An Approach Based on Sum Product Networks for Code Smells Detection

Mostefai Abdelkader

Original scientific article

Abstract—**From a software engineering perspective, a code smell refers to poor code structure. Many studies have shown that there is a significant negative relationship between code smells and code quality. Thus, many approaches have been proposed to detect and manage them. However, detecting code smells remains a challenging problem. This paper introduces a method (CSDSPN) based on a sum product network (SPN); a probabilistic deep architecture not yet evaluated in the context of code smell detection. SPNs are tractable density estimators that compactly represent a joint probability distribution. The main objective of this paper is to study the performance of a Sum-Product Network as a classifier for code smell detection. To fulfill this objective, the paper proposes an approach that utilizes a classifier based on an SPN trained on data from previous projects, to identify code smells in new source code. An empirical study was conducted to assess the effectiveness of the proposed method in detecting 'God Class,' 'Long Method,' and 'Feature Envy' code smells using well-known datasets. The empirical study evaluated the proposed approach against against seven standard and advanced machine learning models. The results of the study demonstrate the potential of the proposed method in effectively detecting code smells.**

Index Terms—**Code smells, Sum product network, Probability distribution, source code.**

I. Introduction

IN software engineering, it is crucial to develop high-quality software and maintain this quality throughout software evolution activities such as bug fixing, adding functionality, modifying features, and so on. Code smells, considered as poor code structures in software engineering [47], have a negative impact on various aspects of code quality, including understandability, testability, extensibility, reusability, and maintainability [3]. While code smells are not bugs, they have a negative relationship with software technical debt. Many studies have demonstrated a negative correlation between code smells and the understandability and maintainability of software systems [14]. Research has shown that the more code smells present in a system, the more expensive the evolution activities become, and the higher the likelihood of introducing additional bugs. Therefore, effectively detecting and managing code smells is an important activity from a software engineering perspective. While this activity incurs a cost, it is considered as a good strategy for reducing evolution costs and saving time in the long term. Therefore, detecting them is crucial to restore code quality through refactoring techniques [3] and to mitigate future maintenance costs and time [2]. Literature highlights that maintenance activities consume 80 % of the total cost of the development process [4] and code smells significantly complicate it [2], [5]–[7]. Despite that many approaches have been proposed to detect code smells, the literature shows that the problem remains a challenging one [8], [9], [38]–[42], [45]. Moreover, Fontana et al [1], [2], [42] indicated that we cannot deem any code smell detection technique superior to another and any approach can contribute to the process of prevention, detection, and fixing. While many approaches based on machine learning techniques have been proposed to detect code smells in software systems [39], the Sum-Product Network (SPN), a probabilistic deep architecture, while evaluated in bug prediction [50], has not yet been evaluated for code smell detection [39]. Sum-product networks (SPNs) are a type of density estimator with tractable inference properties, initially proposed by Poon and Domingos [35]. In many application domains, SPNs are seen as effective tools similar to neural networks [35], [36]. Another advantage of SPNs is their robustness to missing features. Additionally, many machine learning problems, such as classification, can be formulated as inference problems using a density estimator like SPNs. This idea has been previously evaluated in detecting defect in software systems with promising results [44].

Our main thesis is that SPNs are also well-suited for code smell detection. Therefore, this paper aims to evaluate the performance of SPNs as classifiers trained from data in detecting code smells. To achieve this goal, we propose a method called CSDSPN for detecting smells in source code. The proposed method is based on an SPN classifier built from existing 'smelly' and 'non-smelly' source codes from previous projects, using the LearnSPN algorithm [48]. The process uses a representation of source code based on a set of metrics extracted from it, rather than a textual representation. The method uses the learned SPN to determine whether a new source code is smelly or not, employing the MPE (Most Probable Explanation) inference method [35], [37]. An empirical study was conducted to assess the effectiveness of the proposed method in detecting 'God Class,' 'Long Method,' and 'Feature Envy' code smells using well-known datasets. The experiment used eighteen realistic datasets developed over years of work by experienced developers. Additionally, the empirical study evaluated the proposed approach against seven standard and advanced machine learning models to validate its

Manuscript received February 6, 2025; revised March 12, 2025. Date of publication May 14, 2025. Date of current version May 14, 2025. The associate editor prof. Tihana Galinac Grbac has been coordinating the review of this manuscript and approved it for publication.

M. Abdelkader is with the Computer Science Department, Faculty of Mathematics, Informatics, and Telecommunications, Dr. Tahar Moulay University, Saida, Algeria (e-mail: abdelkader.mostefai@univ-saida.dz).

Digital Object Identifier (DOI): 10.24138/jcomss-2024-0106

performance.

The main contributions of this paper are:

1) An approach based on a SPN to detect code smells in source code.
2) An empirical study that evaluates the effectiveness of the proposed approach on 18 well-known datasets.
3) A performance comparative study against seven standard and advanced machine learning models.

The remainder of the paper is organized as follows: Section II presents background information to help in understanding the proposed approach; Section III discusses the state of the art in the code smell detection domain; Section IV describes the proposed approach; Section V presents the empirical study; Section VI discusses the threats to the validity of our proposal; and Section VII concludes and presents future works.

II. Background

A. Code Smells

Fowler et al. [3] were the first to introduce the term "code smells" to the software engineering community in 1999 and presented the main refactoring techniques to restore the design quality of a software system. Code smells are symptoms that indicate low code quality. They negatively impact the main quality attributes such as understandability, reusability, and maintainability [12], [13]. The literature classifies code smells into two main categories: method-level and class-level smells [3], [9]. There are many code smells, including God Class, duplicated code, long method, long switch, long parameter list, and more. For further details about code smells, readers can refer to [11]. In this paper, we study the performance of the proposed approach on two types of code smells: God Class, which is a class-level smell, and feature-envy and long-method, which are method-level smells. These smells are defined as follows:

1) God Class: This smell indicates that a class is burdened with numerous responsibilities, violating the Single Responsibility Principle (SRP).
2) Feature Envy: This occurs when a method primarily uses members of other classes rather than its own, indicating a misplacement of functionality.
3) Long Method: This refers to methods that implement more than one functionality, often characterized by their large size and poor cohesion. Long methods are difficult to understand, extend, and modify. Additionally, they often violate the Single Responsibility Principle (SRP).

B. Sum-product Networks

Probabilistic graphical modelling is an approach that uses graph theory and probability to describe complex problems. These models represent a joint distribution in a compact way. Probabilistic graphical models, such as Bayesian networks and Markov networks, are widely used to model problems involving complex, uncertain knowledge. In this context, a solution to the problem is computed through inference. Examples of such inferences include marginalization and conditional probability computations. While these models allow questions to be answered via inference, the complexity of the inference process is NP-hard [37]. Sum-Product Networks (SPNs) [36] are tractable probabilistic graphical models. They have the ability to represent complex, highly multidimensional distributions compactly. They are computational networks, similar to traditional artificial neural networks, and are composed of nodes that compute either products or weighted sums of their inputs. The weights are strictly positive, and the sum and product nodes adhere to predefined structural constraints. The main advantages of SPNs over other probabilistic models, such as Bayesian networks and Markov networks, include fast and exact inference. With SPNs, inferences such as evidence computation, marginalization, and likelihood estimation are exact and tractable with respect to the size of the model. These properties make SPNs particularly attractive for application domains where machine learning is crucial and where inference must be both tractable and exact [37]. It is also worth noting that the structure and parameters of SPNs can be learned effectively from data. This gives them an advantage over other models, such as neural networks, which require the manual design of an effective structure for the task at hand.

