Features and Supervised Machine Learning Based Method for Singleton Design Pattern Variants Detection

Abir Nacef[1], Sahbi Bahroun[2], Adel Khalfallah[1] and Samir Ben Ahmed[1]
[1]*Faculty of Mathematical, Physical and Natural Sciences of Tunis (FST), Computer Laboratory for Industrial Systems, Tunis El Manar University, Tunisia*
[2]*Higher Institute of Computer Science (ISI), Limtic Laboratory, Tunis El Manar University, Tunisia*

Keywords: Design Patterns, Singleton Variant, RNN-LSTM Classifier, Machine Learning, SD Classifier.

Abstract: Design patterns codify standard solutions to common problems in software design and architecture. Given their importance in improving software quality and facilitating code reuse, many types of research are proposed on their automatic detection. In this paper, we focus on singleton pattern recovery by proposing a method that can identify orthodox implementations and non-standard variants. The recovery process is based on specific data created using a set of relevant features. These features are specific information defining each variant which is extracted from the Java program by syntactical and semantic analysis. We are based on the singleton analysis and different proposed features in ou previous work (Nacef et al., 2022) to create structured data. This data contains a combination of feature values defining each singleton variant to train a supervised Machine Learning (ML) algorithm. The goal is not limited to detecting the singleton pattern but also the specification of the implemented variant as so as the incoherent structure that inhib the pattern intent. We use different ML algorithms to create the Singleton Detector (**SD**) and compare their performance. The empirical results demonstrate that our method based on features and supervised ML, can identify any singleton implementation with the specific variant's name achieving 99% of precision, and recall. We have compared the proposed approach to similar studies namely **DPDf** and **GEML**. The results show that the **SD** outperforms the state-of-the-art approaches by more than 20% on evaluated data constructed from different repositories; **PMART**, **DPB** and **DPDf** corpus in terms of precision.

1 INTRODUCTION

Design patterns (Gamma et al., 1994) have an important role in the software development process. The term has become commonplace among software designers, requirements engineers, and software programmers alike. The language of design patterns is now necessary to understand and work with software because it helps to understand the design intent of pre-developed software. Hence, software reverse engineering and redesign. His recovery can make the maintenance of source code, more easy and enhance her existing analysis tools by bringing program understanding to the design level. Regarding its important role in improving program comprehension and re-engineering, design pattern detection became a more and more active research field and has observed in recent years, a continual improvement in the field of automatic detection. However, there are many difficulties in detecting practical design patterns that arise from the following:

- The non-formalization of the pattern, the variety in implementations of source codes, and the increasing complexity of software projects
- The design structure does not match the intent: Even though the identified pattern instances correspond to their structure, the design intent can be broken. Which can be considered as the main reason behind the increase in the false positive rate.

Most existing methods convert source code and design patterns into intermediate representations, such as rules, models, graphs, products, and languages. Using these intermediate representations makes it easier to extract structural elements such as classes, properties, methods, etc. from the source code. However, they show poor performance compared to methods based on extracted features. At the same time, structural analysis cannot recover the pattern intent of its different representations, so it is necessary to perform semantic analysis on the source code to improve

Nacef, A., Bahroun, S., Khalfallah, A. and Ben Ahmed, S.
Features and Supervised Machine Learning Based Method for Singleton Design Pattern Variants Detection.
DOI: 10.5220/0011992100003464
In *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2023)*, pages 226-237
ISBN: 978-989-758-647-7; ISSN: 2184-4895

the accuracy of the recovered model. On the other hand, extracting semantic information from source code is still a difficult task due to the complexity and diverse representation of the code. Therefore, to solve this problem, we are based on our previous work (Nacef et al., 2022) to define singleton variants rules and create specific data to recover each variant from the source code. We have proposed a set of features for identifying singleton patterns based on a detailed analysis of the variant's structure and behaviors. For analyzing the Java program, we apply syntactical and semantic analysis by the use of **LSTM** model to extract semantic information (features). The **LSTM** model is trained by specifically structured data corresponding to each feature for a classification task.

The main goal of our work is the detection of the singleton design pattern. The **SD** role has not been limited only to the recovery of singleton variants (including non-standard variants and their combinations), but also incoherent implementations with the singleton intent in order to verify that the singleton pattern has been implemented correctly. While designers understand patterns well, developers may not have as much experience. This can lead to incorrect implementation of the pattern or the possibility of later introducing coding errors that break the pattern stage. The recovery of incorrect implementation makes possible the refactoring task. If a singleton pattern is detected, the **SD** can specify the type of the implemented variant.

In this paper, we propose a Feature-Based Singleton Design Pattern Detection approach that uses a set of features extracted from both syntactical and semantic analyses. We create specific structured data based on the feature's combination values corresponding to each singleton variant to train a supervised ML model named **SD**. Trying to construct data containing various implementations, we ameliorate the learning process of the **SD** to recover any implementation of the singleton pattern. The proposed approach is based on the detailed analysis of the pattern previously realized and selected features in (Nacef et al., 2022) to extract rules for pattern identification in the goal to create the data. This dataset serves as training data to make learning the **SD** model. Then we evaluate the proposed approach with a labeled dataset collected from **PMART** (Guéhéneuc, 2007), (Guéhéneuc, 2007), **DPB** (Fontana et al., 2012), **DPDf-corpus** (Nazar et al., 2022) and 94 Java files extracted from a publicly available GitHub Java Corpus. The detector makes a very good result, with higher accuracy compared to the state-of-the-art approaches.

