International Journal of Engineering Trends and Technology
ISSN: 2231–5381 / https://doi.org/10.14445/22315381/IJETT-V71I8P207
Volume 71 Issue 8, 76-84, August 2023

Original Article

Finding Recursive Generics in Java Source Code using Machine Learning

Neha Kumari[1], Rajeev Kumar[2]

[1,2]*School of Computer and Systems Sciences, Jawaharlal Nehru University, Delhi, India*.

[1]*Corresponding Author: nkumari.cse@gmail.com*

Received: 18 May 2023 Revised: 29 July 2023 Accepted: 05 August 2023 Published: 15 August 2023

Abstract *- Understanding a complex type structure and its use in a type-safe manner is a difficult task. The recursive generic type is a complex variant one can expect by finding the recursion. It has major significance in generic programming for solving binary method problems and mimicking self-type. However, improper use of recursive generics can cause vulnerabilities in source code. To avoid unsafe practices, a programmer must be aware of the recursive generic presence in source code. In Java generics, the type recursion can be found at a class or interface declaration. Therefore, it is appropriate to distinguish class type at declaration time itself. In this paper, we use a machine learning approach to find recursive and non-recursive generic types in Java source code. We collect data from ten contemporary Java projects and prepare a dataset with generic-specific attributes. The lesser presence of recursive generic type in Java projects causes an imbalanced dataset. Initially, the dataset results were highly imbalanced. Therefore, we resampled the dataset and used the dataset to train decision tree-based classifiers. Using standard performance metrics, we conduct a comparative analysis to find a (near-) optimal classifier among the six decision tree-based classifiers. Our analysis reasserts that the ensemble-based ``Random Forest Classifier" results best in all nine metrics.*

Keywords *- Classification, Decision Tree, F-bounded, Java Generics, Type-Safe.*

1. Introduction

Parameterization is a programming construct that represents a type in a generic manner [28]. The parameters are declared variables; when implementation is needed, it is instantiated with a specific type. However, the parameterized or generic type does not support subtyping [29]. For example, List<Integer> cannot be assigned to List< number>; instead, the Integer is a subtype of number. Moreover, Object-Oriented Programming Languages (OOPLs) support the inclusion of subtyping with parameterized type in the form of variance. The subtype inclusion enhances re-usability, but it has major concerns regarding safety due to the undecidable nature of variance [9]. In Java, bounded quantification ensures safety for subtype inclusion among generic types [1]. A bounded quantification imposes restrictions over a particular range of subtypes for a type parameter. The syntax of a bounded type parameter is as follows: the type parameter followed by the “extends” keyword and the upper bound. Here, the keyword extends indicates the subtype relation.

```
// Plain Bounded
List<T extends Number>
// Recursively Bounded (F-bound)
T extends Comparable<T>
```

```
//Binary operation using F-bounded
class Person implements Comparable
<Person>{
public int compareTo(Person person)
{} }
```

As given in example 1, the generic type List has a bounded type parameter “T” that inherits a number. Therefore, a List is constrained to have a Number or any subtypes as a parameter. However, the plain bounded constraint is insufficient for a situation where the bounded type parameter is constrained recursively. For example, the binary method operations for class objects [2]. In this, the parameter type of the generic class and the receiving class object should be the same. The recursive bounded generics are suitable to perform such binary operations. However, a safe implementation of recursive bounds is necessary; otherwise, it may result in StackOverflowError. A constraint recursive bound is proposed, which is popularly known as F-bound, and is being used in several OOPLs [3]. As shown in the sample code, the F-bounded *Comparable<T>* interface is used for ordering objects of the class that implements it. It has a binary method *compareTo()* that compares the class object with the specified object. Here, the parameter type of

a comparable interface and the receiving class object is the same (Person). The F-bounded quantification is introduced to constrain such recursive type parameters. It has a significant quantification feature that enhances generics' expressiveness. However, the use of the F-bounded generics is restricted in various contexts. These restrictions avoid unsoundness in the program code [4], [5], [6], [7]. Greenman et al. proposed a trivial approach to implementing F-bound generics safely.