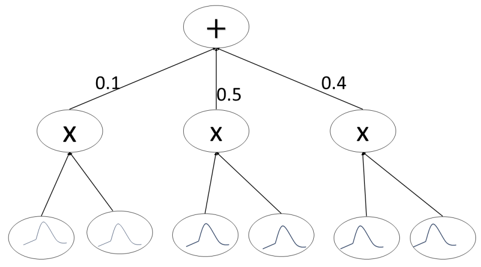

Fig. 1. An example of a SPN.

An SPN over a set of random variables (RVs) $\mathbf{X}$ is a computational network represented by a rooted, weighted, directed acyclic graph (DAG) $G = (N, E)$, where N represents nodes and E represents edges (see Figure 1). An SPN node corresponds to a probability function, and these nodes are categorized into three types: leaf, sum, or product. A sum node represents a weighted sum of probability distributions (a mixture model), given by:

$$p(x) = \sum_k w_k p_k(x) \tag{1}$$

A product node represents the product of probability distributions (a factorized model), given by:

$$p(x) = \prod_d p_d(x^d) \tag{2}$$

The value of the root node is the value of the SPN. Each leaf node corresponds to a univariate distribution over its variable. In SPNs, the sum and product nodes exist at both the root and intermediate levels and are organized into layers. Each layer contains nodes of a single type (sum or product). A layeri is typically followed by a layerj, which contains nodes of a different type than layeri. Each edge emanating from a sum node to its child has a strictly positive weight. Adavntages of

SPN Some advantages of Sum-Product Networks (SPNs) over traditional probabilistic models are:

1) Automatic Structure Design: There is no need to manually design the structure of the network.
2) Tractability: SPNs are tractable probabilistic models, meaning queries can be answered in polynomial time.
3) Robustness to Missing Features: SPNs are robust against missing features, making them suitable for real-world applications where data may be incomplete.

III. State of the Art

Code smell detection is still an active area of research today [19], [38]–[42]. Many techniques have been proposed to detect code smell in software systems [19]. These approaches are classified into three classes [19]. metrics-based [18], rule-based [18] and machine learning based approaches [19]. The metrics-based approach relies on quality metrics and a threshold value, that is difficult to obtain, for each metric. In contrast, the rule-based approach, each code smell is identified on the basis of some rules that are generally manually designed by domain experts and can be demanding in terms of effort. Hence Machine Learning (ML) techniques are the main approaches used to solve the code smell detection problem [21]. In this context, a classifier (such as Naïve Bayes, Logistic Regression, Neural Network, etc.) is trained using historical data from available projects to predict the presence of smells in a new project. Kreimer [22] proposed a method based on decision trees to detect design flaws (code smells) in object-oriented software. In [23], the authors proposed a solution to the problem of detecting if a part of code exhibits more than one smell using multi-label classification methods. The experiment utilized a multi-label dataset constructed from two code smells. The results demonstrated that the approach achieved good performance. Reis et al. [2] presented the results of an empirical study of an approach called the Crowdsmelling approach. This approach is a supervised machine learning approach based on the wisdom of the crowd, i.e., software developers. The crowd created oracles (datasets) that are later used by six machine learning algorithms. The results showed that high performance can be obtained using Naïve Bayes for God Class detection. For Long Method detection, high performance is achieved by AdaBoostM. Sharma et al. [24] explored the application of deep learning models in detecting smells without extensive feature engineering and the use of transfer learning. They trained their proposed model based on Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) with an autoencoder. In this scenario, the model was trained on Csharp examples and later evaluated on Java examples, and vice versa. The study's results showed that transfer learning exhibits comparable performance to direct learning.

Moha et al [25] proposed a method called DECOR and an instance of it named DETEX. The DÉCOR method defines all the specification and detection of code and design smells. The proposition was validated in terms of precision and recall on XERCES v2.7.0. In [26], the authors proposed an SVMCSD technique to detect code smells, based on support vector machine learning. The experiment was conducted on two open-source projects to evaluate the technique's performance in detecting God Class, Feature Envy, Data Class, and Long Method smells. The results indicated that the accuracy of SVMCSD outperformed DETEX in terms of precision and recall metrics. In [27], the authors conducted a large empirical study that evaluated 16 different machine learning algorithms for detecting four code smells (Data Class, Large Class, Feature Envy, Long Method). The study used 74 software systems. The results showed that all algorithms achieved a high performance on the cross-validation data set. Moreover, the study revealed that the J48 and Random Forest performed the best while support vector machines achieved the lowest performance. Di Nucci et al. [28] proposed a new dataset that contains more code smells and replicated the work in [26]. The authors argued that the code smell detection problem remains challenging and relevant, and more work needs to be done to structure datasets appropriately, along with selecting appropriate predictors. In [29], the authors propose the BDTEX (Bayesian Detection Expert) approach. This approach builds a Bayesian belief network using a Goal Question Metric (GQM) derived from the definitions of antipatterns. The approach was illustrated with the Blob antipattern and validated with Blob, Functional Decomposition, and Spaghetti code antipatterns. In [30], Aleksandar et al. presented an evaluation of the performance of multiple machine learning-based code smell detection models for detecting God Class and Long Method code smells against multiple metric-based heuristics. The study evaluated the performance of classically used code metrics against code embeddings (code2vec, code2seq, and CuBERT). Liu et al. [31] proposed a method based on deep learning models that automatically extract features from source code to detect code smells. To address the high demand for training data required by deep learning models, they proposed an automatic approach to generate such data. Milika et al. [32] explored the capacity of pretrained neural code embeddings for code smell detection. The results of the experiments, which are based on different code representations such as code metrics and neural code embeddings (CodeT5 and CuBERT), show that there is no clear difference between them. However, code embeddings have the potential to adapt and scale in response to new programming constructs. Liu et al. [33] proposed an approach that utilizes a Convolutional Neural Network (CNN) with a representation based on structural metrics and semantic (textual) information to detect the feature envy smell. Hadj-kacem and Bouassida [34] also utilized a representation that combines structural and semantic information extracted separately using conventional and deep learning methods. The empirical study conducted on five open-source projects (JHotDraw, Apache Karaf, Freemind, Apache Nutch, and JEdit) showed that the proposed approach can be effective in detecting bad smells. In [45], the authors presented an unsupervised approach based on a feature engineering process for code smell detection. The empirical study conducted on four datasets and compared with supervised approaches for code smell detection showed that the approach is as effective as the supervised approaches.

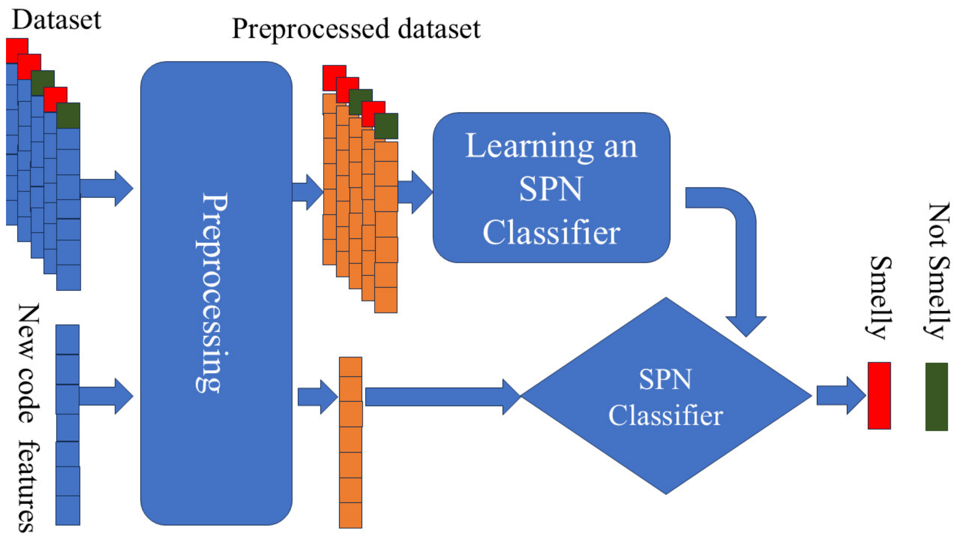

Fig. 2. The CSDSPN method.