The Contribution of the paper can be summarized as follows:

- We introduce a novel approach called "Singleton Design Pattern Variant Detection using features and Supervised Machine Learning" (**SD**) that use 33 features to recover Singleton pattern variants.
- We create a structured data named **DTSD** for training the **SD** classifier. the **DTSD** contains 7000 samples (combination of features values).
- We build two labeled data for evaluating the **SD**; the first is extracted from **DPDf-Corpus** (we take only files with singleton implementation) named **DPDf 2** and we insert the missing variants. The second is building from the singleton pattern file existing in **P-MART**, **DPB** and **DPDf** Corpus.
- We proved that our approach outperforms similar existing approaches with a substantial margin in terms of standard measures.

The rest of the paper is organized as follows; we present the related work and the contribution made to them in Section2. We indicate the reported singleton variants and the highlighted features in Section3. In section 4, we discuss the relevant background of the related technologies and we present the proposed approach. Section 5 presents obtained results. An evaluation of different **ML** algorithms and a comparison between our study and the state-of-the-art are realized and discussed. Finally, in Section 6 a conclusion and future work is presented.

2 RELATED WORK

In recent years, design pattern detection became a more and more active research domain. The problem of recovering design patterns from the source code has been faced and discussed in several works. Many strategies and many techniques are used.

The majority of design patterns mining approaches transform the source code and design patterns into some intermediate representations such as an abstract semantic graph, abstract syntax tree, rules, grammar, etc.... The searching methods diversify also from one to another, and can be classified as metrics, constraint resolver, database queries, eXtended Positional Grammar (XPG), etc....

Several approaches (Wegrzynowicz and Stencel, 2013),(Combemale et al., 2021), (Ahram, 2021) use database queries as a technique for extracting patterns. They use Structured Query Language (SQL) queries to extract pattern-related information, and produce an intermediate representation of the source code. In this case, the performances depend enormously on the underlying database and can be scaled

very well. However, queries are limited to the available information existing in the intermediate representations.

Techniques proposed by (von Detten and Becker, 2011), (Kim and Boldyreff, 2000) use program-related metrics (e.g., aggregations, generalizations, associations, and interface hierarchies) from different source code representations. The detection of DP is based on comparing DP and code source metrics values. This method is computationally efficient because it reduces the search space through filtration (Guéhéneuc et al., 2010). An advanced step proposed by (Satoru Uchiyama,) combine software metrics and machine learning to identify candidates for the roles that compose.

Other techniques are based on graph representation. The detection approach proposed by (Balanyi and Ferenc, 2003) combines graph and software metrics to perform the recognition process. First, a set of candidate classes for each DP role are identified based on software metrics. These metrics were chosen based on the theoretical description of the DP, and they were used to establish clear logical rules that could lead to many false positives. In the second stage, all candidate class combinations are analyzed in detail to find DP matches. To improve the accuracy of results, ML methods (decision trees and artificial neural networks) are used to filter as many as possible false and distinguish similar patterns (Ferenc et al., 2005). Other work proposed by (Zanoni et al., 2015) develop a **MARPLE** tool (Fontana and Zanoni, 2011) which exploited a combination of graph matching and ML techniques.

Influenced by the work given by (Tsantalis et al., 2006), (Thaller et al., 2019) propose a feature Maps for pattern instances based neural networks. (Chihada et al., 2015) propose a design pattern detector that learns based on the information extracted from each pattern instance. They treat the design pattern recognition problem as a learning problem. However, recent work proposed by (Hussain et al., 2018) treats the problem as text categorization, in which they leverage deep learning algorithms for organizing and selecting DPs. Recently, (Nazar et al., 2022) selected 15 feature codes and use machine learning classifiers to automatically train a design pattern detector. Another recent work proposed in (Barbudo et al., 2021) present a novel machine learning-based approach for DPD named GEML. Like other work, the used method explores the ML capacity, but (Barbudo et al., 2021) addressed their limitations by using G3P as a basis of the proposed approach.

Though we base on ML to extract information from source code and to detect singleton variant, our approach differs from the other mentioned approaches in many ways. The difference that can exist is summarized as under.

- (Stencel and Wegrzynowicz,) and other work, detect many variants of singleton pattern. However, we are the first to detect non-only the different variants, but its possible combination and incoherent implementations that inhibit the pattern intent, with their corresponding names.
- Our approach use ML like many other approaches, but it is the first to create singleton specific dataset for training the model (whether at the level of code analysis or pattern extraction), which performs the model to better training, i.e. better results, and make easy to filter false positive.
- As (Hussain et al., 2018), we use deep learning algorithms for text categorization, but extracting a pattern from direct source code is a very hard task and needs an enormous number of data because there are many non-standard implementations. In our work, we consider text categorization as the first step in which we extract needed information from the source code. This information represents features that describe the pattern structure and behavior. The use of features reduces the search space, and the size of training data increases the prediction rate and decreases the false positive number.
- (Nazar et al., 2022) use 15 source code features to identify 12 Design Patterns, (Fontana and Zanoni, 2011) utilize code metrics and (Thaller et al., 2019) use feature maps. However, we employ 33 features, especially for the singleton pattern. The use of specific features gives a detailed definition of the pattern and allows the identification of non-standard implementations.
- The data dedicated to the SD has a size of 7000. However, **P-MART** corpus includes 1039 files, and **DPDf** uses a corpus with 1300 files for training the classifier to recover many design patterns with imbalanced nature. The Corpus Used in both is used for training and evaluating the model. The number of singleton patterns existing in **P-MART** and **DPDf** corpus is respectively 12, and 100 which is not entirely satisfactory to recover all implementations.
- Our SD based on extracted features from structural and semantic analysis of source code and **ML** techniques achieved approximately 98% of precision and recall and prove its capability to recover any non-standard variant and filter false positives. However, **DPDf** and **GEML** did not

exceed 80% in terms of precision, recall and F1 Score.