The proposed method suggests a disjoint of recursive type (shape) and non-recursive type (material). To validate their claim, they analyze millions of Java codes in the form of a usage graph. The usage graph shows self-loop for class-level cycles and ignores parameter-level cycles. For example, the graph shows self-loop in *T extends Comparable<T>*, but it does not show self-loop in *T<E extends Comparable<E>>*. The self-loop at the parameter level can easily be detected by analyzing the source code. Nowadays, machine-learning approaches are being used on a large scale to analyze source code [34]. We train an ML model using our dataset and use the model to identify and classify recursive and non-recursive type signatures at the class declaration.

The machine learning (ML) approach can be a convenient and error-less process to classify recursive generic (F-bounded) and non-recursive generic at class declaration. We require a Java dataset with relevant attributes to implement an ML model. Since an ML model's accuracy depends on the data quality used for training, we collect data from workable Java projects in two popular public repositories. The dataset holds important attributes of Java generics, such as class details, parameters, inheritance relations, etc. Once the dataset is prepared, we select classification models for training. Since the decision tree-based models are simple to understand and interpret and do not depend on normalized data [8], we choose decision tree-based models for our classification problem. The classifier works in two-phase to complete classification.

The first is the learning phase, which takes examples from the dataset and generates a tree-based model. In the second phase, the generated model tests whether it classifies accurately for the given input data. This paper performs a comparative analysis among six decision tree-based classification models to find a (near-) optimal classifier. The comparative analysis uses nine performance metrics based on Accuracy, Confusion table, AUC, and Kappa statistics.

The paper is organized as follows. Section 2 discusses the fundamentals of Java generics and mainly focuses on recursive generics and problem motivation. Section 3 presents related works. Section 4 includes the proposed method implemented in this paper to distinguish recursive and non-recursive generic types. Section 5 majorly discusses the results. Finally, Section 6 concludes the work with future perspectives.

2. Background and Motivating Example

2.1. Generics in Java

The section briefs the evolution of generics and the role of recursive generics in the Java programming language. Generics were added in Java 5 to improve static type checks by replacing explicit cast with type-specific parameters.

```
// Before Generics
List a = a.add("xyz");
String b = (String)a.get(0);
// After Generics
List<String> a = a.add("xyz")
String b = a.get(0);
```

However, Java generics are restricted in various ways to avoid unsafe practices. The invariant nature is one of its limitations. The invariant Java generics do not allow subtyping among type parameters. Subtyping can be enabled with bounded constraints. The bounded quantification ensures safety by fixing the upper (extends) and lower (super) limits for parameters. This upper and lower bound is known as covariant and contravariant, respectively. In Java generics, parameter bounds are restricted in various contexts.

```
// Invariant Generics
List<Number> != List<Integer>
// Co and Contra (variant) Generics
List<? extends Number> = List<Integer>
List<? super Integer> = List<Number>
```

The above code shows that the type of extended class parameter and the receiving class instance can differ. Therefore, this declaration is invalid for classes with binary method operations. The following class declaration is valid only if the extended class parameter and the receiving class instance type are the same as the F-bounded example.

```
// Bounded Quant. with recursive bound
class P<? extends T>
extends Comparable<P<? extends T>>{}
// F-bounded Quantification
class P<T>
implements Comparable<P<T>>{}
```

Hence, the F-bounded quantification is used to constrain the recursive bound. However, the F-bounded suffers from uses limitations due to its complex structure. We discuss these limitations and their impact in the following sub-section.

2.2. Motivating Example

The recursive generic or its other name, such as self-bounding generics, self-referential or F-bounds, are the different names in different OOPLs. The one goal is to constrain parameter type to provide exact type among its own

covariant types. In Java, this recursive pattern is mainly used to solve binary methods or to mimic self-type in Fluent API [32].

The other use of recursive generic can also be found with type families [33]. Apart from the recursive generic applications, the main concern is the way it is being used in the program so that it does not result in code failure. For example, the recursive generic that contain contravariant parameter may result in an infinite loop [9]. Here we will demonstrate how the recursive generic may result in unsafe if not used carefully in Java.