IV. The Proposed Approach

Sum-Product Networks (SPNs) [36] are tractable density estimators that compactly represent a joint probability distribution. Many machine learning problems, such as classification, can be formulated as inference problems using a density estimator like SPNs. This density estimator can either be manually designed or trained from data. Density estimation is the unsupervised activity that aims at learning an estimator (i.e., Model) of an unobservable underlying joint probability density function (PDF) over a set of random variables (RVs). The estimation process is based on observed data (random sample) of this unobservable PDF. With the obtained estimator, inference can be done to answer queries such as marginalization, likelihood, conditional and so on. The code smell detection problem is formulated as a classification problem using SPN as an estimator (i.e., classifier) trained from data.

The proposed method (CSDSPN) that support this idea is a classifier based on a SPN. The classifier is learned from the available code smells datasets and subsequently utilized to predict code smells in new projects. The workflow of the CSDSPN method is depicted in Figure 2, while the core process is elaborated upon in the following section. The CSDSPN method is mainly composed of two activities. The first activity consists of preprocessing the datasets, while the second activity consists of learning the classifier from the processed data.

A. Data Preprocessing

Data preprocessing is a crucial task conducted to address noisy, missing, and inconsistent data within existing datasets before proceeding to more advanced activities. The principal activities of this process include data cleaning, data integration, data reduction, and data transformations [43]. In the data mining community, it is widely acknowledged that normalization (i.e., standardization) has the potential to enhance the performance of data mining techniques. Normalization techniques aim to scale attribute data to give all attributes equal weight, thereby reducing them to a smaller range. Various techniques can be employed for dataset normalization, including min-max normalization, z-score normalization, and normalization by decimal scaling [43]. The proposed method utilizes the min-max normalization technique.

B. Learning an SPN Classifier

Problem solving with an SPN classifier generally requires identifying an effective structure and parameterization. While it's possible to manually design a valid SPN classifier structure, this process is time-consuming and demands significant domain knowledge, followed by weight learning [35], [37]. A more efficient approach, widely adopted in the literature, is to learn both the structure and the parameter weights of the SPN directly from the data [44]. This method ensures that the resulting SPN effectively represents the problem. In our approach, the SPN classifier for code smell detection is built using the LearnSPN algorithm [35], [48] from our preprocessed training dataset. LearnSPN is a foundational learning algorithm, commonly used to construct a valid SPN from data. The algorithm takes as input a set of N i.i.d. samples, $\{x_i\}$, where $i = 1, \ldots, N$, over random variables X. These samples are typically organized as a matrix, with instances as rows and variables as columns. The goal of the learning process is to learn an estimator for the joint probability distribution P_X, using these samples. The samples used to obtain P_X consist of the elements in the training dataset. Each sample represents a collection of metrics extracted from the source code, along with its class, denoted as $(x_1, \ldots, x_N)$, where the variable x_N represents the class of the source code (smelly code or not). The LearnSPN algorithm is composed of two fundamental operations: the chop operation, which divide the variables (columns) in the dataset, and the slice-a-slice operation, which clusters instances (rows). The SPN learning process is summarized in Algorithm 1 [35].

Algorithm 1 LearnSPN

```
Require: A dataset $D$ containing a set of elements $S$ over variable $X$.
Ensure: An SPN representing a distribution over $X$ learned from $D$.
 1: Begin
 2: if $|X| = 1$ then
 3:     return univariate distribution estimated from the variable's values in $S$
 4: else
 5:     Divide $X$ into approximately independent subgroups $X_j$.
 6:     if success then
 7:         return $\prod_j$ LearnSPN$(S, X_j)$
 8:     else
 9:         Divide $S$ into subgroups of similar instances $S_i$.
10:         return $\sum_i \frac{|S_i|}{|S|} \cdot$ LearnSPN$(S_i, X)$
11:     end if
12: end if
13: End
```

C. Detecting Code Smells

To compute the class of a new source code M, we use the Most Probable Explanation (MPE) inference method which is a special case of the maximum a-posteriori (MAP) method [35], [37].

1) The Most Probable Explanation Inference Method: Let P_V be a distribution over a set of random variables V, represented by the obtained SPN from the previous step. Let F denote the evidence, which is the set of metrics E extracted from M, where $|F| = |V| - 1$. Let C be the class of the source code M. The identification of the class C of the new source code M consists of computing the posterior probability $P(C \mid F)$. Formally, MPE$(C, X) = \arg\max_C P(C \mid F)$.

V. Empirical Study

This section presents an empirical study on using SPNs for detecting code smells, specifically focusing on God Class, Feature Envy, and Long Method code smells. The primary objective of this experiment is to assess the performance of CSDSPN method in detecting these code smells. The main hypothesis of this study is that SPNs are suitable for detecting code smells in software systems. More specifically, the empirical study aims to answer the following research question: RQ1: Can a classifier based on a sum product network be effective in detecting code smells in software systems?
To address RQ1, a classifier based on a SPN is trained on a training set and evaluated on a test dataset. This process is conducted using 18 well-known datasets, which are described in the following section.

A. Performance

Code smell detection is a classification problem, and due to the imbalanced nature of code smell datasets [2], [43], we utilize Area Under the receiver operating characteristic Curve (AUC), Accuracy, and F-measure (F1) to assess the performance of the CSDSPN model. These metrics are widely adopted by the code smell detection community [2]. Higher values of these measures indicate better performance. The AUC [2], [43] metric is widely recommended by researchers for evaluating the performance of predictors, as it has the power to reveal their real potential [2], [43]. Unlike other metrics, AUC is not sensitive to changes in data distributions [2], [43]. A value close to 1 indicates better classifier performance, while a value close to 0.5 suggests performance similar to random guessing. A value close to 0 indicates performance worse than random guessing [43].

These metrics are calculated based on the following quantities: true positives (TP), false negatives (FN), false positives (FP), and true negatives (TN), where:

- TP: Instances that are actually positive and classified by the classifier as positive.
- FN: Instances that are actually positive but classified by the classifier as negative.
- FP: Instances that are actually negative but classified by the classifier as positive.
- TN: Instances that are actually negative and classified by the classifier as negative.

These metrics are calculated using formulas 1 to 4.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{3}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{4}$$

F1 (F-measure) is the harmonic mean of precision and recall:

$$\text{F-measure} = \frac{2 \times \text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}} \tag{5}$$

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}} \tag{6}$$

B. Baseline Methods

The performance of our CSDSPN is evaluated relative to the effectiveness of other code smell detectors. We compare it against seven baseline classifiers commonly used in evaluating the effectiveness of code smell detectors. These detectors include AdaBoost(AB), J48, Random Forest (RF), SMO (Sequential Minimal Optimization algorithm), Multilayer Perceptron (MLP), Convolutional neural network (CNN) [1] and Naïve Bayes (NB) [2], [43].