3 REPORTED VARIANTS AND PROPOSED FEATURES

The singleton design pattern is used to ensure that a class has only one instance. That means restricting class instantiation to a single object (or even to a few objects only) in a system and providing a global access point to it. The recognition of instances of singleton patterns in source code is difficult, caused by the different implementations, which are also not formally defined. So that we depart from the singleton pattern analysis and their variants presented in (Nacef et al., 2022).

3.1 Reported Variants

Based on the proposed singleton variants defined in (Gamma et al., 1994), (Stencel and Wegrzynowicz,), our approach can effectively recognize the different variants represented in table 1 and their combinations form.

Table 1: Reported Singleton variants.

Singleton variants	
Eager Instantiation	Lazy Instantiation
Placeholder	Replaceable Instance
Subclassed Singleton	Delegated Construction
Different Access Point	Limiton
Social Singleton	Generic Singleton

3.2 Selected Features

For the singleton pattern detection, we have to use the 33 features proposed by (Nacef et al., 2022) resulting from the singleton variants analysis.
This specific analysis allowed the extraction of the essential information for each variant signature. These features are presented in table 2.

If we just look at its canonical implementation (which is quite simple), the intent of the singleton pattern seems simple. However, careful analysis of the structure may lead to further constraints being defined. These characteristics reflect information that has forty effects on preserving the singleton intent. The only way to keep a singleton instance for future reuse is to store it as a global static variable. With this, we should check the correctness of the different variants and count the number of class attributes and method-generating instances. If the number exceeds one, then the structure is incorrect and the intention is inhibited (case of listing 1 and 2.

Table 2: Used Features.

Abb.	**Feature**
IRE	Inheritance relationship (extends)
IRI	Inheritance relationship (implements)
CA	Class accessibility (public, abstract, final)
GOD	Global class attribute declaration
AA	Class attribute accessibility
SR	Static class attribute
ON	Have only one class attribute
COA	Constructor accessibility
HC	Hidden Constructor
ILC	Instantiate when loading class
GAM	Global accessor method
PSI	Public Static accessor method
GSM	Global setter method
PST	Public static setter method
INC	Use of inner class
EC	Use External Class
RINIC	Returning instance by the inner class
RINEC	Returning instance by the External class
CS	Control instantiation
HGM	Have one method to generate instance
DC	Double check locking
RR	Return reference of the Singleton instance
CNI	Variable to count the number of instances
CII	Create internal static read-only instance
DM	Use delegated method
GMS	Global accessor synchronized method
IGO	Initializing global class attribute
LNI	Limit the number of instances
SCI	Use string to create instance
SB	Static Block
AFL	Allowed Friend List
CAFB	Control access to friend behavior
UR	Using Singleton class as a type for generic instantiation

The common conditions that must be verified in most singleton variants are:

- The global access point to get the instance.
- The Constructor modifier to restrict its accessibility.
- The Control of instantiation; verifying the existence of conditions that limit the number of created instances.
- The verification of the number of declared class attributes and the number of method-creating instances to only one.

Second, specific information for each variant should be verified. In table 3, we categorize features with corresponding variants. Relevant information needed for the identification of each variant is regrouped. This grouping strongly helps SD data creation.

Table 3: Features corresponding to each variant.

Singleton Variants	Features
Eager Implementation	ILC, SB
Lazy Instantiation	GAM, PSI, CS, HC, DC, GMS
Different Placeholder	INC, RINIC
Replaceable Instance	PST, GSM
Subclassed Singleton	CA, IRE
Different Access Point	EC, RINEC
Delegated Construction	DM
Limiton	LNI
Social Singleton	IRI, AFL, CAFB
Generic Singleton	TUR

4 PROPOSED APPROACH

In this section, we will represent the used techniques and discuss the singleton detection process. Finally, we are gone to give details about the created data used for the training phase.

4.1 ML Used Techniques

ML is a subset of artificial intelligence (**AI**) that focuses on creating systems that can learn from data, identify patterns and make decisions with minimal human intervention. The power of ML is its capability to automatically improve performance through experience. ML have penetrated every aspect of our lives, and made the hard task easier to resolve, with higher accuracy.

Algorithms are the engines of ML. In general, two main types of ML algorithms are used today: supervised learning and unsupervised learning. The difference between the two is defined by the method used to process the data to make results, the type of input and output data, and the task that they intend to solve.

In our work, we use the supervised learning type; which is supplied with information about several entities whose class membership is known and which produce from this a characterization of each class. In supervised learning, there are two major types, named regression and classification. In our work, tasks realized have a classification type. The first step realized (Nacef et al., 2022) consist of analyzing Java program by the use of (**RNN-LSTM**) (Sherstinsky, 2018) to extract the value of features with binary or multi-class classification. In the second phase, we use the previous analysis of the singleton pattern and feature's value to create labeled structured data for training the **SD** classifier. The **SD** carries out a multi-class classification task; each class represents a singleton variant (if a singleton pattern is implemented) or none otherwise. For the **SD**, we built various **ML** classifiers such as **Random Forest** (RF) (Elmahdy et al., 2021), **Gradient Boosted Tree** (GBT) (Murphy, 2012), **SVM** (M.Schnyer, 2020), **KNN** (Peterson, 2009), and **Neural Network** (NN) (Doya and Wang, 2022) intending to compare their results.

Figure 1 illustrates the proposed approach process. In the following subsections, we will discuss these phases one by one.