Instead of the various restriction on recursive generics (F-bounded), it behaves suspiciously in a few cases due to its complex structure. Here we discuss two such unsafe cases of F-bound. The first case is about contravariant recursion in inherited class parameters; this results in StackOverflowError at runtime and the contravariant recursion in the base class type parameter; this may lead to incompatible type implementation.

```
class C<P> implements
Comparable<Comparable<? super C<P>>>{}
```

In the above code, to validate whether *C<P>* is a subtype of *Comparable<? super C<P>>* or not. The Comparable<? super *C<P>>* should be a subtype of *Comparable<Comparable<? super C<P>>>*, and this goes on repeatedly that, may lead StackOverflowError.

```
class Xyz<T extends Comparable
<? super T>> {
public int compareTo(T other) {}}
```

The above code uses the recursive interface as a class's type parameter. The Comparable interface can have 'T' or any supertypes as its parameter. However, this semantically doesn't seem right for F-bound constrain since the recursive constraint applies only if the Comparable interface type parameter and the implementing class instance are exactly the same. The second case is about Fluent API. Java does not have a self-type. The recursive generic type is used to mimic self-type. The Fluent API uses an explicit cast to ensure safety, as shown in the below example. Otherwise, the parent class object return instead of the child class. The incompatible return type may result in compile time error [31].

```
public class Parent <B extends Parent<B>> {
public final B method() {...
return (B) this;
} }
```

A programmer must be aware of the presence of recursive generics, and he should understand the uses restrictions to control any mishap. Therefore, we propose an ML-based model to identify recursive patterns from Java source code and categorize the recursive and non-recursive generic classes. We assume the following three recursive patterns of the class declaration are classified as a recursive generic type; otherwise, the classes are non-recursive.

```
1) class Enum < E extends Enum<E>>
2) class SubClass extends X <SubClass>
3) class MyClass <T extends FirstType
<T, U>, U extends SecondType <T,
U>>
```

Based on the above problem statement, we formulate the following research question.

- Using the given dataset, how efficiently can a machine learning model classify a class declaration as a recursive and non-recursive generic type?

3. Related Work

A bounded quantification is a programming approach that safeguards the inclusion of subtyping among parameterized types **[**1]. The bounded quantification ensures safety by allowing the parameter a specific range of subtypes. However, it fails when a type bound is self-type. The self-types are crucial for binary method operations such as comparison, addition, etc. To solve such recursive bounded operations, the F-bounded is introduced [3]. Instead of the advantages, the F-bounded is mainly criticized for its use in an unsound manner. Kennedy and Pierce [9] are among the first researchers to discuss the intricate combination of the wildcard and F-bounded that can lead to unbounded growth during subtype checking. The inference for recursive type found failed due to poor inference context in Java [5]. The recursive lower bound causes non-termination during type check [6]. In the past, many solutions have been proposed to overcome unsoundness among recursive types, for example, the exclusion of expansive inheritance and removing nested contravariance. However, the proposal to disjoint material (non-recursive) and shape (recursive) is practical and decidable for subtyping relation [4].

We consider the F-bounded generics as one of the essential components of the generics type system that needs to be studied broadly. Many object-oriented languages use other alternative higher-kind types in place of the F-bounded for binary operations. Type class is one of the alternatives for the F-bounded [10]. However, this is not a trivial process to switch to another type, especially for the languages like Java that strictly follow backward compatibility. The F-bounded generics work well if used in a specified manner. Therefore, the programmer must be aware of the presence of the recursive type in the source code. The machine learning approaches are popularly used to extract knowledge from

open-source projects [12], [36]. This extracted knowledge is helpful to improve programs and avoid unsafe practices [13], [14]**.** Therefore, we propose to use the ML approach to explore and understand the F-bounded. Various ML-based applications are developed using robust programming like Java [15]. Here, we use ML to learn a crucial component of the Java programming language.

In the proposed methodology, the ML model is used as a classifier. The model identifies the recursive and non-recursive patterns from the given data and classifies them accordingly. For the ML model training with relevant data, we search for several standard Java datasets that are publicly available [16], [17], [18]. Among them, most datasets either contain bug-related attributes or attributes that can help to predict the name of identifiers and methods [19], [20]. Several classification models have been proposed, and their performances have been analysed with various datasets [21]. In this paper, we developed a Java dataset with class declaration information essential for the classification model. Further, we use the dataset to train various classifiers and analyse their performances.

