for assessing software quality, evaluating factors and attributes such as cohesion and coupling within a system [24]. A clean codebase typically exhibits metric values within ranges defined by experts [Livre Software Quality], reflecting adherence to software design principles and best practices.

Code fragments are considered smelly if they violate these principles. Such violations can be identified by measuring the design attributes of the code fragment and comparing them to values from clean (non-smelly) code. For example, the Feature Envy smell is an indicator of poor cohesion and coupling [17]. Metrics-based approaches detect smells by applying formulas that use filters and thresholds on a set of metrics computed from the source code [25]. For instance, a God Class smell can be detected using metrics such as

ATFD (Access to Foreign Data), WMC (Weighted Methods per Class), and TCC (Tight Class Cohesion) [26].

An example of such formulas is defined by Marinescu [27] for detecting ten different code smells. Macia et al. [28] proposed thresholds and formulas that combine eight metrics for detecting aspect-oriented smells. Fard and Mesbah [29] introduced a method called JSNOSE to detect JavaScript code smells, which is metrics-based and combines static and dynamic analysis. Chen et al. [30] defined ten code smells specific to the Python language and proposed a metrics-based method to detect them. Their study utilized and compared thresholds specified by three filtering strategies: the Experience-Based Strategy, the Statistics-Based Strategy, and the Tuning Machine Strategy. This research was conducted on a dataset of 106 Python projects.

3.2. Rules/heuristic-based smell detection methods

In this category, the method's input is a source code model and, optionally, a set of software metrics. Detection is performed using a set of predefined rules or heuristics. These methods rely on specified rules or heuristics and leverage source code models, and optionally metrics, for detecting code smells and, principally, design smells [18, 31]. Rule-based approaches depend on manually specified rules [32]. For instance, DÉCOR [27] relies on expert-designed rules, which must be expressed in a domain-specific language. However, this design process is costly. The DÉCOR approach was validated on the software XERCES v2.7.0.

3.3. Machine learning-based smell detection

In this category, classifiers such as Support Vector Machines (SVM) or Naïve Bayes (NB) are trained on datasets specific to a particular smell. The dataset typically consists of computed models (representations) of code fragments. Once trained, these models are used to predict the class of new code fragments (i.e., whether they are smelly or not). Metrics-based representations are commonly employed in these methods [4, 7, 10].

Maiga et al. [33] proposed an approach called SVMDetect to detect anti-patterns in software systems. This approach leverages SVM, a well-known machine learning algorithm. An empirical study conducted on three systems and four anti-patterns demonstrated that SVMDetect is more accurate than DETEX.

Khomh et al. [34] introduced a process to transform detection rules into a probabilistic model, with a demonstration conducted on the Blob anti-pattern.

Kreimer [35] proposed a method based on decision trees to detect design flaws (code smells) in object-oriented software.

3.4. Deep learning-based smell detection

Sharma et al. [7] proposed an approach for code smell detection based on a deep learning model that combines Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), and autoencoder models. The authors built a dataset from 922 C# and 922 Java repositories downloaded from GitHub. The proposed approach aims to leverage the power of these models without relying on the feature engineering process commonly used by most code smell detection methods (e.g., metrics). The authors also investigated the potential of transfer learning in code smell detection by training the model on C# projects and evaluating the results on Java projects. The empirical study conducted to detect

Feature Envy, Complex Method, Complex Conditional, and Multifaceted Abstraction smells indicated that the results are smell-specific. This means there is no simple, universal solution for detecting all types of smells. The results also indicated that deep learning models without a feature engineering process still require improvement in performance.

Ho et al. [9] proposed an approach called DeepSmells to address the limitations of the method proposed by Sharma et al. [7]. DeepSmells incorporates both structural and semantic features from software units and mitigates the effects of imbalanced data distribution. To achieve this, the method combines Convolutional Neural Networks (CNN) with Long Short-Term Memory networks (LSTM) to learn hierarchical representations of source code, preserve semantic information, and improve the quality of context encoding. The output of this stage is fed into a deep neural network classifier with a weighted loss function to counteract the effects of skewed data distribution. Empirical studies demonstrated that this approach outperforms state-of-the-art tools.

Skipina et al. [36] evaluated and compared machine learning models using code representations based on metrics versus representations based on neural code embeddings (CodeT5 and CuBERT) for detecting Data Class and Feature Envy smells. The evaluation was conducted on the MLCQ dataset, and the results showed no significant differences in performance between the two approaches. However, code embeddings were found to be more scalable and have the potential to adapt to new programming languages.

Hadj-Kacem and Bouassida [37] proposed a combined method using deep autoencoders and Artificial Neural Networks (ANN) for detecting God Class, Data Class, Feature Envy, and Long Method smells. In this method, the autoencoder reduces data dimensionality, and the ANN is used as the classifier. The empirical study indicated that this method is effective, achieving an F-measure of 98.93% for the God Class code smell.

Liu et al. [38] proposed a deep learning-based method for detecting the Feature Envy code smell. The model employed is a Convolutional Neural Network (CNN), where the input is a combination of text and numerical data. The text, consisting of a sequence of the method's name, the class name, and the target class name, is embedded to produce a numerical representation. The evaluation, conducted on seven well-known open-source projects, showed that the method outperforms state-of-the-art tools, achieving an F-measure of 34.32%.

Bo Liu et al. [39] proposed an approach called **feTruth** aimed at improving deep learning models dedicated to detecting the Feature Envy code smell. This objective is achieved by filtering out false positives produced by state-of-the-art tools using a set of heuristics and a decision tree classifier.

Das et al. [40] proposed a deep learning method based on Convolutional Neural Networks (CNN) to detect Brain Class and Brain Method code smells.

Yu et al. [41] proposed a method based on Graph Neural Networks (GNN) for Feature Envy detection. The method leverages code metrics and calling relationships to address the challenge posed by calling relationships between methods, which can hinder the detection process. The evaluation on five open-source projects showed that the performance, in terms of F_1-score, was 37.98% higher than state-of-the-art tools.

Hanyu et al. [42] introduced a deep learning approach based on a Graph Convolutional Network (GCN) for Long Method detection. This model builds a graph neural network by inputting two types of information: nodes and edges. The nodes represent methods and statements, while the edges represent include, control flow, control dependency, and data dependency relationships. The evaluation was based on five groups of datasets.

Zhang et al. [43] proposed a method called **DeleSmell** that combines deep learning and Latent Semantic Analysis (LSA) to detect Brain Class and Brain Method code smells. The deep learning model consists of two branches: a Convolutional Neural Network (CNN) branch and a Gated Recurrent Unit (GRU)-attention branch. A Support Vector Machine (SVM) classifier is used at the final stage. The approach aims to address the issues of incomplete feature representation and unbalanced distribution between positive and negative samples. The evaluation was conducted on a dataset built from 24 real-world projects, with the dataset balanced using a refactoring tool developed for this purpose. The results indicated an improvement of 4.41% in F_1-score compared to state-of-the-art methods.

Xu and Zhang [10] proposed a method for detecting Feature Envy, Insufficient Modularization, Empty Catch Block, and Deficient Encapsulation code smells. The method is based on a deep learning model and Abstract Syntax Trees (ASTs) and does not rely on a feature engineering process. The objective was to overcome the limitations of token-based approaches by leveraging the semantic and structural information of the source code. The experimental results indicate its superiority compared to state-of-the-art approaches for detecting code smells.

Zhang and Dong [44] proposed the **MARS** approach for detecting Brain Class and Brain Method smells. The approach aims to solve the gradient degradation problem using an improved residual network. It employs a metric-attention mechanism to increase the weight value of important code metrics. The approach was evaluated on the BrainCode dataset, which was built from 20 real-world applications. The experimental results show that the average accuracy of MARS is 2.01% higher than state-of-the-art tools.

Li and Zhang [45] proposed a method to optimize code smell detection through a hybrid model with multi-level code representation. In this approach, the result is a function of two predictions at the syntactic, semantic, and token levels. The prediction at the syntactic and semantic levels is computed using a Graph Convolution Network (GCN) that takes as input the AST with control and data flow edges of the source code. The token-level prediction is calculated using a bidirectional Long Short-Term Memory (LSTM) network with an attention mechanism. Experimental results demonstrate that the method performs better in both single code smell detection and multi-label code smell detection cases.

Liu et al. [46] proposed a method based on Convolutional Neural Networks (CNN) and a text embedding technique (i.e., Word2Vec). The CNN model is fed a representation of a code fragment computed using the Word2Vec approach.

For further details, see [21, 23].

4. The proposed approach

In this section, we present an overview of our proposed method, CSDXR, for code smell detection based on the random convolutional transform method. As illustrated in Figure 1, CSDXR combines MiniRocket (CSDMR) or Rocket (CSDR) with advanced classifiers. The proposed method consists of three basic steps. First, the method converts a code snippet into a time series using a tokenization technique. In the second step, MiniRocket (or Rocket) is employed to generate a model of the obtained sequence. Finally, in the third step, CSDXR uses the pooled models to build a classifier for detecting code smells.