4.2 First Phase

The detailed analyses of singleton variants make easier the defining of efficient information and the construction of the training dataset. In (Nacef et al., 2022) a detailed analysis of the singleton pattern is realized, a set of variants are identified, and a 33 relevant features are selected. For extracting feature values from the Java program, a syntactical and semantic analysis is applied by the use of **LSTM**.Based on the analysis and the feature's value extracted from the Java program, we define rules for every singleton variant, and then we construct the dataset for training the **SD**.

4.3 Second Phase

The analysis already done was very useful in terms of defining the rules for each implementation variant, which will be the key behind the construction of the training data **DTSD**. The strength of our approach is that we use specific features, as well as the use of our own created data.

- **Dataset Creation Process.** As we know, the data is the essence of ML, then the more large and more diverse the dataset is, the better results will be obtained. If the classifier is trained by completed and diversified examples, it will be more able to predict correct instances, and achieve a higher accuracy result. Based on the important role of the data in building a performed model, we gave importance to the creation of the training dataset. Contrarily to other approaches, we don't limit ourselves to the existing implementation extracted from the considerate benchmark corpus, because many implementations can be missed, or we can have an imbalance in their numbers (dominance of some implementations over others)

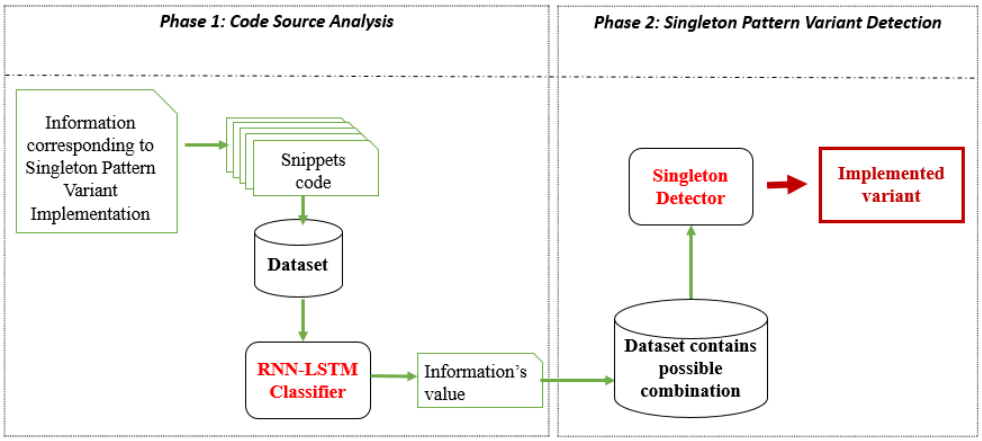

Figure 1: Flowchart of the proposed approach.

which causes an unfair training process and can lead to the incapability of the model to recover all existing implementations. So for training the model, we create a dataset **DTSD** based on rules extracted from singleton variants analyses, and for evaluating the model, we use other created data based on **DPDf** singleton corpus.

- **Defining Rules.** We have identified 27 instances according to different singleton variants. The class candidates are represented in table 4. Based on specific information about each one, we illustrate rules to define them. The rules represent a combination of values of features. The **SD** will be made by learning from information extracted from the **DTSD** data. Table 5 shows an example of important feature values that made each candidate instance true. Noting that, based on these most important features, we can create several implementations by playing on the other features values (we must respect the identity of each variant).

  To achieve high recall (which will be explained in the next section), we need to reduce the number of false-negative predictions. For a singleton design pattern detection, this leads to creating a dataset that comports numerous implementation variants that preserve the meaning of the pattern, and those that can destroy the intent. As an example, the only way to keep a singleton instance for future reuse is to store it as a global static variable, so that we should verify that there is only one declared class attribute in different variants (**ON**). In the case of different placeholder and different access point variants, the instance is held as a static attribute of an inner class or external class, so we should verify the existence of a static class attribute inside both. Another example, as the intent of a singleton pattern is to limit the number of objects to only one (exception Limiton variant), we must verify that there is only one block for creating an Instance (**HGM**). The fact that one of the two features (**ON/HGM**) is false, the implementation will be considered an incorrect singleton structure (as shown in table 11). This error inhibits the singleton intent and can provoke false-negative predictions. This type of error can be caused by a developer's unconsciousness during the implementation, so we decided to consider it, and detect similar implementations if they exist. Listing 1 and 2 represents an example of incorrect lazy and eager implementation, which inhibits the singleton pattern intent. Table 5,6,7,8,9,10 present example of rules defining some singleton variants and table 11 represents an example of a combination of feature values making the implementation error.

Listing 1: Example 1: Singleton Error Implementations.

```
Public class C1 {
  Private static C1 instance = new C1
      ();
  C1 () {}
  public static C1 getInstance () {
    Return (new C1());}
}
```

Listing 2: Example 2: Singleton Error Implementations.

```
Public class C2 {
    Private static C2 instance;
    Private C2 ( ) {}
    Public static C2 getInstance1 ( ) {
      if (instance = = null) instance =
          new C2;
 }
   Public static C2 getInstance2 ()
     {return new C2;}
}
```

Table 4: SD Classifier classes.