4. Methodology

We categorize the proposed methodology into four phases:

- Input Phase: We choose relevant attributes and collect data from the Java source code to prepare the dataset.
- Pre-processing Phase: We perform resampling over the imbalanced dataset and then encode the data into a suitable format for the machine learning model.
- Model-based Learning: We choose six decision tree classifiers for the learning and perform a comparative analysis to find the (near-) optimal classifier **[**22].
- Evaluation Phase: The comparison is based on the results we get for nine performance metrics.

4.1. Input

We need to extract class declaration components from the Java Source code to identify whether the class is a recursive generic type. These components are class names, parameter details, inheritance relations, etc. The dataset contains these class declaration components in the form of nine attributes. To get values for respective attributes, we use JavaParser API that fetches values from AST nodes. After this, we store the extracted data in a CSV file with the help of OpenCSV Library. The next step is to label the data necessary to train the supervised ML algorithms. Classifiers can classify the class declaration as recursive and non-recursive using labelled data. The type of our dataset is categorical and contains nominal and ordinal values. Later, we convert these values into numerical data using standard encoders.

4.2. Pre-Processing

As expected, we find the occurrence of the recursive class significantly lesser than the non-recursive class, resulting in an imbalanced dataset. This presence of bias in the dataset may ignore the minor class. We can observe the classification error using the Weka tool [23], [24]. Wherever we apply a classifier to this imbalanced dataset, the number of incorrectly classified instances equals the number of minor classes in the dataset. Therefore, the dataset needs to be resampled. In this process, first, we reduce the occurrence of major classes. For this, we remove classes labelled as non-recursive, which is neither generic class nor has inheritance relation. We use this constraint because a null value in inheritance relation and a null in class parameter can never be a recursive class. In the next step, we increase the occurrence of minor classes. For this, we randomly duplicate recursive class data. By following these two resampling steps, we successfully reduce bias from the dataset. Moreover, Now, our dataset is ready to train ML classifiers for error-less classification.

4.3. Model-Based Learning

The processed dataset is used to train and validate six decision tree classifiers. A decision tree classifies data by forming a tree structure where each internal node is a decision node, each edge shows the condition or decision, and the leaf shows the items belonging to a single class. The tree makes decisions by splitting nodes. Different decision tree algorithms use different node-splitting criteria such as entropy, Information Gain, Gini Index, Gain Ratio, Chi-Square, etc. The six decision tree classifiers are as follows:

4.3.1. Random Tree

This algorithm forms a number of trees for each decision using random samples from the dataset. The final classification decision is based on the most frequent tree output—random tree results in overfitting.

4.3.2. J48

The J48 is popularly used as a categorical and continuous data classifier. This algorithm uses information gain as a splitting criterion. The final decision is based on the attribute(s) with the highest normalized information gain.

4.3.3. Hoeffding Tree

It is an incremental decision tree algorithm. The Hoeffding tree algorithm can learn from high-volume data streams by seeing each instance once. This algorithm results in the best with a high volume of data.

4.3.4. Logistics Model

Unlike an ordinary decision tree, it has a linear regression model at its leaves to perform supervised learning tasks.

4.3.5. Decision Stump

It is a decision tree with one root node followed by an immediate termination node. Primarily the decision is based

on a single input feature. Decision Stump is often used in ensemble techniques as a base learner.

4.3.6. Random Forest

The Random Forest is an ensemble classifier. It averages multiple decision trees trained on a random part of the same training set. Ensemble classifier results are more accurate as compared to single decision tree classifier.

4.4. Evaluation

We use nine performance metrics based on Accuracy, Confusion table, AUC, and Kappa statistics to evaluate the performances of six decision tree classifiers. The following performance metrics depend on the confusion table: True-Positive (TP) rate, False-Positive (FP) rate, Precision, Recall, and F-Score. Receiver Operating Characteristics (ROC) and Precision-Recall Curves (PRC) represent the classifier's performance through a curve on a two-dimensional axis. However, the Area Under Curve (AUC) can be calculated for both curves that give a performance score. A better classifier should have a result value close to 1 (100%) for all metrics, excluding the FP rate. The False-Positive rate should be low (˜0) as it indicates the rate of incorrectly classified instances.