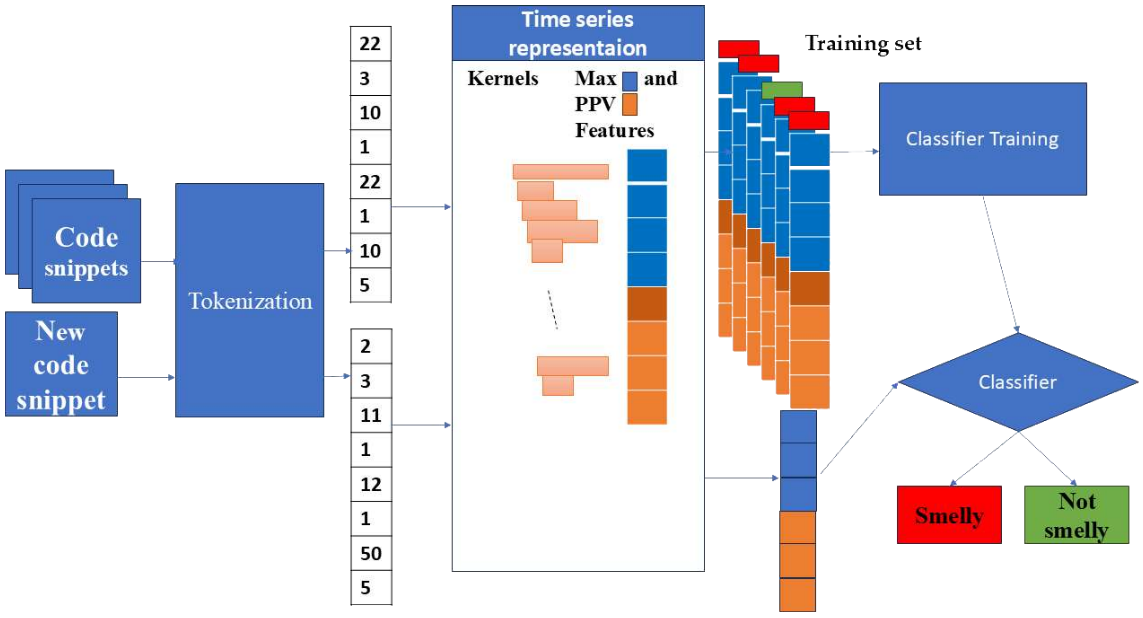


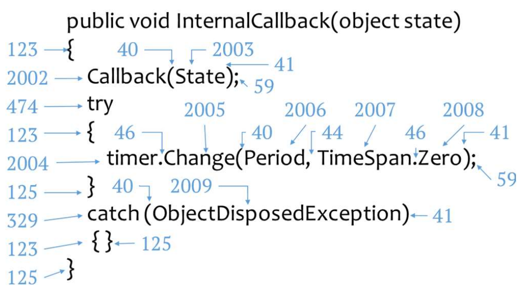
Figure 1. The proposed approach

4.1. Tokenization

The objective of this step is to convert a code snippet into a sequence of numbers that can be treated as a time series. The CSDXR method builds upon the tokenization algorithm provided by Sharma et al. [7]. Sharma et al. [7] released the full pipeline of their deep learning approach and encouraged researchers to extend it, aiming to fully explore the potential of code smell detection methods that do not rely on feature engineering.

The tokenization process works as follows: the code snippet is first decomposed into a sequence of tokens using a lexical analyzer. Each unique token is then assigned a specific number. For example, the token "{" might be assigned the number 123, the token "(" might be assigned the number 40, and so forth. An example of the result of the tokenization of a code snippet is illustrated in Figure 2.

The output of this step is a sequence of numbers (i.e., a time series).



```
public void InternalCallback(object state)
{
    Callback(State);
    try
    {
        timer.Change(Period, TimeSpan.Zero);
    }
    catch (ObjectDisposedException)
    {}
}
```

Figure 2. An example of the result of the tokenization of a code snippet [7]

4.2. Time series modelling

The objective of this phase is to create a representation of the code snippet from the time series produced in the previous step. This is achieved by employing the MiniRocket algorithm or a similar algorithm (e.g., Rocket).

The modelling process works as follows:

1. The steps involved in this process are collectively referred to as the representation phase. In this phase, first, a fixed set of k kernels is produced. Next, each kernel is convolved over the time series. The result of convolving each kernel is an activation map that indicates the location and strength of the pattern. Subsequently, pooling operators are used to summarize this activation map into a set of features. With n kernels and m pooling operators, a time series representation consisting of $n \times m$ features is created. For example, in the case of ROCKET, two pooling operators are used: PPV and MAX. The kernels are randomly generated from the set { 7, 9, 11 }, and their weights are randomly sampled from a normal distribution. In contrast, for MiniROCKET, this process is quasi-deterministic, and only one pooling operator is used (PPV). The kernels have a size of 9, and their weights are randomly selected from the set {-1,2}. For example, the application of the kernel $\mathbf{w} = [-1, 0, 1]$ to the sequence $[0, 1, 3, 2, 9, 1, 1, 15, 4, 9]$ is illustrated in Figure 3. This Figure shows that the obtained activation map by convolving the kernel $\mathbf{w} = [-1, 0, 1]$ over this time series is $[3, 1, 6, -1, -8, 14, 3, -6]$. The feature obtained using the PPV operator is 5/8, and for the MAX pooling operator, it is 14. Finally, all the features obtained by convolving the k kernels are concatenated to form the time series model.

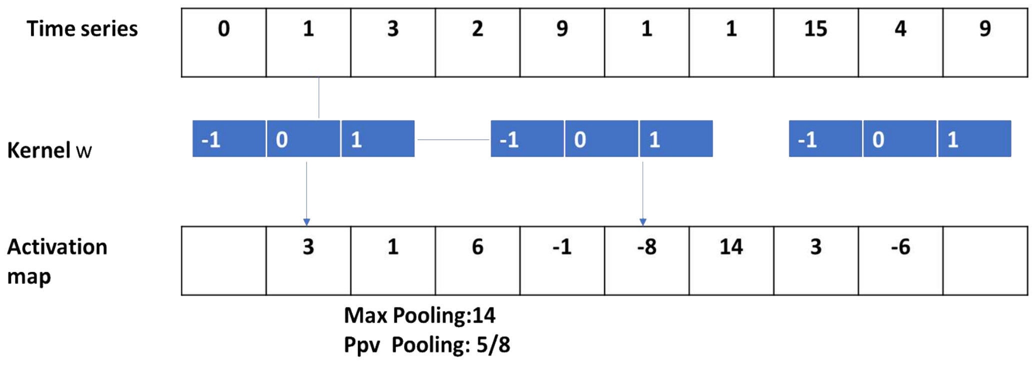


Figure 3. An example of sequence transformation

4.3. Classifier learning

In this step, the feature vectors representing smelly and non-smelly code snippets are used to train a classifier to differentiate between smelly and non-smelly code fragments. Various classifiers have been proposed for this purpose, and this step can be fulfilled using any of these classifiers.

In this paper, we employ the following classifiers: Naïve Bayes, Decision Tree, Logistic Regression, Random Forest, and XGBoost. The trained classifier model is then used to determine whether a new code snippet is smelly or not.

5. Empirical study

This section presents an empirical study on the use of CSDXR for code smell detection. The aim of this experiment is to evaluate the performance of CSDXR in detecting code smells. Specifically, the study addresses the following research objectives and questions:

5.1. Research objectives

The goal of this research is twofold:
1. To explore the feasibility of applying state-of-the-art time series representation methods in the context of code smell detection.
2. To investigate the effectiveness of these representations when used with advanced classifiers.

Based on these goals, the study aims to answer the following research questions:

RQ1: How do variations in the configuration of CSDXR, specifically using MiniRocket and Rocket transformations combined with advanced and standard classifiers, affect the prediction performance in detecting code smells?

We use MiniRocket, Rocket, standard and advanced classifier models in this exploration. MiniRocket and Rocket are fed with time series representing source code snippets. The output is then used with standard and advanced classifiers such as Naïve Bayes, Logistic Regression and XGboost.

Hypothesis 1: It is feasible to detect code smells using classifiers trained on representations pooled by well-configured MiniRocket or its variants from time series representing source codes. The rationale behind this hypothesis is that prior research in time series classification suggests that MiniRocket and Rocket transformations yield distinct feature representations, while classifier choices further modulate performance. Exploring these combinations helps identify optimal configurations for detecting code smells.

RQ2: How does the CSDXR method compare to state-of-the-art baseline tools (DeepSmells, AE-Dense, AE-CNN, and AE-LSTM) in terms of classification metrics such as Precision, Recall, F_1-score, and MCC? Are the performance differences in term of F_1 and MCC statistically significant?

We evaluate how well the CSDXR method performs in comparison to four baseline models presented in Section 4.3.

Hypothesis 2: The CSDXR method can improve the performance of code smell detection. This hypothesis is justified by our thesis that a source code snippet can be viewed as a time series and that Rocket and its variants, including MiniRocket, have proven to be powerful techniques for time series classification. By leveraging these methods, we aim to enhance the effectiveness of detecting code smells.

RQ3: How do the performance (in terms of Precision, Recall, F_1-score, and MCC) and computational cost (transformation time) of CSDXR models with MiniRocket compare to those with Rocket? Are the performance differences in terms of F_1, MCC and transformation time statistically significant?

In this exploration, we replace the MiniRocket-based transformation (CSDXR, corresponding to CSDMR) with the Rocket-based transformation (CSDR). The rationale for this change is supported by existing literature that shows comparable performance of these two transformations in other domains. Since computational cost is a critical factor when selecting the appropriate transformation method in practical applications [19, 20], this question investigates the efficiency of the two designs of the CSDXR method: the

MiniRocket-based CSDMR and the Rocket-based CSDR, with both variants being derived by varying the classifiers used.

Hypothesis 3: We hypothesize that the performance of CSDMR variants (based on MiniRocket) is comparable to CSDR variants (based on Rocket) in the context of code smell detection, with CSDMR variants being faster than CSDR variants. Both variants differ in their classifier selection, which influences their performance and computational efficiency.

This hypothesis is supported by literature showing that MiniRocket and Rocket have comparable performance in other domains and that MiniRocket is faster than Rocket.

To answer RQ1, RQ2, and RQ3, the CSDXR method was trained on a training set and subsequently evaluated on a test dataset. This evaluation used a dataset curated by Sharma et al. [7]. Details of the dataset are provided in the following section.

5.2. Datasets

We conduct our experiments on datasets containing four types of code smells[1]: Complex Method (CM), Complex Conditional (CC), Feature Envy (FE), and Multifaceted Abstraction (MF). Notably, the last two smells, Feature Envy and Multifaceted Abstraction, are particularly challenging to detect [7]. These datasets were curated by Sharma et al. [7] and have been utilized in other studies [10].

The datasets are composed of a total of 416,445 instances, with the following breakdown:

1. **Number of Smelly Instances:** 20,753
2. **Number of Non-Smelly Instances:** 395,692

These datasets are imbalanced, with an average imbalance rate of 4.21%, meaning that on average, positive instances make up around 4.21% of the total instances for each smell. Table 1 presents the Statistics of the Datasets.

Table 1. Statistics of the datasets

Smell	Alias	# Positive	# Negative
Complex Method	CM	12,489	144,460
Complex Conditional	CC	6,186	149,767
Feature Envy	FE	1,788	51,260
Multifaceted Abstraction	MA	290	50,205

5.3. Hardware specification

All experiments are conducted on Google Colab Pro, which provides the necessary computational resources for running the time series transformation and classifier training processes efficiently.

5.4. Evaluation plan

The performance of CSDXR is compared with four baseline models. The baseline models include three variants of an auto-encoder model for code smell detection, introduced by Sharma et al. [7]:

[1] https://github.com/tushartushar/DeepLearningSmells

1. **AE-Dense:** An auto-encoder model using dense layers for both the encoder and decoder.
2. **AE-CNN:** An auto-encoder model employing Convolutional Neural Networks (CNNs) for the encoder and decoder.
3. **AE-LSTM:** An auto-encoder model utilizing Long Short-Term Memory (LSTM) networks.