Singleton Variants	Classes	NO.
Eager Implementation	Simple Implementation	1
	Implementation With static Block	2
	Simple Implementation-Invalid Implementation	3
	Implementation With static Block-Invalid Implementation	4
Lazy Instantiation	Singleton naif	5
	Non-thread safe	6
	Thread safe with synchronized method	7
	Lazy instantiation double lock mechanism	8
	Singleton naif-Invalid Implementation	9
	Non thread safe-Invalid Implementation	10
	Thread safe with synchronized method-Invalid Implementation	11
	Lazy instantiation double lock mechanism-Invalid Implementation	12
Different Placeholder	Different Placeholder	13
	Different Placeholder-Invalid Implementation	14
Replaceable Instance	Replaceable Instance	15
	Replaceable Instance-Invalid Implementation	16
Subclassed Singleton	Subclassed Singleton	17
	Subclassed Singleton-Invalid Implementation	18
Different Access Point	Different Access Point	19
	Different Access Point -Invalid Implementation	20
Delegated Construction	Delegated Construction	21
	Delegated Construction -Invalid Implementation	22
Limiton	Limiton	23
	Limiton-Invalid Implementation	24
Social Singleton	Social Singleton	25
Generic Singleton	Generic Singleton	26
No Singleton	No Singleton	27

Table 5: Data extract: Example of features combination for Eager Singleton variant.

Class	Combination Values											
	CA	GOD	AA	SR	COA	ILC	GAM	PSI	RR	SB	ON	HGM
1	public	True	Private	True	private	True	True	True	True	False	True	True
2	Public	True	Private	True	private	True	True	True	True	True	True	True

Table 6: Data extract: Example of features combination for Lazy Instantiation Singleton variant.

Class	Combination Values													
	CA	GOD	AA	SR	COA	ILC	GAM	PSI	RR	CS	DC	GMS	ON	HGM
6	Public	True	Private	True	private	False	True	True	True	True	False	False	True	True
7	Public	True	Private	True	Private	False	True	True	True	True	False	True	True	True

Table 7: Data extract: Example of features combination for Different Placeholder Singleton variant.

Class	Combination Values									
	CA	COA	ILC	GAM	PSI	RR	INC	RINIC	ON	HGM
13	public	Private	False	True	True	True	True	True	True	True

Table 8: Data extract: Example of features combination for Replaceable Instance Singleton variant.

Class	Combination Values											
	CA	GAD	AA	SR	COA	GAM	PSI	RR	GSM	PSS	ON	HGM
15	public	True	Private	True	Private	True	True	True	True	True	True	True

Table 9: Data extract: Example of features combination for Delegated Construction Singleton variant.

Class	Combination Values										
	CA	GOD	AA	COA	CS	GAM	PSI	RR	DM	ON	HGM
21	public	True	Private	Private	True	True	True	True	True	True	True

Table 10: Data extract: Example of features combination for Social Singleton variant.

Class	Combination Values				
	CA	COA	IRI	AFL	CAFB
25	Public	Private	True	True	True

Table 11: Data extract: Example of features combination for Eager Invalid Implementation Singleton variant.

Class	Combination Values											
	CA	GOD	AA	SR	COA	ILC	GAM	PSI	RR	SB	ON	HGM
3	public	True	private	True	private	True	True	True	True	False	False	False
4	public	True	private	True	private	True	True	True	True	True	False	False

- **Building Singleton Variants Classifier.** The use of an ML algorithm depends on the type of data to treat and generate, and the type of realized task. A singleton design pattern detection is a classification problem, with structured labeled data. To deal with this typical problem, we can use different algorithms. In this case, the better model to use is the one that gives a better result. We choose to use five different algorithms the most already used in classification tasks; **RF**, **GBT**, **SVM**, **KNN**, and **NN**.

5 EVALUATION SETUP

This section presents the criteria used to evaluate our **SD** and the different results made by it. We compared results generated from each model, and interpret and compared them with state-of-the-art approaches.

5.1 Evaluation Protocol

The standard measures to statistically evaluate the efficacy of classifiers are Precision, Recall, and the F1-Score. Prediction: The prediction rate indicates the fraction of positive prediction which was actually correct. It is defined in 1:

$$Precision = \frac{TP}{TP+FP} \quad (1)$$

Recall: The recall indicates the fraction of actual positives which were identified correctly. It is defined in 2:

$$Recall = \frac{TP}{TP+FN} \quad (2)$$

The F1 score is a way to combine both precision and recall into a single number. It represents a harmonic mean of both scores, which is given by this simple formula 3:

$$F1Score = \frac{2*(precision*recall)}{(precision+recall)} \quad (3)$$

5.2 Performed Results

We have created a classifier based on different **ML** models. After training the model we have tested it with the 200 GitHub Java classes referred by **DPDf** Corpus proposed by (Nazar et al., 2022) 100 files contain singleton implementation and 100 files do not contain any type of design patterns i.e. none.

Unfortunately, the corpora used do not contain all singleton variants, which makes the evaluation of the **SD** not completed. To ensure the performance of the detector in recognizing each variant; we collect other GitHub Java classes with missing variants. Next, we took these collected classes and injected blocs that inhibit the pattern intention, in order to verify the ability of the **SD** classifier to recognize also the incorrect implementations.

After collecting the evaluation set, we labeled them with the corresponding singleton variant name. In the first step, we analyze the existing java class with the **LSTM** classifier to determine every feature's values. In the second step, we evaluate the **SD** classifier by the use of the resulting dataset containing the feature's values. The corresponding results of the **SD** classifier are illustrated in table 12.

All used **ML** algorithms make very good results thanks to the use of specific training datasets. Comparing the results of each **ML** algorithm used in the **SD** classifier, we can see that all used algorithms are performed and make closed results. **SVM** model has performed excellent results in terms of precision, recall, and F1-score, and can correctly identify any Sin-

Table 12: SD Classifier results.