5. Experimental Setup and Results

5.1. Dataset

We require a Java source code dataset with class declaration information to implement the classification models. For this, we searched several Java datasets [25], [17], [26], [35]. We collected the datasets that are open to access [16], [17]. These datasets are well-curated and available in standard file formats (CSV, XML, ARFF). However, these datasets focus on bug prediction and contain bug-related attributes. It lacks the attributes that are required for the class declaration classification problem. Therefore, we prepare the dataset with class declaration attributes. We start with data collection. For this, we searched for several Java projects with ample use of generics. We selected Java projects from two popular source code repositories, GitHub and SourceForge. Next, we collected basic statistics of the project, such as the number of lines in code (LOC), the total number of classes, and the total number of generic classes, as shown in Table I. The code analysis tool CodeMR Static Code Analyser3 helps analyse these Java projects in detail. Based on the analysis, we select ten projects containing generic classes. After data collection, we process the following steps to prepare the dataset:

1) Convert Java source code into AST nodes using JavaParser API.
2) Traverse and select attribute nodes.
3) Store the values of attributes.
4) Generate CSV file using OpenCSV Library.
5) Encoding categorical string values into numerical values using LabelEncoder.

Data collection and storage are simultaneously possible for a single project using the above APIs. Therefore, it creates ten CSV files for ten different Java projects. Eventually, we merged all ten CSV files into a single file.

5.1.1. Attributes

The total number of attributes in the dataset is nine. All nine attributes are of the categorical type. The attributes are as follows Project name, Base Class name, Base Class parameter, IsGeneric, IsInherit, Parent class name, Parent class parameter, Child class name, and Child class parameter. The IsGeneric and IsInherit attributes contain the values (True or False) for the base class. It can be generic or non-generic. In the dataset, 22 base classes are F-bounded among 229 generic classes, and 33 are F-bounded among 116 non-generic classes. Out of 257 inheritance relations, 55 are for recursive class declarations.

Table 1. Data collection

Source	Project Name	#LOC	#Classes	#Generic Classes
Github	LinemobAPI	952	21	2
Github	Spring Generic CRUD	148	16	5
Github	JICI	7845	170	23
Github	Java Type Resolver	852	69	28
Github	Generics Resolver	2778	93	28
SourceForge	GUMP-2.0.0	1619	186	7
SourceForge	Extended Java WordNet Library	1475	181	16
SourceForge	Domain Persistence	312	43	24
SourceForge	JavaMicroWeb	1006	42	40
SourceForge	MOF 2 for Java	33560	853	46
Total	10	50547	1674	219

5.1.2. The Imbalanced Dataset

Initially, we found the dataset highly imbalanced, with the skewness of the labelled class at 100:1. The imbalanced dataset causes erroneous model training that results mainly in the wrong classification. We find that the percentage of incorrect classification equals the percentage of minor class instances in the dataset. The classification algorithms work well on a balanced dataset. Therefore, we perform a resampling approach to reduce the imbalance. Currently, the dataset has a skewness of labelled classes as 6:1. This skewness ratio is enough to classify accurately using supervised classifiers.

5.2. Results

Once a dataset with a required set of attributes is available, it can train various machine learning classifiers. During the training phase, the main concern is preparing an error-less classification model to classify class labels for the given test data accurately. Several performance metrics are available to measure the accuracy of a machine-learning model. In this paper, we used nine metrics to evaluate the performance of the classifiers. The result of these performance metrics helps us compare different classification models. For the comparative analysis, we selected six decision tree classifiers. The 10-fold Cross-validation method is used to test each classification model. Using the Weka tool, we calculated performance metrics for each classifier. Table II shows the weighted average of metrics results for all six classifiers. Among nine metrics, the results of eight metrics are considered good with increasing values (˜1), and the one matrix (FP rate) is considered good with decreasing value (˜0). As we can observe, the Random Forest classifier gives the best results in all metrics. We can assert that the Random Forest is the most accurate classifier for classifying recursive and non-recursive classes. Although, if we compare classifiers based on the time taken (Figure 1), the Random Forest appears as the slowest classifier among all the six. The Random Forest uses an ensemble of decision trees. In this work, the decision depends on the majority output taken from the multiple decision trees. The trees are formed during the training stage with random examples.

Consequently, it takes more time but classifies more accurately than other decision tree classifiers. Here, we prefer accuracy over speed. Therefore, we can conclude that the Random Forest is better for the recursive class classification problem among all six decision tree classifiers.