The fourth baseline model is **DeepSmells**, introduced by Ho et al. [9].

The evaluation process involves comparing the performance metrics of CSDXR with those reported for the baseline models in the study by Ho et al. [9]. This comparison is carried out across four datasets that include Complex Method (CM), Complex Conditional (CC), Feature Envy (FE), and Multifaceted Abstraction (MF).

Each dataset is shuffled and then is split into training and testing subsets with a 70%/30% ratio. The training set is used to train the models, while the testing set is used to evaluate their performance.

The experiments utilize the Sktime library, a Python framework for time series analysis, to implement the time series transformation process required for model training and evaluation.

5.5. Performance

Given the heavy imbalance in code smells datasets [47], using accuracy alone to evaluate classifier performance can lead to misleading results [48]. Therefore, this study uses Precision, Recall, F-measure (F_1), and Matthews Correlation Coefficient (MCC) to assess the performance of the CSDXR method. These metrics are commonly used in code smell detection studies [7, 9, 10, 22, 47] and provide a more reliable evaluation of classifier performance in imbalanced datasets.

– **Precision** measures the proportion of true positive predictions among all positive predictions made by the classifier. It indicates how many of the detected positives are actually true positives.
– **Recall** measures the proportion of true positives that were correctly identified by the classifier out of all actual positives. It reflects the classifier's ability to identify all relevant instances.
– **F-measure** (F_1-score) is the harmonic mean of *precision* and *recall*. It provides a single metric that balances the trade-off between Precision and Recall, making it useful when there is an uneven class distribution.
– **Matthews Correlation Coefficient** (MCC) is a more robust metric compared to accuracy and F-measure. It provides a balanced measure that takes into account all four categories of the confusion matrix: True Positives (TP), False Negatives (FN), False Positives (FP), and True Negatives (TN). The MCC is particularly useful for evaluating performance on imbalanced datasets. It ranges from -1 to $+1$, where $+1$ indicates a perfect prediction, -1 indicates total disagreement, and 0 indicates no better than random prediction.

The confusion matrix is used to calculate these metrics and is summarized as follows:

– **True Positives** (TP): Instances that are actually positive and correctly classified as positive.
– **False Negatives** (FN): Instances that are actually positive but incorrectly classified as negative.
– **False Positives** (FP): Instances that are actually negative but incorrectly classified as positive.

– **True Negatives** (TN): Instances that are actually negative and correctly classified as negative.
These metrics are calculated using the following formulas:

$$precision = \frac{TP}{TP + FP} \tag{2}$$

$$recall = \frac{TP}{TP + FN} \tag{3}$$

F_1 (F-measure): F_1 is the harmonic mean of precision and recall

$$F_1 = \frac{2 \cdot recall \cdot precision}{recall + precision} \tag{4}$$

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP) \cdot (TP + FN) \cdot (TN + FP) \cdot (TN \cdot FN)}} \tag{5}$$

Table 2. Confusion matrix

	Positive (predicted)	Negative (predicted)
True (Actual)	*TP*	*FN*
False (Actual)	*FP*	*TN*

6. Results and discussion

This section presents and discusses the experimental results for the CSDXR method in the context of code smell detection.

6.1. Results for RQ1

RQ1: How do different CSDXR configurations affect the prediction performance?

Approach The CSDXR method leverages either MiniRocket or a similar method such as Rocket for pooling a time series model. These methods consist of two main components:
1. **Pooling a Model**: This involves transforming the time series into a feature vector using MiniRocket or Rocket.
2. **Classification**: This involves using a classifier to predict the class of the time series based on the pooled feature vector.

While the literature typically employs MiniRocket and Rocket with linear classifiers, these methods can theoretically be used with any classifier [19]. Therefore, the CSDXR method's performance depends on various hyperparameters related to both the pooling method and the classifier.

Hyperparameter Tuning Identifying the optimal parameters for the CSDXR approach is a challenging task, as it involves searching across a vast space of possible parameter combinations. This problem, commonly referred to in the literature as hyperparameter tuning, has been extensively studied, with proposed solutions ranging from basic methods, such as grid search, to more advanced metaheuristic-based approaches [49, 50]. In the case of CSDMR, the MiniRocket hyperparameters, such as the number of kernels and dilation, were systematically varied during the training process using a simple grid search approach. The process began by setting dilation to the widely used value of 32, as reported in the literature, and varying the number of kernels across the values {84, 168, 252, 1000, 10000, 10120}. Once the optimal number of kernels was identified, the dilation size was varied across {1, 22, 32, 44} to further refine the hyperparameters for best performance.

In the case of CSDR only the number of kernels was varied, as dilation is randomly set within the Rocket algorithm. The best hyperparameters found during training were then applied in the testing phase to ensure consistency and fairness in evaluation. Classifier hyperparameters were set to their default values as provided by the software packages used.

Design Alternatives To evaluate how different configurations affect the performance of CSDXR, the following design alternatives were studied:
1. **CSDXR with Logistic Regression** (CCDMR_LR) [51]: A linear classifier that models the relationship between features and the target class.
2. **CSDXR with XGBoost** (CCDMR_XGB) [52]: An ensemble method that combines multiple decision trees to improve predictive performance.
3. **CSDXR with Random Forest** (CCDMR_RF) [53]: An ensemble method that aggregates multiple decision trees to enhance robustness and accuracy.
4. **CSDXR with Naïve Bayes** (CCDMR_NB) [54]: A probabilistic classifier based on Bayes' theorem, assuming feature independence. It is important to note that time series data inherently contains correlations between consecutive data points, which violate the assumption of feature independence in models like Naïve Bayes. Despite this, the Naïve Bayes model was selected for its simplicity and efficiency.
5. **CSDXR with Decision Tree** (CCDMR_DT) [55]: A model that splits data based on feature values to make predictions.

The goal was to assess how each classifier, in combination with MiniRocket or Rocket, impacts the overall effectiveness of CSDXR in detecting code smells. The experiments aimed to determine the optimal configuration and hyperparameters for achieving the best performance. The performance of the CSDR design alternatives is detailed in Section 6.3.

Results
1. Effect of the number of kernels on the effectiveness of different design alternatives of the CSDXR model

   Regarding the effect of the number of kernels, Figure 4 shows the F_1-scores of different CSDXR design alternatives as the number of kernels vary, while Figure 5 illustrates the corresponding Matthews Correlation Coefficient (MCC) scores.

   The analysis of these figures reveals that the CSDXR variants achieved their highest performance with 10120 kernels. Specifically, CSDMR_XGB exhibited the highest F_1-score and MCC measures across the CC, CM, and MF datasets. This variant outperformed others with a significant margin, demonstrating its strong capability in detecting code smells effectively.

   In comparison, CSDMR_DT also showed robust performance, particularly on the CC, FE, and MF datasets, although it did not surpass CSDMR_XGB in overall metrics.

   The summary of average performance metrics across the different design alternatives is provided in Table 3. According to this table, CSDMR_XGB leads with an F_1-score of 0.41, followed by CSDMR_DT with 0.35. CSDMR_RF, CSDMR_NB, and CSDMR_-

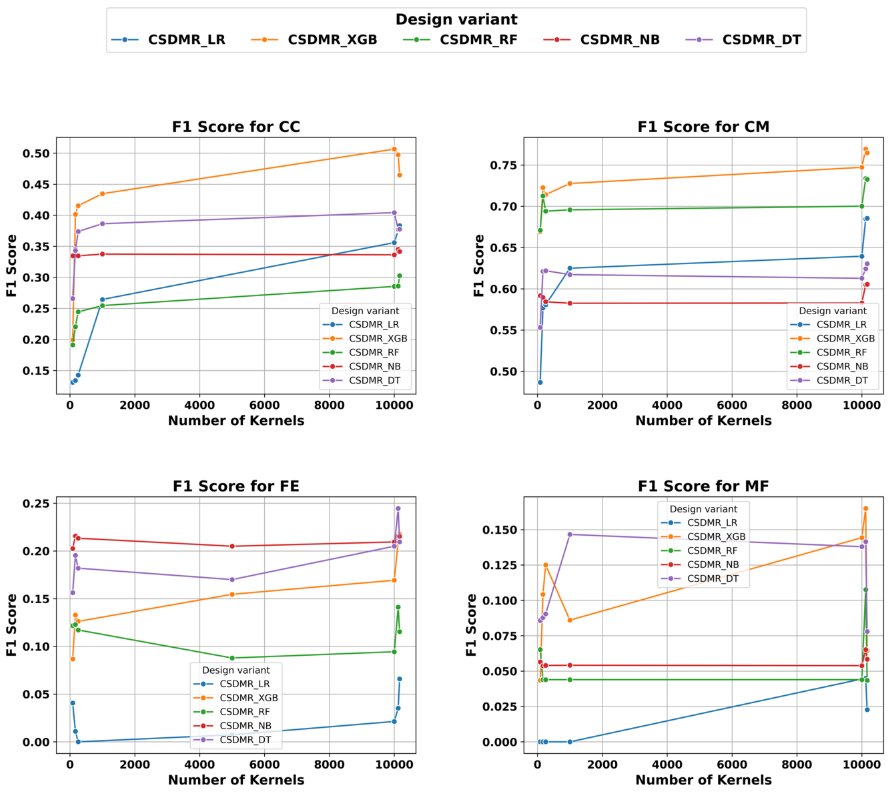


Figure 4. The F_1-score for each CSDMR variant across different smell types

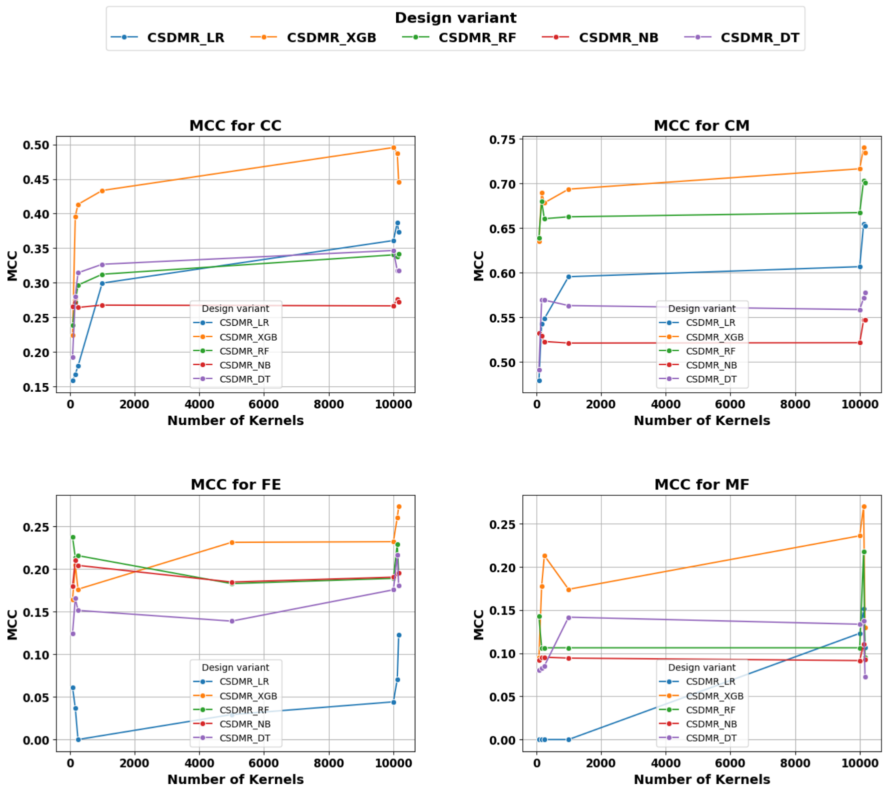


Figure 5. *MCC* score for each CSDMR variant across different smell types

Table 3. Average F_1 and MCC scores for each CSDMR variant across different smell types