SD Classifiers	Measures (%)		
	Precision	Rappel	F1
RF	96.24	96.47	95.81
GBT	98.85	98.65	98.64
KNN	92.3	87.05	86.55
SVM	**99.49**	**99.42**	**99.41**
NN	98.3	98.7	98.49

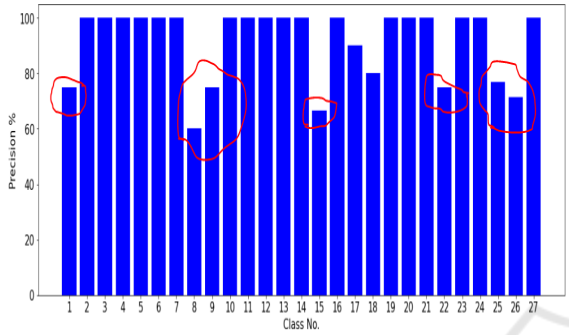

Figure 2: KNN precision results.

gleton candidate with 100% of precision as shown in figure 3 (except one candidate 87%). However, the **KNN** algorithm is the classifier that makes fewer results. Contrary to **SVM**, **KNN** fails to detect correctly some Variant like Lazy instantiation with double lock mechanism and Replaceable Instance with less than 70% of precision like showing in figure 2.

5.3 Comparison with Similar Existing Approaches

In this work, we propose a machine learning-based method to recover the singleton pattern. To position our work against state-of-the-art studies, we select two recent relevant approaches working also with Machine Learning. The first is proposed by (Barbudo et al., 2021) named **GEML** and the second is proposed by (Nazar et al., 2022) named **DPDf**.

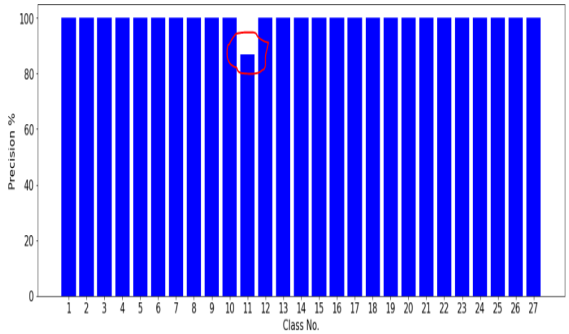

Figure 3: SVM precision results.

5.3.1 The Benchmark DP Detection Approaches

- **GEML Approach**

  GEML is a novel DPD approach based on ML and grammar-guided genetic programming (G3P). By the use of software properties, GEML extracts DP characteristics formulated in terms of human-readable rules. Then a machine learning classifier is built based on the established rules to recover 5 DP roles.

  In (Barbudo et al., 2021) a comparison between GEML and other DPD methods, including both ML and non-ML are realized. GEML outperform MARPLE techniques (Zanoni et al., 2015) with highly values in terms of accuracy and f1 score. It's also more competitive than two other reference DPD tools which are frequently used for comparative purposes (MLDA, SparT and DePATOS).

  The experimentation in GEML covers 15 roles of DP and uses implementations from two repositories to compare obtained results against other works. The singleton repository details are:

  – **DPB:** created by the authors of (Fontana et al., 2012) and used to compare the results with MARPLE.
  – **JHotDraw** from PMART (Guéhéneuc, 2007) used to compare with non ML-based methods like DePATOS (Yu et al., 2018), MLDA (Al-Obeidallah et al., 2018) and SparT (Xiong and Li, 2019).

  Table 13 illustrate the comparison results of GEML against other DPD studies. The comparison is conducted based on common ground truth. Four methods are under study. The use of the "-" symbol, is to indicate that the particular DP is not supported by the corresponding approach. The first comparison is based on the DPB corpus. Both GEML and MARPLE perform good results in singleton detection, but GEML outperforms MARPLE by more than 7% and 3% of improves in terms of accuracy and F1 score. The second comparison is based on the JHotDraw project. GEML, MLDA, and SparT can recover all singleton instances. Their great performance is related to the reduced number of true Singleton instances in the test data.

- **DPDf Approach**

  The approach proposed in (Nazar et al., 2022) is the first to employ lexical-based code features and ML to recover a wide range of design patterns with higher accuracy compared to the state-of-the-art. Our approach also combines semantic anal-

Table 13: Comparing GEML with the state-of-the-art results.

ML techniques	Singleton corpus				
	DPB-Corpus		JHotDraw		
	Accuracy (%)	F1-score (%)	Precision (%)	Recall (%)	F1-score (%)
MARPLE	88	91	-	-	-
GEML	95.61	94.11	100	100	100
DePATOS	-	-	-	-	-
MLDA	-	-	100	100	100
SparT	-	-	100	100	100

ysis, features, and ML algorithms as techniques, but we have focused only on Singleton Variants recovering. Based on the relevancy measure, we chose to compare our study with the study proposed in (Nazar et al., 2022).

DPDf. Developed in (Nazar et al., 2022) generates a Software Syntactic and Lexical Representation (**SSLR**) by building a call graph and extracting 15 source code features. The **SSLR** is used as an input to build a word-space geometrical model by applying the Word2Vec algorithm. Then, a **DPDf** ML classifier is created and trained by a labeled dataset and geometrical model.

DPDf Reported Results. Nazar et al. has compared his study with two approaches based on code features and ML, which are developed by (Thaller et al., 2019) and (Fontana and Zanoni, 2011). For comparing the results, they uses two benchmark Corpus:

- **P-MART** Corpus; containing only 12 Singleton implemented classes.
- **DPDf** Corpus; Contain 100 singleton implemented class.

The compared results, reported by (Nazar et al., 2022) in the detection of the singleton design pattern, are illustrated in table 14. **DPDf** has improved more performance in recovering singleton candidates from the **DPDf-corpus** but has fairly recovered it from the **P-MART** Corpus. These unbalanced results are caused by the number of instances of singleton existing in each corpus, which have a great impact on the learning of the classifier.