C. Setup

To address RQ1, The study used a stratified 10-fold cross-validation approach to assess the performance of CSDSPN and compare it with many advanced machine learning models. In this approach, the dataset is divided into 10 folds of equal size, ensuring a balanced distribution of code smells. The validation process consists of 10 iterations, where in each iteration, 9 folds are used as the training set, and the remaining fold is used as the test set.

1) Hyperparameter Tuning: The performance of CSDSPN is highly dependent on hyperparameters. It is well-known that finding the best hyperparameter values for a model is a challenging problem, often referred to as hyperparameter tuning. Many approaches have been proposed to address this issue, ranging from grid search to more advanced metaheuristic-based methods [16]. In this study, the CSDSPN approach was evaluated using a grid-based tuning strategy. In this strategy, the hyperparameter Minimum number of rows required to split was varied across the values 5, 10, 15, 20, 25, 30, 40 to identify the optimal value.

The SPN classifier parameters were configured as follows: all features were considered Gaussian variables, except for the class label and binary metrics, which were set to Bernoulli variables. Leaf distributions are learned using Maximum Likelihood Estimation (MLE) [43]. The K-Means algorithm [35], [43] is employed for splitting rows (data points), while the Randomized Dependency Coefficient (RDC) [35] is utilized for splitting columns (features, variables). The minimum number of rows required to continue the split is adjusted until a satisfactory result is achieved. Table I presents a summary of the CSDSPN and baseline models' hyperparameter values used in the experiment. The implementation of the CSDSPN method utilized the DeeProb-kit, a Python library publicly available on GitHub (https://github.com/deeprob-org/deeprob-kit). All experiments were conducted in Python on the Colab platform.

D. Datasets

The empirical study utilized 18 well-known datasets [2] to evaluate the proposed method. The main reason for choosing these datasets is that they are more realistic than the proposed datasets in the literature as indicated by [2], [22]. These datasets were created and validated by numerous developers over three years of work. The main principle of the construction process is to use different tools to identify code smells from various software projects. Then, a fragment flagged by the tools as "smelly" is reviewed by multiple developers before being included in the final dataset. Table II presents the characteristics of the datasets used to evaluate the proposed approach. The first column gives the name of the dataset, the second indicates the code smell to which the dataset refers, the third presents the total number of cases, the fourth presents the number of true instances, the fifth presents the percentage of true instances, the sixth presents the number of false instances, and the seventh presents the percentage of false instances. The identifier of each dataset corresponds to the year or years of its constitution. For example, the identifier '2018' represents the dataset from the year 2018, while '2018+2019+2020' represents the dataset produced by combining datasets from the years 2018, 2019, and 2020. According to José et al. [2], the datasets are not normalized in size to balance the two classes, as done by other researchers [22]. This deliberate decision aims to make the detection more challenging, mirroring real-world scenarios [2].

The datasets used in this study are accessible from https://github.com/dataset-cs-surveys/Crowdsmelling.

E. Experiment Results

This section presents the results of the empirical study conducted to answer the proposed question RQ1.
We evaluate the performance of our method in detecting three code smells: Long method, Feature envy, and God class, across 18 datasets. The obtained results are compared with the performance of the baseline approaches. Since multiple metrics (accuracy, F-measure, AUC) can be used for comparison, and the best model can vary depending on the chosen metric, we decided to first analyze the performance according to the AUC metric. Subsequently, we assess accuracy and F-measure.

1) AUC: Figure 3 shows the box plot of the AUC for the CSDSPN approach and the baseline models on the feature envy datasets. Figure 4 shows the box plot of the AUC for the CSDSPN approach and the baseline models on the god class datasets. Figure 5 shows the box plot of the AUC for the CSDSPN approach and the baseline models on the long method datasets. These box plots illustrate the range of AUC values. On each box, the median is indicated by a red central mark, while the 25th and 75th percentiles are represented by the edges. Outliers are depicted as small circles. The main observation that can be drawn from these figures is that no model outperformed all others in terms of AUC across every dataset. From Figure 3, it can be observed that the AUC for CSDSPN on the Feature Envy dataset ranged between 0.50 and 0.60 (see Table III). The table also shows that CSDSPN achieved the best median average AUC on the 2019, 2019+2018, and 2020+2019 Feature Envy datasets, with average AUC values of 0.55, 0.60, and 0.54, respectively. The figure indicates that CSDSPN outperformed both the MLP and CNN models in terms of average AUC on the 2019, 2020, 2019+2018, and 2020+2019 datasets. On the 2020 dataset, the best AUC was 0.50, achieved by AB, SMO, RF, CNN, and J48. On the 2020+2019+2018 dataset, the best AUC of 0.53 was achieved by MLP.

From Figure 4, it can be observed that the average AUC for CSDSPN ranged between 0.68 and 0.86 (see Table III). CSDSPN achieved the best median average AUC on the 2018, 2019, and 2019+2018 God Class datasets, with average AUC values of 0.68, 0.76, and 0.76, respectively. The figure shows that CSDSPN outperformed both MLP and CNN models on the 2018, 2019, and 2019+2018 God Class datasets. It also performed better than MLP on the 2020+2019+2018 dataset, where it matched the performance of CNN. On the 2020 dataset, NB, RF, and SMO were found to be effective, achieving a score of 0.86, with RF and SMO being the top performers with an AUC score of 0.81. On the 2020+2019 dataset, RF and SMO were again the best performers in terms of AUC, with a score of 0.81. On the 2020+2019+2018 dataset, NB was the best performer in terms of AUC, followed by SPN, which was the second-best performer along with RF and AB.

From Figure 5, it can be observed that the AUC for CSDSPN ranged between 0.54 and 0.82 (see Table III). The figure shows that CSDSPN achieved the best median average AUC compared to NB on all long method datasets. Additionally, CSDSPN achieved the best median average AUC compared to MLP on the 2019+2018 long method datasets. In general, on the long method datasets, CSDSPN achieved competitive average scores compared to the other models. More specifically, AB achieved the best average on the 2018, 2019, 2018+2019, and 2020+2019 datasets, with values of 0.54, 0.66, 0.66, and 0.78, respectively. On the 2020 dataset, the best average AUC was achieved by CNN with a score of 0.82, while on the 2020+2019+2018 dataset, the best average

TABLE I
SUMMARY OF CSDSPN AND BASELINE MODELS' HYPERPARAMETER VALUES USED IN THE EXPERIMENT.

Model	Parameter	Value
SPN	Learn leaf	Mle
	Split rows (Minimum number of rows required to split)	Kmeans
	Split cols	RDC
	min_rows_slice	Varied between: 10-50
AB	max_depth	2
	n_estimator	100
	learning_rate	0.1
NB	-	-
J48	criterion	Entropy
	max_depth	10
	min_samples_split	10
	min_samples_leaf	5
RF	n_estimators	100
	criterion	Entropy
	max_depth	20
	min_samples_split	10
	min_samples_leaf	5
MLP	First hidden_layer_size	100
	Second hidden_layer_size	50
	solver	Adam
	alpha	0.0001
	Max_iteration	100
SMO	Kernel	Rbf
	C	1.0
	gamma	Scale
CNN	First Conv1D Layer kernel Filters, size, and activation	3, 64, and ReLu
	First MaxPooling1D Layer pool size	2
	Second Conv1D Layer kernel Filters, size, and activation	32, 3, and ReLu
	Second MaxPooling1D Layer pool size	2
	Fully Connected Layer Units, activation	50, ReLu
	Dropout rate	0.5

TABLE II
STATISTICS OF THE DATASETS.