Design variant	Smell	F_1-score	MCC
CSDMR_LR	CC	0.38	0.39
	CM	0.68	0.65
	FE	0.04	0.07
	MF	0.04	0.15
average		0.29	0.32
CSDMR_XGB	CC	0.50	0.49
	CM	0.77	0.74
	FE	0.21	0.26
	MF	0.16	0.27
Average		0.41	0.44
CSDMR_RF	CC	0.29	0.34
	CM	0.73	0.70
	FE	0.14	0.23
	MF	0.11	0.22
Average		0.32	0.37
CSDMR_NB	CC	0.35	0.28
	CM	0.60	0.55
	FE	0.22	0.19
	MF	0.07	0.11
Average		0.31	0.28
CSDMR_DT	CC	0.38	0.32
	CM	0.62	0.57
	FE	0.24	0.22
	MF	0.14	0.14
average		0.35	0.31

LR showed lower scores of 0.32, 0.31, and 0.29, respectively. For MCC, CSDMR_XGB achieved a score of 0.44, with CSDMR_RF next at 0.37. CSDMR_LR scored 0.32, CSDMR_DT 0.31, and CSDMR_NB 0.28.

The consistent trend across both F_1-score and MCC metrics indicates that the number of kernels plays a crucial role in the performance of CSDXR. The optimal kernel number of 10120 maximizes the feature representation capability of MiniRocket, thus enhancing the effectiveness of the classifiers. CSDMR_XGB's superior performance highlights its potential for robust code smell detection, while CSDMR_DT also proves to be a strong contender, particularly in certain datasets.

Overall, the results suggest that the choice of kernel number and the specific design variant significantly impact the performance of the CSDXR method. The figures and table provide a clear visualization of these effects, supporting the effectiveness of the CSDMR_XGB design in particular.

2. The effect of dilation

   In this analysis, we examine how different dilation sizes impact the performance of the CSDXR model. The experiment explored four scenarios with dilation sizes set to 1, 22, 32, and 44, while maintaining the number of kernels at 10120, which was previously identified as optimal.

   Figure 6 displays the performance of each CSDMR variant across the four datasets, demonstrating sensitivity to the dilation size. Figure 7 further illustrates the specific F_1 and MCC scores obtained for various dilation sizes.

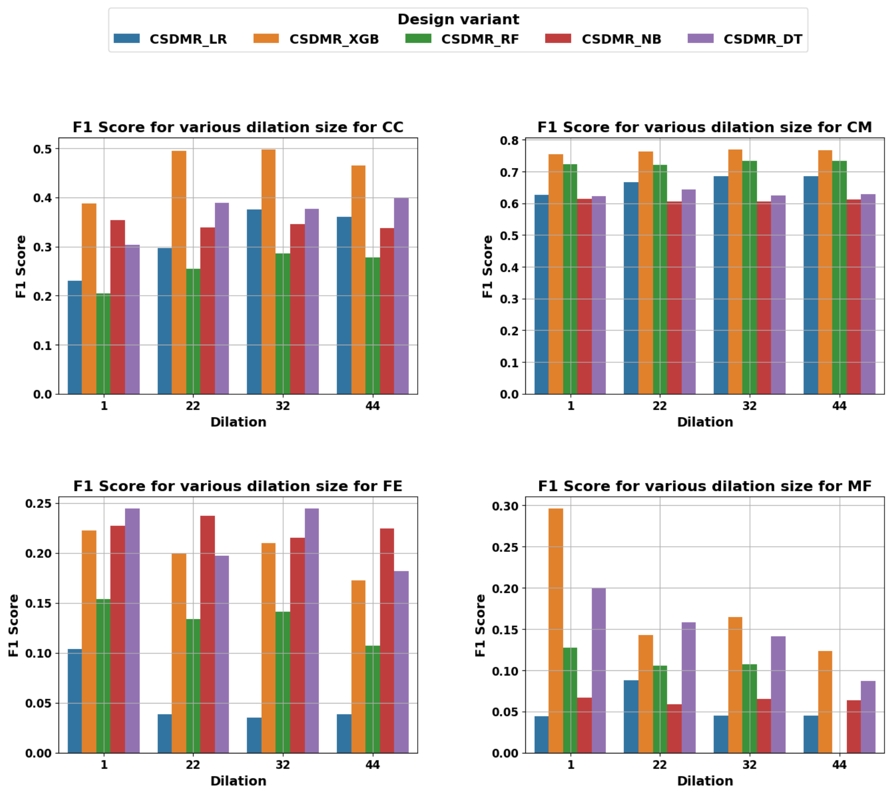


Figure 6. F_1-score by dilation size

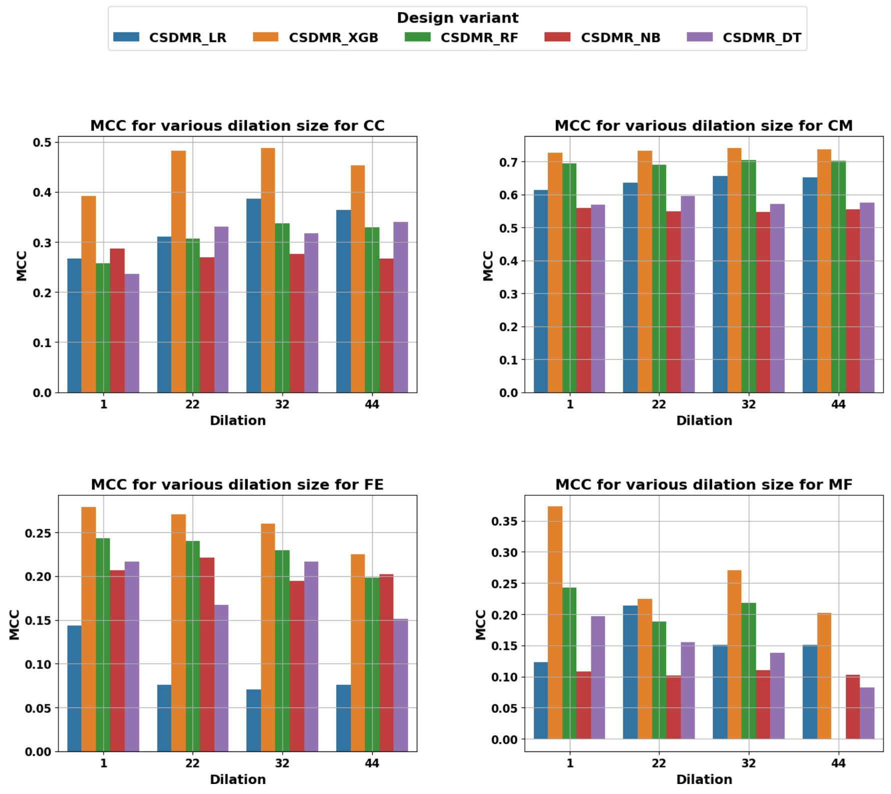


Figure 7. *MCC* score by dilation size

Table 4. Best performance of each CSDMR variant across different smell types by dilation size

Smell	Design variant	Dilation	Precision	Recall	F_1-score	*MCC*
CC	CSDMR_LR	32	0.67	0.26	0.38	0.39
	CSDMR_XGB	32	0.70	0.39	**0.50**	**0.49**
	CSDMR_RF	32	**0.74**	0.18	0.29	0.34
	CSDMR_NB	32	0.30	0.41	0.35	0.28
	CSDMR_DT	44	0.38	**0.42**	0.40	0.34
CM	CSDMR_LR	32	0.32	0.02	0.04	0.07
	CSDMR_XGB	32	**0.81**	**0.73**	**0.77**	**0.74**
	CSDMR_RF	32	0.80	0.68	0.73	0.70
	CSDMR_NB	1	0.59	0.65	0.61	0.56
	CSDMR_DT	22	0.64	0.65	0.64	0.59
FE	CSDMR_LR	1	0.40	0.06	0.10	0.14
	CSDMR_XGB	1	0.61	0.14	0.22	**0.28**
	CSDMR_RF	1	**0.73**	0.09	0.15	0.24
	CSDMR_NB	22	0.17	0.40	**0.24**	0.22
	CSDMR_DT	32	0.23	**0.26**	**0.24**	0.22
MF	CSDMR_LR	22	**1.00**	0.05	0.09	0.21
	CSDMR_XGB	1	0.76	0.18	**0.30**	**0.37**
	CSDMR_RF	1	0.86	0.07	0.13	0.24
	CSDMR_NB	1	0.04	**0.43**	0.07	0.11
	CSDMR_DT	44	0.08	0.10	0.20	0.20

The results reveal that CSDMR_XGB consistently achieved the highest F_1 and *MCC* scores on the CC and CM datasets across all dilation sizes. For the MF dataset, CSDMR_XGB attained the best F_1-scores for dilation sizes of 1, 32, and 44. On the FE dataset, CSDMR_XGB excelled in *MCC* score with a dilation size of 1. However, for the FE dataset's F_1-score, CSDMR_DT and CSDMR_NB were the top performers for dilation sizes of 1 and 32, respectively.