5.3.2 Comparison Against GEML and DPDf Methods

- **Comparison Strategy**

Results made by (Nazar et al., 2022) and (Barbudo et al., 2021) do not refer exactly to the performance of both approaches to recover singleton pattern instance because other patterns are included. Therefore, we have tested the **DPDf** and **GEML** with singleton specific data. The **P-MART** Corpus contains a few instances of singleton pattern, contrarily to **DPDf** and **DPB** corpus. Otherwise, they represent a none complete data; some variants are absent. Consequently, we construct new data for the evaluation; we bring together all singleton instances existing in **PMART**, **DPB** and **DPDf** repositories, and complete the data with missing variants. We also construct an incoherent structure by injecting, to correct instances, a structure that inhibits the singleton intent. We try to evaluate with complete data containing a variety of implementations. Details of constructed data are presented in table 15, with the according to the number of correct, incorrect, and incoherent samples in each repository.

- **Comparison Results**

We reproduce the public works of (Nazar et al., 2022) and (Barbudo et al., 2021) with only singleton pattern. Then, we evaluate them by the newly created data. The experiment results are conducted by the use of a RF classifier in the evaluation process. Table 16 illustrate obtained results from the experiments.

- **Discussion**

The comparative results illustrated in table 16 show the performance of the **SD** to recover any variant of the singleton pattern, whatever the implementation is. The **SD** reaches the best result (98% of F1 Score), and outperforms **GEML** and **DPDf** by respectively 21% and 24% improvements in terms of F1 Score. Both **GEML** and **DPDf** use approximately a small sample for training the model, which is not enough for the best training. The classifier in this case is not able to recover all singleton implementations. His capacity enormously depends on the variants existing in the training data, new variants which not fully trained cannot be detected. However, by the use of complete data that is well created as in the case of **SD** (7000 samples), the classifier will be better trained, and always gives a great performance in the detection process. Our **SD** has the ability not only to recognize the existence of singleton patterns, but also the type of implementation. It has also the capacity of recovering

Table 14: Comparing DPDf results with MARPLE and Feature Maps.

ML techniques	Singleton Corpus					
	DPDF Corpus			Labelled P-MART		
	Precision (%)	Recall (%)	F1-score (%)	Precision (%)	Recall (%)	F1-score (%)
Feature Maps	65	67	65.98	63	59	60.93
MARPLE	74.24	69.23	71	74.23	70.18	72.15
DPDf	81.6	68.22	**74.31**	43.33	40	**41.6**

Table 15: Evaluating Data Composition.

	Singleton Corpus			
	PMART	**DPB**	**DPDf**	**Others**
Correct	12	58	100	53
Incorrect	-	96	100	-
Incoherent	-	-	-	41
Total size	460			

Table 16: Comparing SD with DPDf and GEML

Approaches	**Measures (%)**		
	Precision	**Recall**	**F1**
DPDf	78.5	70.23	74.13
GEML	81.6	73.4	77.28
SD	**98.96**	**97.63**	**98.29**

incoherent implementations with the singleton intent.

The imbalanced results obtained from the different DPD methods are explained by the number and the variety of implementations existing in the training data. There is another factor that strongly affects the results, which is the characteristics of the source code which are highlighted. **DPDf** approach uses only 15 features to define 12 design patterns, **GEML** propose 23 Grammar operators to describe a variety of design pattern implementations. However, in our work, we use 33 features for defining only the singleton pattern. These enormous numbers of features make a detailed description of the pattern and provide all information needed to identify any variant.

6 CONCLUSION AND FUTURE WORK

We propose in this paper a new singleton design pattern approach based on features and ML techniques. We are based on previous work (Nacef et al., 2022), and different proposed features to create **DTSD** dataset. **DTSD** is used to train the **SD** classifier, in which we try to make rules defining a various number of singleton implementations. In the dataset, we give a combination of feature values to categorize each variant. We have tried to be sure that the data contain the most number of implementations to perfectly train the model. Next, we build a Singleton Detector based on different ML classifiers. We trained the supervised classifier by the labeled dataset **DTSD**, and we evaluate its performance with other data collected from the GitHub Java corpus, and singleton variants existing in **DPDf**, **DPB** and **P-MART** corpus.

We apply three standard statistical measures namely precision, recall, and F1-Score to evaluate the performance of the created classifiers. A comparison between different used ML algorithms to create the **SD** is realized. The different results show that the use of the created dataset makes any classifier easily able to recover any variants of the singleton pattern although there is a slight difference in performance. The Empirical result shows that our proposed approach can recover any non-standard singleton variant, even incorrect implementation destroyed the singleton intent with approximately 99% on both precision and recall. Our approach outperforms recent relevance approaches by more than 20% improvements in terms of standard measures.

While our classifier's performance is promising, as furfur work we gonna applies it to recover a wide range of software design patterns. The choice of source code as refactoring support is very important and interesting, but we should not limit ourselves only to this support, we try to switch to the model as refactoring support.

REFERENCES

Ahram, T. Z., editor (2021). *Advances in Artificial Intelligence, Software and Systems Engineering*, volume 1213 of *Advances in Intelligent Systems and Computing*. Springer.

Al-Obeidallah, M., Petridis, M., and Kapetanakis, S. (2018). A multiple phases approach for design patterns recovery based on structural and method signature features. *Int. J. Softw. Innov.*, 6(3):36–52.