Dataset	Code smell	No of Cases	True	% True	False	% False
2018	Feature Envy	10	3	30%	7	70%
2019	Feature Envy	197	110	56%	87	44%
2019+2018	Feature Envy	207	113	55%	94	45%
2020	Feature Envy	123	79	64%	44	36%
2020+2019	Feature Envy	320	189	59%	131	41%
2020+2019+2018	Feature Envy	330	192	58%	138	42%
2018	God class	22	8	36%	14	64%
2019	God class	129	74	57%	55	43%
2019+2018	God class	151	82	54%	69	46%
2020	God class	136	84	62%	52	38%
2020+2019	God class	265	158	60%	107	40%
2020+2019+2018	God class	287	166	58%	121	42%
2018	Long Method	59	24	41%	35	59%
2019	Long Method	414	180	43%	234	57%
2019+2018	Long Method	473	204	43%	269	57%
2020	Long Method	853	350	41%	503	59%
2020+2019	Long Method	1267	530	42%	737	58%
2020+2019+2018	Long Method	1326	554	42%	772	58%

AUC scores were 0.77, achieved by Adaboost, J48, and RF. A statistical test in terms of AUC was conducted to identify

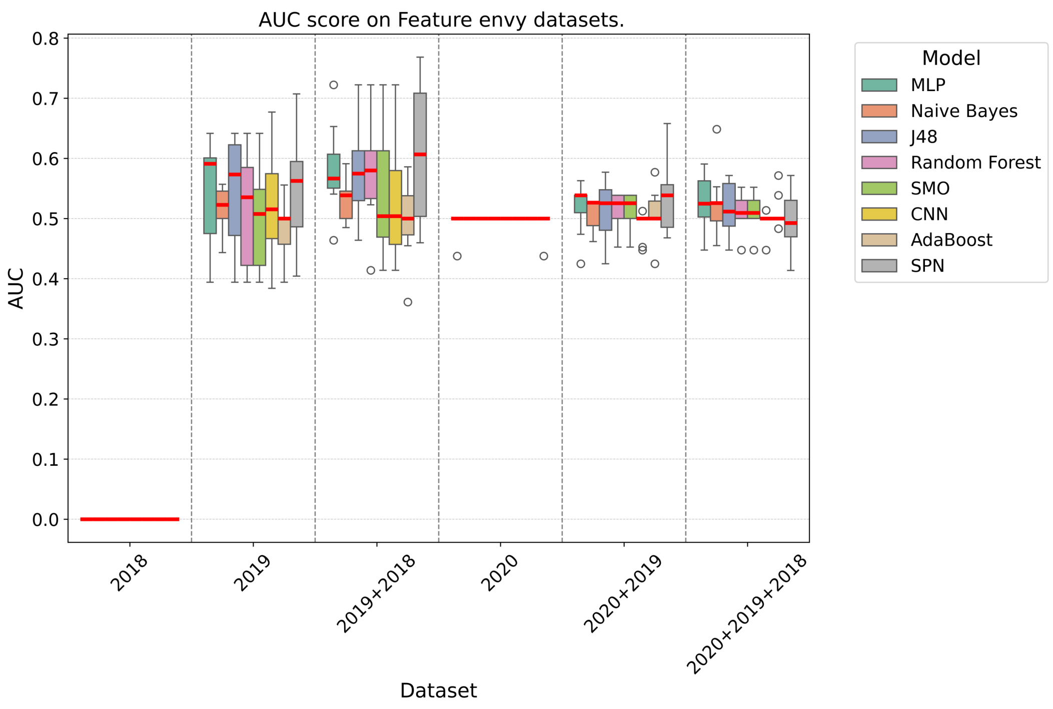

Fig. 3. Box plot of AUC for the CSDSPN approach and baseline methods on Feature Envy datasets.

whether there are significant differences between the CSDSPN and baseline model results. Due to the non-normal distribution of the data, the Mann-Whitney U Test [15] was chosen for this analysis. Table IV presents the p-values for CSDSPN and baseline models in terms of AUC scores. Table V presents the mean AUC of CSDSPN and baseline models on all datasets. Table IV shows that all the obtained p-values are greater than 0.05, indicating that, on average, there are no significant differences between CSDSPN and the baseline models, which means that CSDSPN is as effective as any of these models.

2) Accuracy and F-measure: Table III presents the obtained accuracy and F-measure values of CSDSPN and the seven baseline techniques on the 18 datasets for the three code smells: God Class, Feature Envy, and Long Method. From this table, it can be observed also that no single method outperforms all others in every case in terms of accuracy and F1-score. This table shows that on the 2018 God Class dataset, CSDSPN achieved the best F1 and accuracy scores, with values of 0.78 and 0.67, respectively. On the 2019 dataset, the best scores were achieved by MLP, with an F1 value of 0.81 and an accuracy value of 0.78. NB performed best in terms of F1 and accuracy on the 2019+2018 dataset, with values of 0.81 and 0.78, respectively. On the 2020 dataset, NB and RF performed best in terms of F1 and accuracy, with scores of 0.92 and 0.89, respectively. On the 2020+2019 dataset, SPN, AB, J48, RF, MLP, and CNN were the best performers in terms of F1, with a score of 0.86. NB, RF, and SMO performed better in terms of accuracy, with a score of 0.83. On the 2020+2019+2018 dataset, NB was the best performer in terms of AUC, F1, and accuracy, while SPN was the second-best performer, along with RF and AB. On the same dataset, the best F1 score (0.74) was achieved by AB, and the best accuracy score (0.59) was achieved by J48 and MLP. In the case of the Long Method datasets, the best F1 and accuracy scores were achieved by AB on the 2018, 2019, 2018+2019, 2020+2019, and 2020+2019+2018 datasets. On the 2020 dataset, the best F1 and accuracy scores were achieved by AB, CNN, and SMO, with an F1 score of 0.78 and an accuracy score of 0.81.

3) Discussion: From these figures and tables, we can observe that the AUC for SPN in the case of the long method smell was between 0.54 and 0.82. The AUC for the GC case ranged between 0.68 and 0.86. These values indicate that the CSDSN approach is effective in detecting long method and God class smells, as they show that the CSDSPN model performs better than random guessing. However, in the case of feature envy, the CSDSPN AUC for the feature envy dataset ranged between 0.50 and 0.60, except for the 2018 dataset, where all models failed due to the small dataset size (only 10 instances). The baseline models' performance in terms of AUC was also between 0.5 and 0.6. These values suggest that all models struggle to detect the feature envy code smell, and the results indicate that the models are essentially providing random guesses. The main reason for this is that feature envy is a type of smell where the method is more dependent on information existing in other classes than the its own class. A representation based on metrics alone is insufficient to reveal such semantic relationships, so more advanced representations need to be devised to detect this type of smell.