To better understand the optimal dilation size for each variant, Table 4 presents the dilation sizes that achieved the highest F_1 and *MCC* scores for each CSDMR variant. The table indicates that CSDMR_XGB outperforms other variants in terms of F_1-score on most datasets, except for FE, where its F_1-score of 0.22 is lower compared to the 0.24 achieved by CSDMR_DT and CSDMR_NB. Nonetheless, the *MCC* scores for CSDMR_XGB were superior across all datasets, with values of 0.49 for CC, 0.74 for CM, 0.28 for FE, and 0.37 for MF, reflecting a more comprehensive and reliable measure of performance.

The best dilation sizes for each smell type were found to be 32 for CC, 32 for CM, 1 for FE, and 1 for MF. The F_1-scores ranged from 0.22 to 0.77, and the *MCC* values ranged from 0.28 to 0.74.

Overall, the dilation size significantly affects the performance of the CSDXR model, with the optimal size varying depending on the dataset and the specific variant used. The findings suggest that fine-tuning dilation sizes is crucial for achieving the best performance in code smell detection.

Moreover, this table shows that, firstly, for CSDMR_DT, the best performance was observed with dilation sizes of 44, 22, 32, and 44 for CC, CM, FE, and MF smells, respectively. Consequently, the F_1-scores for these settings ranged from 0.20 to 0.64, and the *MCC* values ranged from 0.20 to 0.59.

In contrast, **CSDMR_RF** achieved its best performance with dilation sizes of 32, 32, 1, and 1 for CC, CM, FE, and MF smells. Therefore, the F_1-scores ranged from 0.13 to 0.75, and the *MCC* values varied between 0.24 and 0.70.

Furthermore, **CSDMR_NB** showed optimal performance with dilation sizes of 32, 1, 22, and 1 for CC, CM, FE, and MF smells. In this case, the F_1-scores ranged from 0.07 to 0.61, with *MCC* values between 0.11 and 0.56.

On the other hand, **CSDMR_LR** performed best with dilation sizes of 32, 32, 1, and 22 for CC, CM, FE, and MF smells. The F_1-scores for these sizes ranged from 0.04 to 0.38, and the *MCC* values ranged from 0.07 to 0.39.

Regarding precision, **CSDMR_RF** achieved the highest scores for CC, **CSDMR_-XGB** for CM, **CSDMR_RF** for FE, and **CSDMR_LR** for MF, with precision values of 0.74, 0.81, 0.73, and 1.00, respectively.

In terms of recall, **CSDMR_DT** led for CC, **CSDMR_XGB** for CM, **CSDMR_-DT** for FE, and **CSDMR_NB** for MF, with recall values of 0.42, 0.73, 0.26, and 0.43, respectively.

In summary, the results underscore that dilation size significantly impacts model performance. Different variants exhibit varying sensitivities to dilation size, thus highlighting the need for careful tuning to optimize performance for specific code smells and variant configurations.

RQ1. Hypothesis 1: It is feasible to detect code smells using classifiers trained on representations pooled by well-configured MiniRocket or its variants from time series representing source codes.

It is evident from the study that the CSDMR variants can achieve F_1 and *MCC* scores of 0.74 or higher on certain datasets, demonstrating the feasibility of detecting code smells using classifiers trained on well-configured MiniRocket representations of time series from source code. However, the performance of these classifiers is not uniform across all datasets; for some datasets, the F_1-score may be as low as 0.3. This variability highlights the sensitivity of performance to the specific type of code smell being detected, the characteristics of the dataset, and the configuration of hyperparameters. These findings emphasize the importance of careful dataset selection and hyperparameter tuning in achieving optimal results.

6.2. Results of RQ2

RQ2: How efficient is the CSDXR method?

Approach The performance of CSDMR (Code Smell Detection using MiniRocket) is compared with four baseline models. The baseline models include three variants of an auto-encoder model for code smell detection, introduced by Sharma et al. [7]:

- **AE-Dense**: An auto-encoder model using dense layers for both the encoder and decoder.
- **AE-CNN**: An auto-encoder model employing Convolutional Neural Networks (CNNs) for the encoder and decoder.
- **AE-LSTM**: An auto-encoder model utilizing Long Short-Term Memory (LSTM) networks.

The fourth baseline model is **DeepSmells**, introduced by Ho et al. [9].

Results Regarding the performance of the baseline models, Table 5 presents the results for each type of smell. The data indicate that the CSDMR_XGB model surpasses the

Table 5. Performance of baseline models across different smell types

Smell	Model	Metric			
		P	R	F_1	*MCC*
CM	AE-Dense	0.483	0.630	0.547	0.508
	AE-CNN	0.472	0.582	0.521	0.478
	AE-LSTM	0.468	0.615	0.532	0.491
	DeepSmells	0.731	**0.779**	0.754	0.734
	CSDMR_LR	0.323	0.019	0.035	0.071
	CSDMR_XGB	**0.811**	0.731	**0.769**	**0.740**
	CSDMR_RF	0.802	0.676	0.734	0.703
	CSDMR_NB	0.585	0.648	0.615	0.559
	CSDMR_DT	0.643	0.645	0.644	0.594
CC	AE-Dense	0.170	0.387	0.237	0.211
	AE-CNN	0.194	0.276	0.228	0.193
	AE-LSTM	0.180	0.329	0.232	0.201
	DeepSmells	0.575	**0.604**	**0.589**	**0.568**
	CSDMR_LR	0.667	0.262	0.376	0.387
	CSDMR_XGB	0.697	0.387	0.497	0.487
	CSDMR_RF	**0.737**	0.177	0.286	0.337
	CSDMR_NB	0.298	0.410	0.345	0.277
	CSDMR_DT	0.382	0.420	0.400	0.340
FE	AE-Dense	0.170	0.387	0.237	0.211
	AE-CNN	0.157	**0.493**	0.238	0.235
	AE-LSTM	0.197	0.254	0.222	0.197
	DeepSmells	0.341	0.258	**0.294**	0.269
	CSDMR_LR	0.395	0.060	0.104	0.143
	CSDMR_XGB	0.613	0.136	0.223	**0.279**
	CSDMR_RF	**0.730**	0.086	0.154	0.243
	CSDMR_NB	0.168	0.405	0.237	0.221
	CSDMR_DT	0.230	0.261	0.245	0.217
MA	AE-Dense	0.031	**0.747**	0.060	0.135
	AE-CNN	0.031	0.678	0.060	0.127
	AE-LSTM	0.033	0.402	0.061	0.099
	DeepSmells	0.287	0.272	0.279	0.275
	CSDMR_LR	**1.000**	0.046	0.088	0.214
	CSDMR_XGB	0.762	0.184	**0.296**	**0.373**
	CSDMR_RF	0.857	0.069	0.128	0.242
	CSDMR_NB	0.036	0.425	0.067	0.109
	CSDMR_DT	0.076	0.103	0.199	0.197

other models in terms of both F_1 and *MCC* evaluation metrics for the CM and MF smells, achieving F_1-scores of 0.769 and 0.296, and *MCC* scores of 0.740 and 0.373, respectively. Specifically, the enhancement rates in terms of F_1-score were 1.99% and 6.09% for Complex Method and Multifaceted Abstraction smells, respectively. In terms of *MCC*, the improvement rates were 0.82% and 35.64% for these two smells, respectively. The CSDMR_XGB model outperformed all baseline models in term of *MCC* on Feature Envy dataset. The obtained score was 0.279.

Additionally, the CSDMR_XGB model excels in Precision for the CM and CC smells, with Precision values of 0.811 and 0.697, respectively. The CSDMR_LR model achieves a Precision score of 1.00 for the MF smell and also exhibits superior Precision on the CC smell with a value of 0.737 compared to all other models.

In terms of Recall, the AE-CNN model stands out, achieving the highest Recall scores for the FE and MF smells, with values of 0.493 and 0.747, respectively.

Table 5 also shows that DeepSmells outperforms all other models in terms of F_1 and MCC scores for the CC and FE smells. Although DeepSmells excels on these datasets, the CSDMR_XGBoost model surpasses AE-Dense, AE-CNN, and AE-LSTM on the CC dataset, achieving the best performance compared to AE-LSTM across all datasets. For the FE smell, CSDMR_DT outperforms AE-Dense, AE-CNN, and AE-LSTM. The performance of CSDMR_NB is comparable to that of AE-Dense.

Table 6. Mean F_1-scores and MCC values of CSDMR and baseline models

Model	Mean F_1	Mean MCC
CSDMR_LR	0.15	0.20
CSDMR_XGB	0.45	0.47
CSDMR_RF	0.33	0.38
CSDMR_NB	0.32	0.29
CSDMR_DT	0.37	0.34
AE-Dense	0.27	0.27
AE-CNN	0.26	0.26
AE-LSTM	0.26	0.25
DeepSmells	0.48	0.46

In terms of mean F_1 and mean MCC scores, the results in Table 6 demonstrate that the CSDMR_XGB model outperforms AE-Dense, AE-CNN, and AE-LSTM and achieves performance comparable to DeepSmells. In terms of mean F_1 and mean MCC scores, the results in Table 6 demonstrate that the CSDMR_XGB model outperforms AE-Dense, AE-CNN, and AE-LSTM and achieves performance comparable to DeepSmells. To validate this hypothesis and after confirming the normality of the data, we conducted a Student's t-test (t-test) [56] to detect whether performance differences between CSDMR variants and baseline models are statistically significant. The test used significance rate α equals to 0.05. In hypothesis testing, α denotes the probability of making a Type I error (falsely rejecting the null hypothesis, H_0). An α set to 0.05 means there is only a 5% probability of concluding that an effect exists when it does not. A Result is considered statistically significant when the obtained p-value from the Student's t-test is less than the alpha ($p < \alpha$).However, statistical significance alone is insufficient because p-values do not show the magnitude of the observed effect. Therefore, In addition to significance, we also assessed practical significance by reporting and interpreting effect sizes, which quantify performance differences between models. The magnitude of the difference is quantified using Hedges' g [57] with a 95% confidence interval (CI) and in terms of both the obtained F_1 and MCC scores. Hedges' g was chosen over Cohen's d due to our small sample size. The performance difference is quantified in terms of both the obtained F_1 and MCC scores. The effect sizes were interpreted using Cohen's d guidelines [58]:

- Negligible effect: < 0.2
- Small effect $= 0.2$
- Medium effect $= 0.5$
- Large effect $= 0.8$

Table 7 Shows p-values for F_1 and MCC scores when comparing CSDMR variants with baseline models, along with effect sizes for significant results (p-value < 0.05)and power values for non-significant results. Meanwhile, Table 6 shows Mean F_1-scores and MCC values of CSDMR and baseline models. These tables show that, in term of MCC:

1. CSDMR_XGB significantly outperformed AE_Dense, AE_CNN, and AE_LSTM with a large effect size.

Table 7. Comparison of models with statistical tests: p-value of the statistical test along with effect size (Hedges' g) in case of a significant test, and power values in case of non-significant test. S? indicates whether the result is significant (Yes) or not (No). Negative effect size indicates a performance superiority for the baseline model

		MCC				F_1			
Model 1	Model 2	p-value	Effect size	Power	S?	p-value	Effect size	Power	S?
CSDMR_LR	AE-Dense	0.67		0.08	No	0.46		0.13	No
CSDMR_LR	AE-CNN	0.71		0.07	No	0.48		0.12	No
CSDMR_LR	AE-LSTM	0.77		0.06	No	0.48		0.12	No
CSDMR_LR	DeepSmells	0.16		0.37	No	0.09		0.51	No
CSDMR_XGB	AE-Dense	0.02	0.97		Yes	0.07		0.16	No
CSDMR_XGB	AE-CNN	0.03	1.04		Yes	0.07		0.17	No
CSDMR_XGB	AE-LSTM	0.02	1.05		Yes	0.06		0.17	No
CSDMR_XGB	DeepSmells	0.84		0.05	No	0.33		0.05	No
CSDMR_RF	AE-Dense	0.04	0.52		Yes	0.39		0.06	No
CSDMR_RF	AE-CNN	0.07		0.12	No	0.37		0.06	No
CSDMR_RF	AE-LSTM	0.03	0.60		Yes	0.33		0.06	No
CSDMR_RF	DeepSmells	0.21		0.07	No	0.08		0.11	No
CSDMR_NB	AE-Dense	0.31		0.05	No	0.17		0.06	No
CSDMR_NB	AE-CNN	0.33		0.06	No	0.17		0.06	No
CSDMR_NB	AE-LSTM	0.07		0.06	No	0.13		0.06	No
CSDMR_NB	DeepSmells	0.04	−0.70		Yes	0.03	−0.61		Yes
CSDMR_DT	AE-Dense	0.07		0.08	No	0.06		0.09	No
CSDMR_DT	AE-CNN	0.12		0.09	No	0.05		0.10	No
CSDMR_DT	AE-LSTM	0.04	0.44		Yes	0.04	0.48		Yes
CSDMR_DT	DeepSmells	0.05		0.11	No	0.04	-0.43		Yes

2. CSDMR_RF significantly outperformed AE_Dense and AE_LSTM with a medium effect size.
3. DeepSmells significantly outperformed CSDMR_NB with a medium effect size.
4. CSDMR_DT significantly outperformed AE_LSTM with a small effect size.

These tables show also that in term of F_1:

1. DeepSmells significantly outperformed CSDMR_NB with a medium effect size.
2. AE_LSTM significantly outperformed CSDMR_NB with a small effect size.
3. DeepSmells significantly outperformed CSDMR_DT with a small effect size.

Additionally, a post-hoc power analysis (i.e., retrospective power analysis) is conducted for cases where statistically nonsignificant results were obtained to assess the reliability of these findings. It is important to mention that while researchers agree on the importance of prospective power analysis to determine an adequate sample size for a planned research study [59, 60], they disagree about the value of a post hoc power analysis. Some researchers still recommend that power analysis can be done retrospectively, especially when a statistically nonsignificant result is obtained [59,60]. In this case, a post hoc power analysis is conducted to determine if the lack of significance is due to low statistical power or to a truly small effect. Power analysis is based on four related parameters (the sample size, the effect size, the significance level (α, often set to 0.05), and the statistical power. A power analysis is generally conducted to estimate one of these four parameters given the remaining three values. The statistical power of a test is the probability of rejecting H_0 when it is really false (i.e., the capacity to detect an effect if it is really there). This power is tied by an inverse relation with β (i.e., the probability of making a Type II error) and is equal to $1 - \beta$. A low power value indicates that there is a high risk of Type II errors, while a high value indicates a low risk of Type II errors. The literature shows that 0.20 is the acceptable level of β, so the desired power is 0.80. In our study, the estimated post hoc power was found to be low and range between 0.5 and 0.05 in each nonsignificant case, which leads to the conclusion that the non-significance is due to low power and suggests that more powerful research should be conducted.

RQ2. Hypothesis 2: The CSDXR method can improve the performance of code smell detection.

Overall, the results indicate that the CSDMR_XGB model achieves superior F_1 and MCC scores compared to the baseline models on two types of smells. Specifically, the enhancement rates in terms of F_1-score were 1.99% and 6.09% for Complex Method and Multifaceted Abstraction smells, respectively. In terms of MCC, the improvement rates were 0.82% and 35.64% for these two smells, respectively. Additionally, it demonstrates enhanced Precision for one specific type of smell. The CSDMR_LR model also excels in Precision for two types of smells. For each type of smell, at least one CSDMR variant surpasses the performance of the AE-Dense, AE-CNN, and AE-LSTM models.
In general, The Student's test indicate that CSDMR outperformed AE-Dense, AE-CNN, and AE-LSTM models while achieving performance comparable to DeepSmells.
Moreover, the use of a simple grid search strategy for hyperparameter tuning provides an initial baseline for the performance of the CSDXR approach. Therefore, we accept the hypothesis that CSDXR models can improve the performance of code smell detection. However, this strategy may not fully leverage the model's potential. Incorporating more advanced hyperparameter tuning methods, such as evolutionary algorithms, could lead to improved performance. Future work will be dedicated to the exploration of these advanced strategies to reveal the potential of CSDXR. Finally, it is also important to note that the statistical power of the analysis suggests that future studies need to be carried out to validate the reliability of nonsignificant.

6.3. Results of RQ3

RQ3: How does the CSDMR's performance and computational cost compare to that of CSDR?

Approach The CSDXR method, which consists of transformation and classification components, is compared with the CSDR method. For this comparison, we implemented the transformation component using the Rocket technique (CSDR) and assessed the performance of CSDR variants against CSDMR variants in terms of F_1-score and MCC. Additionally, we compared the transformation times logged for both CSDMR and CSDR methods to convert the training dataset.

Results Table 10 presents the performance metrics for each design variant of CSDR and CSDMR, with configurations that produce feature vectors of size S for each type of smell(O.size). The table includes performance results in terms of F_1-score, MCC, and transformation time (Trans. Times) in seconds. It also highlights the differences in F_1 and MCC scores between each CSDR variant and its corresponding CSDMR variant. The final column shows the Transformation Time Ratio (TTR) CSDMR relative to CSDR for each smell.

The results indicate that the performance of CSDMR and CSDR variants is comparable, with differences in F_1 and *MCC* scores ranging from 0 to 0.16. In all cases, there were no significant performance differences between CSDR and CSDMR. There were no performance differences between CSDR and CSDMR in seven cases. CSDMR performed worse than CSDR in 17 cases, while it outperformed CSDR in 16 cases. Regarding transformation times, CSDMR is significantly faster than CSDR. Specifically, CSDMR demonstrated a speed rate approximately 16 times faster on the CC dataset with an output vector size of 84. The speed rate varied between 2 and 16 across different datasets, with an average speed rate of 12.7.

A statistical test was conducted to examine whether there are significant differences between the CSDMR and CSDR results, specifically in terms of F_1-scores, *MCC* scores, and transformation times.

The Kolmogorov-Smirnov test [61] was chosen for this analysis due to the non-normal distribution of the data. A p-value less than 0.05 indicates the presence of significant differences between the CSDMR and CSDR approaches. Table 8 presents the p-values for CSDMR and CSDR in terms of F_1 and *MCC* scores, while Table 10 presents the mean execution times for CSDMR and CSDR, along with the p-values for transformation times.

Table 9 shows that all the obtained p-values for F_1 and *MCC* scores are greater than 0.05, indicating that there are no significant differences between CSDMR and CSDR in these metrics. However, the p-value for transformation time is less than 0.05 (see Table 10), suggesting a significant difference between CSDMR and CSDR in terms of transformation time. Based on the mean execution times of CSDMR and CSDR and the p-value presented in this table, we can conclude that CSDMR is faster than CSDR in terms of transformation time.

This analysis shows that while the performance of CSDMR and CSDR is generally similar, CSDMR offers a considerable advantage in terms of transformation time, making it a more efficient choice for code smell detection.

RQ3. Hypothesis 3: CSDMR variants' performance is comparable to CSDR variants in the context of code smell detection, while CSDMR variants are faster than CSDR variants.

Therefore, we conclude that the performance of CSDMR variants is comparable to that of CSDR variants. Additionally, CSDMR variants are significantly faster than CSDR variants.

6.4. Discussion

This study demonstrated that the proposed method, CSDXR, have the potential to detect smells without the use an extensive feature engineering process. The study revealed also that MiniRocket combined with the XGBoost classifier outperforms other variants in terms of detection performance.

However, the obtained results demonstrate that the performance of the CSDXR method is highly sensitive to the type of code smell. While slight improvements were observed for two code smells (CM and MA smells), the results indicate that there is still room for significant enhancement, as the highest F_1 and *MCC* scores achieved were 0.769 and 0.740 respectively.