5.3. Analysis

The machine learning approach is extensively used in the programming language domain for various tasks. This paper used six ML models to classify the class declaration patterns. We aim to acquaint the recursive classes in the source code with the help of ML models. As we already discussed, recursive classes are less frequent in Java projects—these results in an imbalanced dataset.

However, we successfully reduced the imbalance through resampling. The prepared dataset trains and validates ML models for the classification problem. There are several performance metrics available to evaluate ML algorithms for classification problems. We categorized these metrics as confusion matrix-dependent and classification dependent.

A confusion matrix holds information regarding actual classes and predicted classes. The accuracy can be calculated in both positive and negative aspects. Therefore, confusion table-based metrics are considered more accurate for performance evaluation. The metrics such as TPR (True-Positive Rate), FPR (False-positive Rate), Precision, Recall, F-score, ROC, and PRC use the information from the confusion matrix. These metrics tell how accurately and inaccurately a class is classified from the given dataset. Higher values of these metrics (except FPR) indicate higher accuracy.

As mentioned in Table 2, we noticed metrics values for the two classifiers, namely Hoeffiding and Decision Stump Tree. We found that the number of incorrectly classified instances is comparatively higher in these two classifiers, causing a decrease in metrics values. Accuracy and Kappa metrics are classification dependent.

Table 2. Performance metrics

Classifiers	TP Rate	FP Rate	Accuracy	Precision	Recall	F-Score	Kappa	ROC	PRC
Random Tree	0.985	0.062	0.985	0.985	0.985	0.985	0.945	0.962	0.975
J48 Decision Tree	0.991	0.016	0.991	0.991	0.991	0.991	0.967	0.987	0.988
Hoeffding Tree	0.831	0.783	0.831	0.770	0.831	0.781	0.072	0.539	0.759
Logistics Tree	0.991	0.031	0.991	0.991	0.991	0.991	0.967	0.985	0.990
Decision Stump Tree	0.890	0.183	0.889	0.905	0.890	0.895	0.632	0.790	0.849
Random Forest	0.994	0.016	0.994	0.994	0.994	0.994	0.978	1.000	1.000

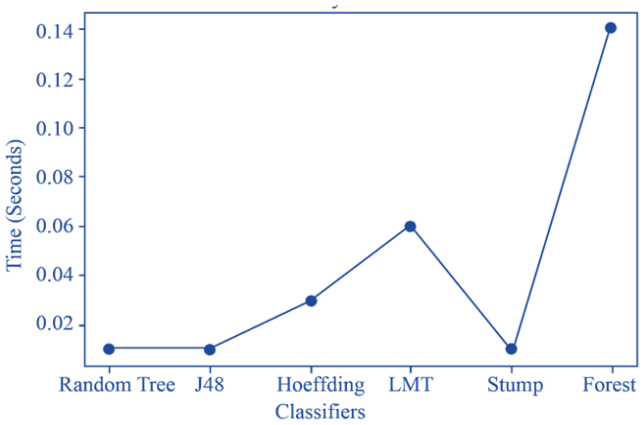

Fig. 1 Time taken by classifiers

Accuracy is the measurement of correct predictions out of the total predictions. However, accuracy cannot be considered a sufficient performance matrix as it hides or ignores abnormalities. The Kappa matrix compares observed accuracy with randomly expected accuracy. The 100% Kappa values indicate perfect classification accuracy.

Among all six classifiers, we find that the accuracy of four is very good, with minor differences. These four classifiers are Random tree, J48, LMT, and Random Forest tree. Among them, the most accurate classifier is the Random Forest which follows ensemble learning for classification.

The other two classifiers with less accuracy are Hoeffding and the Decision Stump tree. The accuracy is lower since the dataset has a lower volume of data, and Hoeffding requires high-volume data to perform well. Also, the decision of the stump tree depends on a single attribute, though the classification problem does not depend only on a single attribute of the dataset.

5.3.1. Comparative Analysis

Keeping disjoint shape (recursive class) and materials (non-recursive) is one of the simple methods to eliminate the chances of code failure due to recursive generics [4]. To validate their claim, they analyze millions of Java codes in the form of a usage graph. The usage graph shows self-loop for class-level cycles and ignores parameter-level cycles. For example, the graph shows self-loop in *T extends Comparable<T>*, but it does not show self-loop in *T<E extends Comparable<E>>*.