4) Effect of Model Parameters: This section explores how the minimum number of rows required to continue the split (MRS) affects the performance of the model. Figure 6 presents the effect of the parameter MRS on the AUC for Feature Envy in the case of the 2019 dataset, the effect of the parameter

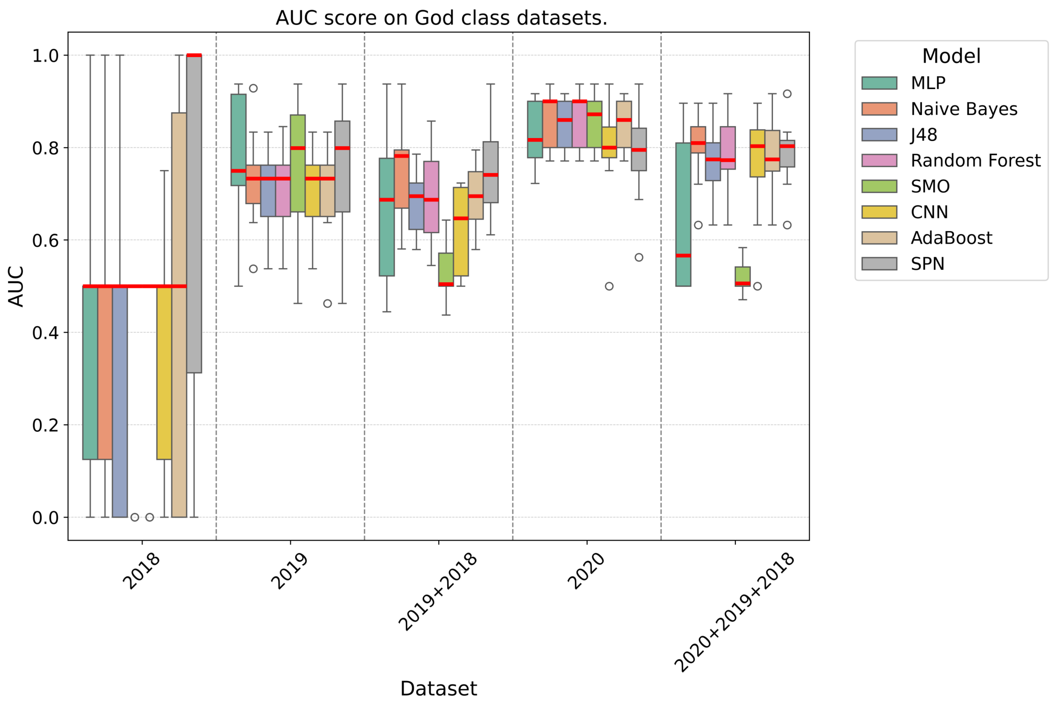

Fig. 4. Box plot of AUC for the CSDSPN approach and baseline methods on God Class datasets.

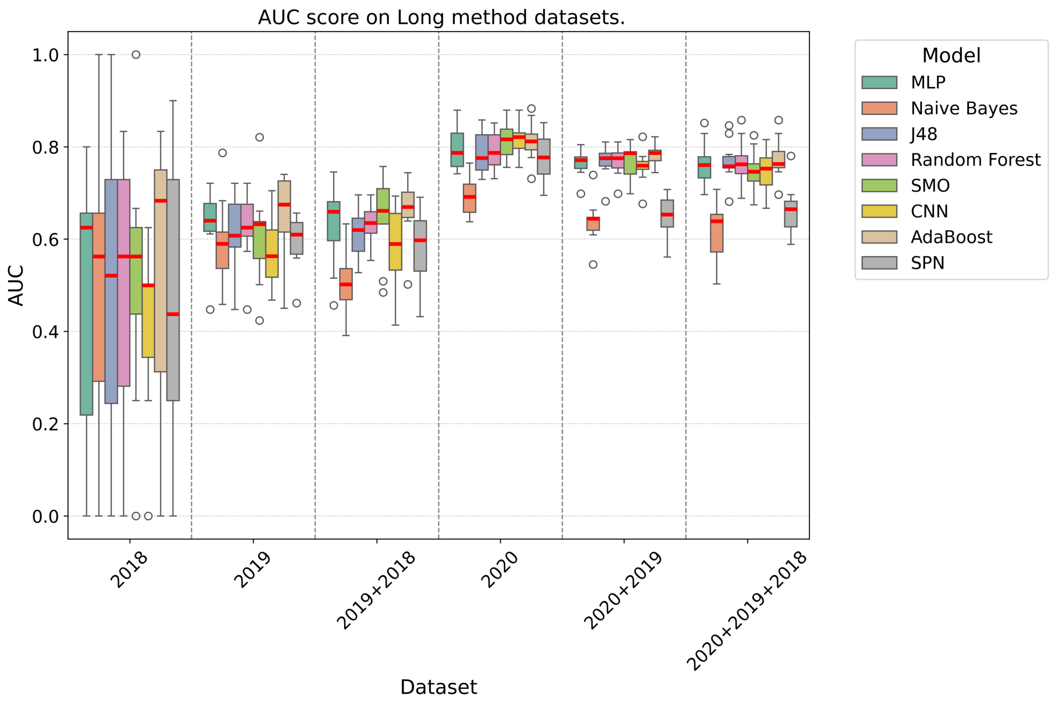

Fig. 5. Box plot of AUC for the CSDSPN approach and baseline methods on Long Method datasets.

MRS on the AUC for Long Method in the case of the 2020 dataset and the effect of the parameter MRS on the AUC for God class in the case of the 2018 dataset. From this figure, we conclude that the AUC values of the method are sensitive to the MRS values, especially in the case of the Feature Envy and Long Method datasets. In contrast, for the God class dataset, the variation in AUC values was found to be low. This figure demonstrates that it is difficult to determine the best value in advance. Therefore, it is crucial to explore approaches that aid in identifying the optimal value.

5) Summary of RQ1 Results: From Figures 3 4 5 and Table III, it is clear that our approach can effectively detect code smells such as God class and Long Method. However, detecting the Feature Envy code smell remains a challenging

TABLE III
THE AVERAGE F1, ACCURACY (ACC) AND AUC RESULTS OF ALL METHODS FOR THE THREE CODE SMELLS ON ALL DATASETS.