Table 8. Performance metrics for each design variant of CSDR and CSDMR

			CSDR			CSDMR					
Smell	O. size	Model variant	F_1	*MCC*	Trans. Time(s)	F_1	*MCC*	Trans. Time (s)	F_1 diff.	*MCC* diff.	TTR
	82	CSDXR_LR	0.00	0.00	27.82	**0.13**	**0.16**	1.74	-0.13	-0.16	**16.01**
	82	CSDXR_XGB	**0.30**	**0.29**		0.20	0.22		0.10	0.07	
CC	82	CSDXR_RF	**0.21**	**0.27**		0.19	0.24		0.02	0.03	
	82	CSDXR_NB	**0.34**	**0.28**		0.33	0.27		0.01	0.02	
	82	CSDXR_DT	**0.31**	**0.25**		0.27	0.19		0.05	0.05	
	82	CSDXR_LR	**0.60**	**0.55**	27.78	0.49	0.48	2.31	0.11	0.07	12.03
	82	CSDXR_XGB	**0.69**	**0.65**		0.67	0.63		0.02	0.01	
CM	82	CSDXR_RF	**0.69**	**0.65**		0.67	0.64		0.02	0.01	
	82	CSDXR_NB	**0.60**	**0.54**		0.59	0.53		0.01	0.01	
	82	CSDXR_DT	**0.57**	**0.51**		0.55	0.49		0.02	0.02	
	82	CSDXR_LR	0.00	0.00	198.58	**0.04**	**0.06**	14.71	-0.04	-0.06	13.50
	82	CSDXR_XGB	0.08	0.14		**0.09**	**0.16**		0.00	-0.02	
FE	82	CSDXR_RF	0.07	0.16		**0.12**	**0.24**		-0.05	-0.07	
	82	CSDXR_NB	**0.21**	**0.22**		0.20	0.18		0.01	0.04	
	82	CSDXR_DT	0.14	0.11		**0.16**	**0.12**		-0.01	-0.01	
	82	CSDXR_LR	0.00	0.00	218.23	0.00	0.00	16.27	0.00	0.00	13.42
	82	CSDXR_XGB	0.02	0.06		**0.04**	**0.09**		-0.02	-0.03	
MF	82	CSDXR_RF	0.02	0.11		**0.07**	**0.14**		-0.04	-0.04	
	82	CSDXR_NB	0.05	**0.14**		**0.06**	0.09		0.00	0.05	
	82	CSDXR_DT	0.15	0.14		0.09	0.08		0.06	0.06	
	1000	CSDXR_LR	0.10	0.13	261.60	**0.26**	**0.30**	17.07	-0.16	-0.17	15.32
	1000	CSDXR_XGB	0.42	0.40		**0.43**	**0.43**		-0.02	-0.03	
CC	1000	CSDXR_RF	**0.29**	**0.34**		0.25	0.31		0.04	0.02	
	1000	CSDXR_NB	**0.34**	**0.27**		**0.34**	**0.27**		0.00	0.00	
	1000	CSDXR_DT	0.36	0.30		0.39	0.33		-0.03	-0.03	
	1000	CSDXR_LR	**0.64**	**0.60**	286.03	0.62	**0.60**	20.68	0.01	0.00	13.83
	1000	CSDXR_XGB	0.72	**0.69**		**0.73**	**0.69**		0.00	0.00	
CM	1000	CSDXR_RF	**0.72**	**0.68**		0.70	0.66		0.02	0.02	
	1000	CSDXR_NB	**0.59**	**0.53**		0.58	0.52		0.00	0.01	
	1000	CSDXR_DT	0.60	0.55		0.62	0.56		-0.01	-0.02	
	1000	CSDXR_LR	**0.09**	**0.20**	1937.35	0.02	0.04	790.66	0.07	0.15	2.45
	1000	CSDXR_XGB	0.16	0.22		**0.17**	**0.23**		-0.01	-0.01	
FE	1000	CSDXR_RF	**0.13**	**0.23**		0.09	0.19		0.03	0.04	
	1000	CSDXR_NB	0.19	**0.21**		**0.21**	0.19		-0.02	0.02	
	1000	CSDXR_DT	0.17	0.14		0.20	0.18		-0.03	-0.03	
	1000	CSDXR_LR	**0.15**	**0.19**	2103.54	0.00	0.00	169.46	0.15	0.19	12.41
	1000	CSDXR_XGB	0.04	0.09		**0.09**	**0.17**		-0.04	-0.08	
MF	1000	CSDXR_RF	0.02	0.06		**0.04**	**0.11**		-0.02	-0.05	
	1000	CSDXR_NB	0.05	0.13		0.05	0.09		0.00	0.04	
	1000	CSDXR_DT	0.08	0.07		0.15	0.14		-0.07	-0.07	

Table 9. *p*-values for F_1 and *MCC* scores comparing CSDMR and CSDR

		CSDR			
	Smell	CC	CM	FE	MF
CSDMR	F_1	0.87	0.87	0.87	0.87
	MCC	0.5	0.87	0.99	0.99

Table 10. Mean execution time and p-values for transformation times comparing CSDMR and CSDR

Approach	Mean execution time (s)
CSDMR	129.11
CSDR	632.62
p-value	0.019

While, the CSDXR low performance obtained can be attributed to the imbalanced nature of the dataset used, which makes the detection process more challenging. In general, better performance could likely be achieved with more balanced datasets.

Additionally, three other primary reasons can explain these performance differences:

1. **Nature of the Code Smell**: The inherent characteristics of different code smells play a significant role. Some code smells exhibit identifiable patterns in their structure, while others do not, making them harder to detect.
2. **Nature of the CSDXR Approach**: The performance is influenced by the ability of the kernels and/or pooling operators used in the Rocket and the MiniRocket method to detect and summarize patterns present in the source code. This highlights the importance of kernel design in identifying meaningful patterns.
3. **Source Code Transformation**: The transformation approach used to convert source code into time series data may fail to adequately reveal the patterns present in the source code. This can impact the ability of the method to effectively detect certain code smells.

For instance, the superior results on the CM and MA datasets may be attributed to the presence of well-defined patterns in the smelly source code for these datasets. In contrast, the other datasets may lack such patterns, resulting in lower performance, or the patterns that identify the smell may be present, but the approach lacks the capability to detect them. Thus, our conjecture is that the variation in CSDXR performance across different types of smells can be explained by the fact that the CSDXR approach is fundamentally a token-based method. Code smells can be detected by leveraging various types of information present in a code fragment—such as lexical, structural, semantic, or contextual information. While some smells can be identified using just one of these types, others require a combination, making the detection process more complex. An effective detection method should ideally incorporate all of them. In our case, the CSDXR method relies primarily on lexical features and only a limited amount of structural information (i.e., tokens and their order of appearance), which may limit its effectiveness for certain smells. For example, detecting the Feature Envy smell also requires semantic information—like the meaning of identifiers, data and control dependencies, comments, and so on.

To clearly identify the reasons behind these results, it is crucial to design new experiments. These should investigate the potential of novel kernel designs, improved pooling operators, and alternative source code transformation approaches to better capture the underlying patterns in source code.

Additionally, while this study primarily focused on evaluating efficiency in terms of transformation time, we recognize that cost associated with memory usage is another critical parameter that significantly influences scalability and practical applicability. Addressing cost related to memory usage will be essential for enhancing the approach's performance in real-world scenarios.

7. Threats to validity

7.1. Internal validity

The internal validity of the study may be affected by several factors. First, the use of the Sktime library for implementing the time series transformation module and setting classifier hyperparameters to package default values may introduce biases or limit the method's performance. Second, while the number of kernels and dilation parameters were varied until satisfactory results were achieved, this approach may not fully capture the robustness of the findings. Additionally, the absence of cross-validation could impact the reliability of the reported results. To address these limitations, future work should include a more comprehensive experimental design involving extensive hyperparameter tuning and the use of cross-validation to ensure a clearer understanding of the method's capabilities and more reliable conclusions.

7.2. External validity

External validity concerns the generalizability of the study's findings. This study evaluated the CSDXR approach on only four types of code smells, each with distinct patterns. However, code smells vary in their properties, and the CSDXR approach shows promise in detecting smells that exhibit clear patterns. It may not, however, be as effective for detecting smells with subtler or less distinct patterns. To enhance the generalizability of the results, it would be beneficial to test the CSDXR approach on a broader range of code smells, including those with less obvious patterns. This would help determine whether the approach can be applied effectively in different contexts and scenarios, ultimately assessing its broader applicability in code smell detection.

8. Conclusion

This paper introduces a novel approach for code smell detection using advanced time series classification techniques such as Rocket and MiniRocket. The proposed CSDXR method involves converting a code source snippet into a sequence of numbers through tokenization, generating a vectorial representation using a random convolutional transform method, and then training a classifier on these vector representations, labeled as smelly or non-smelly. This classifier is subsequently used to determine if new code snippets are smelly based on their representations.

The empirical study, conducted on a well-known dataset to detect four code smells, shows that the CSDXR approach outperforms four state-of-the-art methods, particularly in detecting Complex Method and Multi-Faceted Smells. Although the DeepSmells method performs better than CSDXR, the CSDXR approach surpasses the performance of AE-Dense, AE-CNN, and AE-LSTM models.

Future work will focus on exploring advanced time series representations to further improve code smell detection capabilities.

Acknowledgment

The authors acknowledge the use of language models, including ChatGPT and DeepSeek, which assisted in enhancing the grammatical accuracy and overall clarity of the manuscript during the writing process. All content was thoroughly reviewed, verified, and edited by the authors to ensure it accurately reflects the scientific contributions of this work.

CRediT authorship contribution statement

Dr. Mostefai Abdelkader – conceptualization, methodology, software, data curation, investigation, visualization, writing – original draft, writing – writing – review & editing.
Dr. Mekour Mansour – conceptualization, writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The code smell datasets used in this research is provided on the link: https://zenodo.org/records/15602620.

Funding

No funding was received to assist with the preparation of this manuscript.

References

[1] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, “Managing technical debt in software engineering (Dagstuhl seminar 16162),” *Dagstuhl Reports*, Vol. 6, No. 4, 2016.
[2] P. Kruchten, R.L. Nord, and I. Ozkaya, “Technical debt: From metaphor to theory and practice,” *IEEE Software*, Vol. 29, No. 6, 2012, pp. 18–21.
[3] A. Alazba, H. Aljamaan, and M. Alshayeb, “Deep learning approaches for bad smell detection: A systematic literature review,” *Empirical Software Engineering*, Vol. 28, No. 77, 2023.
[4] T. Sharma and D. Spinellis, “A survey on software smells,” *Journal of Systems and Software*, Vol. 138, 2018, pp. 158–173.
[5] N. Sae-Lim, S. Hayashi, and M. Saeki, “How do developers select and prioritize code smells? a preliminary study,” in *Proceedings IEEE 33rd International Conference Software Maintenance and Evolution (ICSME)*, 2017, pp. 484–488.
[6] M. Hozano, A. Garcia, B. Fonseca, and E. Costa, “Are you smelling it? Investigating how similar developers detect code smells,” *Information and Software Technology*, Vol. 93, 2018, pp. 130–146.
[7] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, “Code smell detection by deep direct-learning and transfer-learning,” *Journal of Systems and Software*, Vol. 176, 2021, p. 110936.

[8] M.I. Azeem, F. Palomba, L. Shi, and Q. Wang, “Machine learning techniques for code smell detection: A systematic literature review and meta-analysis,” *Information and Software Technology*, Vol. 108, 2019, pp. 115–138.

[9] A. Ho, A.M. Bui, P.T. Nguyen, and A.D. Salle, “Fusion of deep convolutional and LSTM recurrent neural networks for automated detection of code smells,” in *Proceedings 27th International Conference Evaluation and Assessment in Software Engineering*, 2023, pp. 229–234.

[10] W. Xu and X. Zhang, “Multi-granularity code smell detection using deep learning method based on abstract syntax tree,” in *Proceedings International Conference Software Engineering and Knowledge Engineering*, 2021.

[11] A. Bagnall, J. Lines, A. Bostrom, J. Large, and E. Keogh, “The great time series classification bake off: A review and experimental evaluation of recent algorithmic advances,” *Data Mining and Knowledge Discovery*, Vol. 31, No. 3, 2017, pp. 606–660.