The self-loop at the parameter level can easily be detected by analyzing the source code. Using machine learning approaches, the source code analysis becomes trivial. Also, their analysis method only finds 17 recursive patterns in millions of Java codes. However, our dataset prepared by fetching essential program constructs from 10 Java projects has 55 recursive patterns.

6. Conclusion

In this paper, we perform a comparative analysis among six decision-tree-based classifiers for classifying recursive and non-recursive generic types at class declaration. The classifiers are trained and tested on the given data from the dataset. The dataset was prepared by collecting data from ten open Java projects. The analysis depends on the results of nine performance metrics. These nine metrics' results reassert that the ensemble-based Random Forest is the most accurate.

The paper focuses on classifying the recursive and non-recursive generic types at class declaration. The dataset can be used in future work to model bug pattern prediction in the recursive class declaration. We propose to expand the dataset with more attributes and data. Hence, the dataset can further train ML models to predict unsound recursive patterns like expansive inheritance.

References

[1] Luca Cardelli, and Peter Wegner, "On Understanding Types, Data Abstraction and Polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, pp. 471–523, 1985. [CrossRef] [Google Scholar] [Publisher Link]

[2] María Lucía Barrón–Estrada, and Ryan Stansifer, "Inheritance, Generics and Binary Methods in Java," *Computing and Systems*, vol. 7, no. 2, pp. 113–122, 2003. [Google Scholar] [Publisher Link]

[3] Peter Canning et al., "F-Bounded Polymorphism for Object-Oriented Programming," *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pp. 273–280, 1989. [CrossRef] [Google Scholar] [Publisher Link]

[4] Ben Greenman, Fabian Muehlboeck, and Ross Tate, "Getting F-Bounded Polymorphism Into Shape," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 89–99, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[5] Daniel Smith, and Robert Cartwright, "Java Type Inference is Broken: Can We Fix It?," *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, pp. 505–524, 2008. [CrossRef] [Google Scholar] [Publisher Link]

[6] Ross Tate, Alan Leung, and Sorin Lucian Lerner, "Taming Wildcards in Java's Type System," *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 614–627, 2011. [CrossRef] [Google Scholar] [Publisher Link]

[7] Stefan Wehr, Ralf Lämmel, and Peter Thiemann, "JavaGI: Generalized Interfaces for Java," *European Conference on Object-Oriented Programming*, pp. 347–372, 2007. [CrossRef] [Google Scholar] [Publisher Link]

[8] Monalisa Jena, and Satchidananda Dehuri, "Decision Tree for Classification and Regression: A State-of-the-Art Review," *Informatica*, vol. 44, no. 4, pp. 405–420, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[9] Andrew J Kennedy, and Benjamin C Pierce, "On Decidability of Nominal Subtyping with Variance," *Proceeding FOOL/WOOD ACM*, 2007. [Google Scholar] [Publisher Link]

[10] P. Wadler, and S. Blott, "How to Make Ad-Hoc Polymorphism Less Ad Hoc," *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 60–76, 1989. [CrossRef] [Google Scholar] [Publisher Link]

[11] Tumula Mani Harsha et al., "Survey on Resume Screening Mechanisms," *SSRG International Journal of Computer Science and Engineering*, vol. 9, no. 4, pp. 14-22, 2022. [CrossRef] [Publisher Link]

[12] Fitzroy Nembhard, Marco Carvalho, and Thomas Eskridge, "Extracting Knowledge from Open Source Projects to Improve Program Security," *Proceeding SoutheastCon, IEEE*, pp. 1–7, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[13] Timothy Chappelly et al., "Machine Learning for Finding Bugs: An Initial Report," *Proceeding IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation*, pp. 21–26, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[14] Nevena Rankovic et al., "Influence of Input Values on the Prediction Model Error using Artificial Neural Network Based on Taguchi's Orthogonal Array," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 20, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[15] Neha Kumari, and Rajeev Kumar, "Profiling JVM for AI Applications using Deep Learning Libraries," *Machine Learning for Predictive Analysis Springer Singapore*, pp. 395–404, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[16] Miltiadis Allamanis, and Charles Sutton, "Mining Source Code Repositories at Massive Scale using Language Modeling," *Proceeding 10th Working Conference on Mining Software Repositories, IEEE*, pp. 207–216, 2013. [CrossRef] [Google Scholar] [Publisher Link]