Datasets	**Model**	**FE**			**God Class**			**Long Method**		
		F1	**Acc.**	**AUC**	**F1**	**Acc.**	**AUC**	**F1**	**Acc.**	**AUC**
2018	AB	0.00	**0.70**	0.00	0.30	0.62	0.45	**0.46**	0.62	**0.54**
	CNN	0.00	0.40	0.00	0.13	0.53	0.38	0.05	0.50	0.41
	J48	0.00	**0.70**	0.00	0.33	0.52	0.40	0.44	**0.64**	0.51
	MLP	0.00	**0.70**	0.00	0.22	0.60	0.40	0.33	0.54	0.47
	NB	0.00	0.30	0.00	0.17	0.47	0.40	0.39	0.59	0.50
	RF	0.00	**0.70**	0.00	0.00	0.63	0.40	0.39	0.60	0.50
	SMO	0.00	**0.70**	0.00	0.00	0.63	0.40	0.34	0.59	0.52
	SPN	0.00	**0.70**	0.00	**0.67**	**0.78**	**0.68**	0.44	0.55	0.47
2019	AB	**0.66**	0.53	0.48	0.71	0.71	0.70	**0.63**	**0.65**	**0.66**
	CNN	0.62	0.54	0.53	0.76	0.73	0.71	0.48	0.57	0.57
	J48	0.62	**0.56**	0.54	0.76	0.73	0.71	0.55	0.63	0.62
	MLP	0.62	**0.56**	0.55	**0.81**	**0.78**	**0.76**	0.59	0.64	0.63
	NB	0.17	0.47	0.51	0.80	0.75	0.73	0.53	0.60	0.59
	RF	0.62	0.54	0.52	0.77	0.74	0.72	0.56	0.63	0.62
	SMO	0.62	0.53	0.51	0.67	0.74	0.77	0.53	0.61	0.61
	SPN	0.52	0.54	**0.55**	0.66	0.72	0.75	0.54	0.60	0.60
2019+2018	AB	**0.64**	0.53	0.50	0.69	0.69	0.70	**0.65**	**0.65**	**0.66**
	CNN	0.63	0.54	0.53	0.56	0.62	0.62	0.48	0.60	0.58
	J48	0.62	**0.58**	0.58	0.68	0.68	0.68	0.55	0.62	0.61
	MLP	0.62	**0.58**	0.58	0.75	0.68	0.67	0.57	0.63	0.63
	NB	0.22	0.50	0.53	**0.79**	**0.76**	0.75	0.35	0.53	0.50
	RF	**0.64**	**0.58**	0.58	0.72	0.70	0.69	0.59	0.64	0.64
	SMO	0.62	0.55	0.54	0.71	0.57	0.53	0.60	**0.65**	0.65
	SPN	0.60	0.60	**0.60**	0.69	0.74	**0.76**	0.51	0.59	0.58
2020	AB	**0.78**	**0.64**	**0.50**	0.90	0.87	0.85	**0.78**	**0.81**	0.81
	CNN	**0.78**	**0.64**	**0.50**	0.78	0.80	0.79	**0.78**	**0.81**	**0.82**
	J48	**0.78**	**0.64**	**0.50**	0.90	0.87	0.85	0.75	0.79	0.79
	MLP	**0.78**	0.63	0.49	0.85	0.83	0.83	0.75	0.80	0.80
	NB	0.06	0.37	**0.50**	**0.92**	**0.89**	**0.86**	0.60	0.72	0.69
	RF	**0.78**	**0.64**	**0.50**	**0.92**	**0.89**	**0.86**	0.76	0.80	0.79
	SMO	**0.78**	**0.64**	**0.50**	0.91	0.88	**0.86**	**0.78**	**0.81**	0.81
	SPN	**0.78**	0.63	0.49	0.71	0.74	0.78	0.73	0.79	0.78
2020+2019	AB	**0.74**	0.59	0.51	**0.86**	0.82	0.80	**0.75**	**0.78**	**0.78**
	CNN	0.73	0.58	0.49	**0.86**	0.82	0.80	0.73	0.75	0.76
	J48	0.73	0.59	0.52	**0.86**	0.82	0.80	0.73	0.77	0.77
	MLP	**0.74**	**0.60**	0.52	0.80	0.79	0.79	0.73	0.76	0.77
	NB	0.13	0.43	0.51	**0.86**	**0.83**	0.80	0.54	0.67	0.64
	RF	**0.74**	**0.60**	0.52	**0.86**	**0.83**	**0.81**	0.73	0.77	0.77
	SMO	**0.74**	**0.60**	0.51	**0.86**	**0.83**	**0.81**	0.74	0.77	0.77
	SPN	0.61	0.56	**0.54**	**0.86**	0.82	0.80	0.64	0.63	0.65
2020+2019+2018	AB	**0.74**	0.59	0.51	0.84	0.81	0.79	**0.75**	**0.77**	**0.77**
	CNN	0.73	0.58	0.50	0.84	0.79	0.77	0.71	0.75	0.75
	J48	0.73	**0.59**	0.52	0.83	0.79	0.77	0.74	0.76	**0.77**
	MLP	0.72	**0.59**	**0.53**	0.78	0.70	0.65	0.73	0.76	0.76
	NB	0.16	0.45	0.52	**0.85**	**0.82**	**0.80**	0.49	0.66	0.62
	RF	0.73	0.58	0.51	**0.85**	0.81	0.79	0.73	0.76	**0.77**
	SMO	0.73	0.58	0.51	0.74	0.59	0.52	0.71	0.75	0.75
	SPN	0.68	0.55	0.50	0.84	0.80	0.79	0.56	0.66	0.66

task. The statistical test in terms of AUC demonstrated that CSDSPN is at least as effective as the baseline models. In conclusion, the CSDSPN method shows promise in detecting code smells.

VI. THREATS TO VALIDITY

A. Internal Validity

the proposed method was implemented using the Deepro library, a Python Library for Deep Probabilistic Modeling. The parameters utilized in the empirical study include Maximum Likelihood Estimation (MLE) for leaf distribution learning, K-Means for row splitting, and randomized dependency coef-

TABLE IV
P-VALUES FOR AUC SCORES COMPARING CSDSPN AND BASELINE MODELS.

Model	p-value
AB	0.89
CNN	0.61
J48	0.97
MLP	0.89
NB	0.55
RF	0.91
SMO	0.63

TABLE V
MEAN AUC SCORES OF CSDSPN AND BASELINE MODELS.

Model	Average AUC
AB	0.61
CNN	0.58
J48	0.61
MLP	0.60
NB	0.58
RF	0.61
SMO	0.59
SPN	0.61

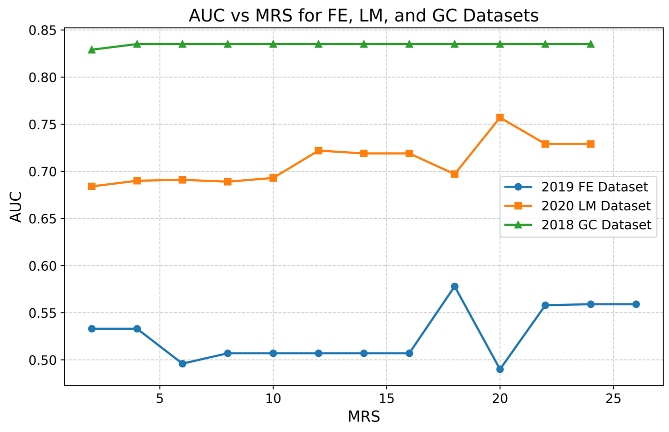

Fig. 6. Effect of MRS value on the AUC for Feature Envy, God Class and Long Method datasets

ficient (RDC) for variable splitting. The minimum number of rows required to continue the split was varied until a satisfactory result was achieved. However, due to this variability, there is a potential threat to internal validity. This problem is commonly known as hyperparameter tuning as decribed in the previous sections and involves finding the best parameters for models. It is well-studied in the literature, and many approaches have been proposed to solve it. Therefore, it is advisable to design controlled experiments for hyperparameter tuning based on these proposed approaches to mitigate this threat.

B. External Validity

External validity concerns the generalizability of the study results. This study aims to assess the effectiveness of the proposed method in detecting God Class, Long Method, and Feature Envy code smells across 18 datasets. As such, the obtained results can only be generalized to these specific datasets and tasks mentioned earlier. Additionally, These datasets were created and validated by numerous developers over several years of work. However, it is important to note that the dataset may still contain inconsistencies, as the decision to label a code fragment as ”smelly” remains subjective. Therefore, conducting additional studies on different datasets and tasks is essential to obtain a more accurate understanding of the method’s real-world performance.

VII. CONCLUSION

This paper proposed a novel method for code smell detection based on Sum Product Networks. The approach formulates the code smell detection problem as a classification task using an SPN learned from historical data. The proposed method, CSDSPN, which supports this idea, first preprocesses the dataset, then an SPN classifier is learned from the data using the LearnSPN algorithm. The class (smelly or not smelly) of a new code source is determined using the Most Probable Explanation inference method by feeding its representation into the learned SPN. The evaluation of the proposed approach on eighteen well-known datasets and against seven standard and advanced machine learning models demonstrates its potential in detecting code smells. Additionally, a statistical test conducted to determine whether there were significant differences between the results of the CSDSPN and the baseline model highlighted the need for further studies on larger datasets to better assess the true potential of the proposed approach. In our future work, we will explore the performance of our approach on various types of smells. Additionally, we plan to investigate the impact of different SPN learning methods, as well as the effect of hyperparameters on the performance of the CSDSPN approach. Another important direction will involve studying potential enhancements to further improve the effectiveness of the proposed method.