[12] M. Middlehurst, P. Schäfer, and A. Bagnall, “Bake off redux: A review and experimental evaluation of recent time series classification algorithms,” *Data Mining and Knowledge Discovery*, 2024, pp. 1–74.

[13] J. Zhang, F.Y. Wang, K. Wang, W.H. Lin, X. Xu et al., “Data-driven intelligent transportation systems: A survey,” *IEEE Transactions on Intelligent Transportation Systems*, Vol. 12, No. 4, 2011, pp. 1624–1639.

[14] Y. Zhou, Z. Ding, Q. Wen, and Y. Wang, “Robust load forecasting towards adversarial attacks via Bayesian learning,” *IEEE Transactions on Power Systems*, Vol. 38, No. 2, 2023, pp. 1445–1459.

[15] A.A. Cook, G. Mısırlı, and Z. Fan, “Anomaly detection for IoT time-series data: A survey,” *IEEE Internet of Things Journal*, Vol. 7, No. 7, 2019, pp. 6481–6494.

[16] M. Abdelkader, “A novel method for code clone detection based on minimally random kernel convolutional transform,” *IEEE Access*, Vol. 12, 2024, pp. 158 579–158 596.

[17] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1999.

[18] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt.* Morgan Kaufmann, 2014.

[19] A. Dempster, F. Petitjean, and G.I. Webb, “ROCKET: Exceptionally fast and accurate time series classification using random convolutional kernels,” *Data Mining and Knowledge Discovery*, Vol. 34, No. 5, 2020.

[20] A. Dempster, D.F. Schmidt, and G.I. Webb, “MiniRocket: A very fast (almost) deterministic transform for time series classification,” in *Proceedings International Conference Knowledge Discovery and Data Mining*, 2021, pp. 248–257.

[21] B. Nyirongo, Y. Jiang, H. Jiang, and H. Liu, “A survey of deep learning based software refactoring,” *arXiv preprint arXiv:2404.19226*, 2024.

[22] T. Sharma and M. Kessentini, “QScored: a large dataset of code smells and quality metrics,” in *Proceedings IEEE/ACM 18th International Conference Mining Software Repositories (MSR)*, 2021, pp. 590–594.

[23] Y. Zhang, C. Ge, H. Liu, and K. Zheng, “Code smell detection based on supervised learning models: A survey,” *Neurocomputing*, Vol. 565, 2024, p. 127014.

[24] R.S. Pressman, *Software Engineering: A Practitioner’s Approach*, 6th ed. Palgrave Macmillan, 2005.

[25] N.E. Fenton, *Software Metrics – A Rigorous Approach.* London: Chapman and Hall, 1991.

[26] R. Marinescu, “Detection strategies: Metrics-based rules for detecting design flaws,” in *Proceedings 20th International Conference Software Maintenance (ICSM)*, Chicago, IL, USA, 2004, pp. 350–359.

[27] R. Marinescu, “Measurement and quality in object-oriented design,” in *Proceedings 21st IEEE International Conference Software Maintenance (ICSM)*, Budapest, Hungary, 2005, pp. 701–704.

[28] I.M. Bertran, A. Garcia, and A. von Staa, “Defining and applying detection strategies for aspect-oriented code smells,” in *24th Brazilian Symposium on Software Engineering, SBES*, Salvador, Bahia, Brazil, 2010, pp. 60–69.

[29] A.M. Fard and A. Mesbah, “JSNOSE: Detecting JavaScript code smells,” in *Proceedings 13th IEEE International Working Conference Source Code Analysis and Manipulation (SCAM)*, Eindhoven, Netherlands, 2013, pp. 116–125.

[30] Z. Chen, L. Chen, W. Ma, X. Zhou, Y. Zhou et al., “Understanding metric-based detectable smells in Python software: A comparative study,” *Information and Software Technology*, Vol. 94, 2018, pp. 14–29.

[31] N. Moha, Y.G. Guéhéneuc, L. Duchien, and A.F.L. Meur, “DECOR: A method for the specification and detection of code and design smells,” *IEEE Transactions on Software Engineering*, Vol. 36, No. 1, 2010, pp. 20–36.

[32] F.A. Fontana, M.V. Mäntylä, M. Zanoni, and A. Marino, “Comparing and experimenting with machine learning techniques for code smell detection,” *Empirical Software Engineering*, Vol. 21, No. 3, 2016, pp. 1143–1191.

[33] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.G. Guéhéneuc et al., “Support vector machines for anti-pattern detection,” in *Proceedings 27th IEEE/ACM International Conference Automated Software Engineering (ASE)*, 2012, pp. 278–281.

[34] F. Khomh, S. Vaucher, Y.G. Guéhéneuc, and H. Sahraoui, “A Bayesian approach for the detection of code and design smells,” in *Proceedings 9th International Conference Quality Software (QSIC)*, 2009, pp. 305–314.

[35] J. Kreimer, “Adaptive detection of design flaws,” *Electronic Notes in Theoretical Computer Science*, Vol. 141, No. 4, 2005, pp. 117–136.

[36] M. Škipina, J. Slivka, N. Luburić, and A. Kovačević, “Automatic detection of code smells using metrics and codeT5 embeddings: A case study in C,” *Neural Computing and Applications*, 2024, pp. 1–18.

[37] M. Hadj-Kacem and N. Bouassida, “A hybrid approach to detect code smells using deep learning,” in *Proceedings International Conference Evaluation of Novel Approaches to Software Engineering*, 2018.

[38] H. Liu, Z. Xu, and Y. Zou, “Deep learning-based feature envy detection,” in *Proceedings 33rd IEEE/ACM International Conference Automated Software Engineering (ASE)*, 2018, pp. 385–396.

[39] B. Liu, H. Liu, G. Li, N. Niu, Z. Xu et al., “Deep learning-based feature envy detection boosted by real-world examples,” in *Proceedings 31st ACM Joint European Software Engineering Conference and Sympossium Foundations of Software Engineering (ESEC/FSE)*, 2023, pp. 908–920.

[40] A.K. Das, S. Yadav, and S. Dhal, “Detecting code smells using deep learning,” in *Proceedings TENCON – IEEE Region 10 Conference*, 2019, pp. 2081–2086.

[41] D. Yu, Y. Xu, L. Weng, J. Chen, X. Chen et al., “Detecting and refactoring feature envy based on graph neural network,” in *Proceedings IEEE 33rd International Symposium Software Reliability Engineering (ISSRE)*, 2022, pp. 458–469.

[42] H. Zhang and T. Kishi, “Long method detection using graph convolutional networks,” *Journal of Information Processing*, Vol. 31, Aug. 2023, pp. 469–477.

[43] Y. Zhang, C. Ge, S. Hong, R. Tian, C.R. Dong et al., “DeleSmell: Code smell detection based on deep learning and latent semantic analysis,” *Knowledge-Based Systems*, Vol. 255, 2022, p. 109737.

[44] Y. Zhang and C. Dong, “MARS: Detecting brain class/method code smell based on metric – attention mechanism and residual network,” *Journal of Software: Evolution and Process*, 2021.

[45] Y. Li and X. Zhang, “Multi-label code smell detection with hybrid model based on deep learning,” in *Proceedings International Conference Software Engineering and Knowledge Engineering*, 2022.

[46] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu et al., “Deep learning based code smell detection,” *IEEE Transactions on Software Engineering*, Vol. 47, No. 9, 2019, pp. 1811–1837.

[47] K. Alkharabsheh, S. Alawadi, V.R. Kebande, Y. Crespo, M.F. Delgado et al., “A comparison of machine learning algorithms on design smell detection using balanced and imbalanced dataset: A study of god class,” *Information and Software Technology*, Vol. 143, 2022, p. 106736.

[48] P. Probst, B. Bischl, and A.L. Boulesteix, “Tunability: Importance of hyperparameters of machine learning algorithms,” *arXiv preprint arXiv:1802.09596*, 2018.

[49] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, “Hyperband: A novel bandit-based approach to hyperparameter optimization,” in *Proceedings International Conference Learning Representations*, 2018, pp. 1–48.

[50] L. Li, K. Jamieson, A. Rostamizadeh, E. Gonina, J. Ben-Tzur et al., “A system for massively parallel hyperparameter tuning,” in *Proceedings Machine Learning and Systems*, Vol. 2, 2020, pp. 230–246.

[51] S.H. Walker and D.B. Duncan, “Estimation of the probability of an event as a function of several independent variables,” *Biometrika*, Vol. 54, No. 1–2, 1967, pp. 167–179.

[52] T. Chen and C. Guestrin, “XGBoost: A scalable tree boosting system,” in *Proceedings International Conference Knowledge Discovery and Data Mining*, 2016, pp. 785–794.

[53] T.K. Ho, “Random decision forests,” in *Proceedings 3rd International Conference Document Analysis and Recognition*, 1995, pp. 278–282.

[54] J.G.H. John and P. Langley, “Estimating continuous distributions in Bayesian classifiers,” *arXiv preprint arXiv:1302.4964*, 2013.

[55] F. Yang, “An extended idea about decision trees,” in *International Conference on Computational Science and Computational Intelligence (CSCI)*, 2019, pp. 349–354.

[56] Student, “The probable error of a mean,” *Biometrika*, Vol. 6, 1908, pp. 1–25.

[57] L.V. Hedges, “Estimation of effect size from a series of independent experiments,” *Psychological Bulletin*, Vol. 92, No. 2, 1982, pp. 490–499.

[58] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed. Routledge, 1988.

[59] P.D. Ellis, *The essential guide to effect sizes: Statistical power, meta-analysis, and the interpretation of research results.* Cambridge University Press, 2010.

[60] J.M. Hoenig and D.M. Heisey, “The abuse of power: The pervasive fallacy of power calculations for data analysis,” *The American Statistician*, Vol. 55, No. 1, 2001, pp. 1–6.

[61] F.J. Massey, “The Kolmogorov–Smirnov test for goodness of fit,” *Journal of the American Statistical Association*, Vol. 46, No. 253, 1951, pp. 68–78.

Authors and affiliations

Mostefai Abdelkader
e-mail: abdelkader.mostefai@univ-saida.dz
ORCID: https://orcid.org/0000-0003-0408-331X
Department of Computer Science,
University of Saida, Dr. Taher Moulay, Algeria

Mekour Mansour
e-mail: mansour.mekour@univ-saida.dz
ORCID: https://orcid.org/0000-0001-8486-9114
Department of Computer Science,
University of Saida, Dr. Taher Moulay, Algeria