[17] Rudolf Ferenc et al., "A Public Unified Bug Dataset for Java and its Assessment Regarding Metrics and Bug Prediction," *Software Quality Journal*, vol. 28, pp. 1447–1506, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[18] Alexander LeClair, and Collin McMillan, "Recommendations for Datasets for Source Code Summarization," *arXiv Preprint*, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[19] Miltiadis Allamanis et al., "Suggesting Accurate Method and Class Names," *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, pp. 38–49, 2015. [CrossRef] [Google Scholar] [Publisher Link]

[20] Uri Alon et al., "A General Path-Based Representation for Predicting Program Properties," *Proceeding ACM SIGPLAN Notices*, pp. 404–419, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[21] Nayak Suvra et al., "Comparative Analysis of Har Datasets Using Classification Algorithms," *Computer Science and Information Systems*, vol. 19, no. 1, pp. 47–63, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[22] Sumitra Nuanmeesri, Wongkot Sriurai, and Nattanon Lamsamut, "Stroke Patients Classification using Resampling Techniques and Decision Tree Learning," *International Journal of Engineering Trends and Technology*, vol. 69, no. 6, pp. 115–120, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[23] Eibe Frank et al., "Weka-A Machine Learning Workbench for Data Mining," *Data Mining and Knowledge Discovery Handbook Springer*, pp. 1269–1277, 2010. [CrossRef] [Google Scholar] [Publisher Link]

[24] Ritu Ratra, and Preeti Gulia, "Experimental Evaluation of Open-Source Data Mining Tools (Weka and Orange)," *International Journal of Engineering Trends and Technology*, vol. 68, no. 8, pp. 30–35, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[25] Jens Dietrich et al., "Xcorpus–An Executable Corpus of Java Programs," *Journal of Object Technology*, vol. 16, no. 4, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[26] Cristina V. Lopes et al., "Dejavu: A Map of Code Duplicates on GitHub," *Proceeding ACM Programming Languages*, vol. 1, no. 84, pp. 1-28, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[27] G. Anil Kumar, "Research Methodology on Code Clone Detection with Refactoring using Textual and Metrics Analysis in Software," *SSRG International Journal of Computer Science and Engineering*, vol. 2, no. 12, pp. 19-23, 2015. [CrossRef] [Publisher Link]

[28] Robin Milner, "A Theory of Type Polymorphism in Programming," *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348-375, 1978. [CrossRef] [Google Scholar] [Publisher Link]

[29] Mark Day et al., "Subtypes vs. Where Clauses: Constraining Parametric Polymorphism," *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 156-168, 1995. [CrossRef] [Google Scholar] [Publisher Link]

[30] John Altidor, Shanshan Huang, and Yannis Smaragdakis, "Taming the Wildcards: Combining Definition-and Use-Site Variance," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 602-613, 2011. [CrossRef] [Google Scholar] [Publisher Link]

[31] Luis Mastrangelo, Matthias Hauswirth, and Nathaniel Nystrom, "Casting About in the Dark: An Empirical Study of Cast Operations in Java Programs," *Proceedings of the ACM on Programming Languages*, pp. 1-31, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[32] Kim B. Bruce, "Some Challenging Typing Issues in Object-Oriented Languages," *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 8, pp. 1-29, 2003. [CrossRef] [Google Scholar] [Publisher Link]

[33] Erik Ernst, "Family Polymorphism," *European Conference on Object-Oriented Programming, Springer Berlin Heidelberg*, pp. 303-326, 2001. [CrossRef] [Google Scholar] [Publisher Link]

[34] Francisco Ortin, Guillermo Facundo, and Miguel Garcia, "Analyzing Syntactic Constructs of Java Programs with Machine Learning," *Expert Systems with Applications*, vol. 215, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[35] Ewan Tempero et al., "The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies," *Proceeding 17th Asia Pacific Software Engineering Conference, IEEE*, pp. 336–345, 2010. [CrossRef] [Google Scholar] [Publisher Link]

[36] Imad Eddine Araar, and Hassina Seridi, "Software Features Extraction from Object-Oriented Source Code using an Overlapping Clustering Approach," *Informatica*, vol. 40, no. 2, 2016. [Google Scholar] [Publisher Link]