REFERENCES

[1] Y. LeCun, Y. Bengio, G. Hinton, “Deep learning.” *Nature*, 521(7553): 436–444, 2015.

[2] J. P. d. Reis, F. B. e. Abreu, and G. d. F. Carneiro, “Crowdsmelling: A preliminary study on using collective knowledge in code smells detection,” *Empirical Software Engineering*, vol. 27, p. 69, 2022.

[3] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[4] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, “Detecting defects in object-oriented designs: Using reading techniques to increase software quality,” in *Proceedings of the 14th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM Press, 1999, pp. 47–56.

[5] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. G. da Fonseca, “Performance assessment of multiobjective optimizers: An analysis and review,” *IEEE Transactions on Evolutionary Computation*, vol. 7, no. 2, pp. 117–132, 2003.

[6] D. Dig, K. Manzoor, R. Johnson, and T. Nguyen, “Refactoring-Aware Configuration Management for Object-Oriented Programs,” in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, 2007, pp. 427–436.

[7] T. Hall, M. Zhang, D. Bowes, and Y. Sun, “Some Code Smells have a Significant but Small Effect on Faults,” *Transactions on Software Engineering and Methodology (TOSEM)*, 2014.

[8] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, “A review-based comparative study of bad smell detection tools,” in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, 2016.

[9] A. Alazba and H. Aljamaan, “Code smell detection using feature selection and stacking ensemble: An empirical investigation,” *Information and Software Technology*, vol. 138, p. 106648, 2021.

[10] U. Mansoor, M. Kessentini, B. R. Maxim, and K. Deb, "Multi-objective code-smells detection using good and bad design examples," *Software Quality Journal*, vol. 25, no. 2, pp. 529–552, 2017.

[11] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012.

[12] J. Pérez, "Refactoring planning for design smell correction: Summary, opportunities and lessons learned," in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, IEEE Computer Society, 2013.

[13] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," *Journal of Systems and Software*, vol. 86, pp. 2639–2653, 2013.

[14] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 59–69.

[15] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Ann. Math. Stat.*, vol. 18, no. 1, pp. 50–60, 1947.

[16] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," in *Proc. Int. Conf. Learning Representations*, 2018, pp. 1–48.

[17] E. Alpaydin, *Introduction to Machine Learning*. MIT Press, 2014.

[18] S. Charalampidou, A. Ampatzoglou, and P. Avgeriou, "Size and cohesion metrics as indicators of the long method bad smell: An empirical study," in *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2015.

[19] 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*, vol. 36, no. 1, pp. 20–36, 2010.

[20] K. Alkharabsheh, Y. Crespo, E. Manso, and J. A. Taboada, "Software design smell detection: A systematic mapping study," *Software Quality Journal*, vol. 27, no. 3, pp. 1069–1148, 2019.

[21] J. Pereira dos Reis, F. Brito e Abreu, G. de Figueiredo Carneiro, and C. Anslow, "Code smells detection and visualization: A systematic literature review," *Archives of Computational Methods in Engineering*, 2021.

[22] F. A. Fontana, M. V. Mantylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[23] J. Kreimer, "Adaptive detection of design flaws," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 4, pp. 117–136, 2005.

[24] T. Guggulothu and S. A. Moiz, "Code smell detection using multi-label classification approach," *Software Quality Journal*, vol. 28, pp. 1063–1086, 2020.

[25] 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, p. 110936, 2021.

[26] 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*, vol. 36, no. 1, pp. 20–36, 2009.

[27] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[28] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?" in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 612–621.

[29] F. Khomh, S. Vaucher, Y. Guéhéneuc, and H. A. Sahraoui, "Bdtex: A gqm-based bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011.

[30] 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*, vol. 204, p. 117607, 2022.

[31] L. Hui, J. Jiahao, X. Zhifeng, B. Yifan, Z. Yanzhen, and Z. Lu, "Deep learning based code smell detection," *IEEE Transactions on Software Engineering*, 2021.

[32] M. Škipina, J. Slivka, N. Luburić, and A. Kovačević, "Automatic detection of Feature Envy and Data Class code smells using machine learning," *Expert Systems with Applications*, vol. 243, p. 122855, 2024.

[33] H. Liu, Z. Xu, and Y. Zou, "Deep learning-based Feature Envy detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[34] M. Hadj-Kacem and N. Bouassida, "Improving the identification of code smells by combining structural and semantic information," in *International Conference on Neural Information Processing*, 2019.

[35] C. J. Butz, J. S. Oliveira, A. E. Santos, A. L. Teixeira, P. Poupart, and A. Kalra, "An empirical study of methods for SPN learning and inference," in *International Conference on Probabilistic Graphical Models*, 2018, pp. 49–60.

[36] H. Poon and P. Domingos, "Sum-product networks: A new deep architecture," in *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence*, 2011.

[37] R. Sánchez-Cauce, I. París, and F. J. Díez, "Sum-product networks: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

[38] Y. Zhang and C. Dong, "MARS: Detecting brain class/method code smell based on metric–attention mechanism and residual network," 2024.

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

[40] J. Wang, J. Chen, and J. Gao, "ECC multi-label code smell detection method based on ranking loss," *Journal of Computer Research and Development*, vol. 58, no. 1, pp. 178–188, 2021.

[41] A. Abuhassan, M. Alshayeb, and L. Ghouti, "Software smell detection techniques: A systematic literature review," *Journal of Software Evolution and Process*, 2020.

[42] 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*, vol. 255, p. 109737, 2022.

[43] J. Han, J. Pei, and M. Kamber, *Data Mining: Concepts and Techniques*. Elsevier, 2011.

[44] A. Mostefai, "Using sum product networks to predict defects in software systems," *International Journal of Information Technology*, 2024.

[45] R. Gupta, N. Kumar, S. Kumar, and J. Kumar Seth, "Unsupervised Machine Learning for Effective Code Smell Detection: A Novel Method," *Journal of Communications Software and Systems*, vol. 20, no. 4, pp. 307–316, 2024.

[46] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," *Journal of Systems and Software*, vol. 167, p. 110610, 2020.

[47] M. Jerzyk and L. Madeyski, "Code Smells: A comprehensive online catalog and taxonomy," in *Developments in Information and Knowledge Management Systems for Business Applications*, vol. 7, Springer, 2023, pp. 543–576.

[48] R. Gens and P. Domingos, "Learning the structure of sum product networks," in *Proceedings of the Thirtieth International Conference on Machine Learning*, 2013.

[49] A. Molina, A. Vergari, K. Stelzner, R. Peharz, P. Subramani, N. Di Mauro, P. Poupart, and K. Kersting, "Spflow: An easy and extensible library for deep probabilistic learning using sum-product networks," 2019.

[50] R. Peharz, R. Gens, F. Pernkopf, and P. Domingos, "On the latent variable interpretation in sum–product networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, pp. 2030–2044, 2016.

Mostefai Abdelkader was born in 1978 in Saida, Algeria. He received his engineering degree from Djilali Liabess University, Algeria, in 2000, followed by a Master's degree in 2008, and a Ph.D. degree in 2014, both from the same institution. He is currently a professor at Dr. Tahar Moulay University of Saida, Algeria. His research interests include software engineering, artificial intelligence, and machine learning.