Balanyi, Z. and Ferenc, R. (2003). Mining design patterns from C++ source code. In *19th International Conference on Software Maintenance (ICSM 2003)*, pages 305–314. IEEE Computer Society.

Barbudo, R., Ramírez, A., Servant, F., and Romero, J. R. (2021). GEML: A grammar-based evolutionary ma-

chine learning approach for design-pattern detection. *J. Syst. Softw.*, 175:110919.

Chihada, A., Jalili, S., Hasheminejad, S. M. H., and Zangooei, M. H. (2015). Source code and design conformance, design pattern detection from source code by classification approach. *Appl. Soft Comput.*, 26:357–367.

Combemale, B., Kienzle, J., Mussbacher, G., Ali, H., Amyot, D., Bagherzadeh, M., Batot, E., Bencomo, N., Benni, B., Bruel, J., Cabot, J., Cheng, B. H. C., Collet, P., Engels, G., Heinrich, R., Jézéquel, J., Koziolek, A., Mosser, S., Reussner, R. H., Sahraoui, H. A., Saini, R., Sallou, J., Stinckwich, S., Syriani, E., and Wimmer, M. (2021). A hitchhiker's guide to model-driven engineering for data-centric systems. *IEEE Softw.*, 38(4):71–84.

Doya, K. and Wang, D. (2022). Announcement of the neural networks best paper award. *Neural Networks*, 145:xix.

Elmahdy, S., Ali, T., and Mohamed, M. (2021). Regional mapping of groundwater potential in ar rub al khali, arabian peninsula using the classification and regression trees model. *Remote. Sens.*, 13(12):2300.

Ferenc, R., Beszédes, Á., Fülöp, L. J., and Lele, J. (2005). Design pattern mining enhanced by machine learning. In *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*, pages 295–304. IEEE Computer Society.

Fontana, F. A., Caracciolo, A., and Zanoni, M. (2012). DPB: A benchmark for design pattern detection tools. In *16th European Conference on Software Maintenance and Reengineering*, pages 235–244. IEEE Computer Society.

Fontana, F. A. and Zanoni, M. (2011). A tool for design pattern detection and software architecture reconstruction. *Inf. Sci.*, 181(7):1306–1324.

Yann-Gaël Guéhéneuc P-MARt: Pattern-like Micro Architecture Repository.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition.

Guéhéneuc, Y., Guyomarc'h, J., and Sahraoui, H. A. (2010). Improving design-pattern identification: a new approach and an exploratory study. *Softw. Qual. J.*, 18(1):145–174.

Guéhéneuc, Y.-G. (2007). P-mart : Pattern-like micro architecture repository.

Hussain, S., Keung, J., Khan, A. A., Ahmad, A., Cuomo, S., Piccialli, F., Jeon, G., and Akhunzada, A. (2018). Implications of deep learning for the automation of design patterns organization. *J. Parallel Distributed Comput.*, 117:256–266.

Kim, H. and Boldyreff, C. (2000). A method to recover design patterns using software product metrics. In *Software Reuse: Advances in Software Reusability, 6th International Conerence*, volume 1844 of *Lecture Notes in Computer Science*, pages 318–335. Springer.

M.Schnyer, D. A. (2020). Support vector machine. In *Machine Learning*, pages 101–121. ScienceDirect.

Murphy, K. P. (2012). *Machine learning - a probabilistic perspective*. Adaptive computation and machine learning series. MIT Press.

Nacef, A., Khalfallah, A., Bahroun, S., and Ben Ahmed, S. (2022). Defining and extracting singleton design pattern information from object-oriented software program. In *Advances in Computational Collective Intelligence*, pages 713–726, Cham. Springer International Publishing.

Nazar, N., Aleti, A., and Zheng, Y. (2022). Feature-based software design pattern detection. *J. Syst. Softw.*, 185:111179.

Peterson, L. E. (2009). K-nearest neighbor.

Satoru Uchiyama, Atsuto Kubo, H. W. Y. F. Detecting design patterns in object-oriented program source code by using metrics and machine learning. Proceedings of the 5th International Workshop on Software Quality and Maintainability.

Sherstinsky, A. (2018). Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *CoRR*, abs/1808.03314.

Stencel, K. and Wegrzynowicz, P. Implementation variants of the singleton design pattern. In *On the Move to Meaningful Internet Systems, series = Lecture Notes in Computer Science, volume = 5333, pages = 396–406, publisher = Springer, year = 2008*.

Thaller, H., Linsbauer, L., and Egyed, A. (2019). Feature maps: A comprehensible software representation for design pattern detection. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 207–217.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., and Halkidis, S. T. (2006). Design pattern detection using similarity scoring. *IEEE Trans. Software Eng.*, 32(11):896–909.

von Detten, M. and Becker, S. (2011). Combining clustering and pattern detection for the reengineering of component-based software systems. In *7th International Conference on the Quality of Software Architectures*, pages 23–32. ACM.

Wegrzynowicz, P. and Stencel, K. (2013). Relaxing queries to detect variants of design patterns. In *Proceedings of the 2013 Federated Conference on Computer Science and Information Systems, Kraków, Poland, September 8-11, 2013*, pages 1559–1566.

Xiong, R. and Li, B. (2019). Accurate design pattern detection based on idiomatic implementation matching in java language context. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 163–174. IEEE.

Yu, D., Zhang, P., Yang, J., Chen, Z., Liu, C., and Chen, J. (2018). Efficiently detecting structural design pattern instances based on ordered sequences. *J. Syst. Softw.*, 142:35–56.

Zanoni, M., Fontana, F. A., and Stella, F. (2015). On applying machine learning techniques for design pattern detection. *J. Syst. Softw.*, 103:102–117.