Development of a Genetic Algorithm for the evolution of Decision Trees for Code Smell Detection

Authors: Salerno Daniele, Simone Benedetto

ID: 0522501357, 0522501334

Supervisor: Prof. Fabio Palomba, Prof. Dario Di Nucci

Tutor: Dr. Manuel De Stefano

Department of Computer Science, UNISA , University of Salerno - June 6, 2022

Abstract: Code smells represent a well-known problem in software engineering because they reduce software quality. Therefore, the goal of our study is to build a Genetic Algorithm to evolve Decision Tree populations and evaluate their performance. From the results obtained from the empirical study, we cannot establish the validity of the proposed method as an effective solution for code smell detection.

Keywords: Code Smell, Genetic Algorithm, Decision Tree.

1 INTRODUCTION

As Lehman's first law states, change in a software system is unavoidable. Developers are therefore forced to make, for example, changes to implement new features or to correct defects found by users. Unfortunately, developers are not always able to produce high-quality software due to the pressure of short deadlines or high workloads. As a result, they may ignore good programming practices to help deliver the product on time, introducing so-called technical debt. One of the most significant forms of technical debt are code smells. They refer to poor design or implementation choices applied by developers during the maintenance operations of a software system, which reduce the quality of the software [1]. For this reason, our study aims to evaluate the effectiveness of using a Genetic Algorithm to evolve a population of Decision Tree for the classification of code smells. From the experiments performed, we cannot affirm whether using Evolutionary Decision Trees is a possible solution to the problem.

> To provide replicability and reproducibility of our experiment, we included the datasets and scripts in the bibliography [2].

2 RELATED WORK

For code smell detection are used heuristics-based and machine learning-based approaches.

For ***heuristics-based*** approaches, a set of metrics are first computed and then are defined thresholds on these metrics to discriminate smelly and non-smelly instances. Of course, the selection of thresholds strongly influences their accuracy.

For ***machine learning-based*** approaches, is applied the supervised learning, which means that a set of independent variables (predictors) are used to predict the value of a dependent variable (the smelliness of a class) using a machine learning classifier. The model can be trained using two strategies: *within-project*, using multiple versions of the project under analysis to build the dataset, or *cross-project*, using similar projects but considering a single version.

De Stefano *et al.*[3] showed that there were no significant differences between the two strategies.

Machine learning-based approaches clearly differ from heuristics-based approaches because they are based on classifiers to discriminate class smell instead of predefined thresholds on computed metrics.

Pecorelli *et al.*[4] showed that heuristic-based approaches provide slightly better performance than machine learning-based approaches, although the results are still too low.

3 CONTEXT OF THE STUDY

The context of our study is composed by a set of projects and the code smells types that we intend to identify.

Dataset. For our study, we used a public code smell dataset [5] composed by multiple version of 30 Java open-source projects. Of these projects we considered different version of ***9*** of them. Tables 2 and 3 reports the main characteristics of these projects.

For each projects are stored the values of the metrics useful to identify the different types of code smell. Table 4 reports the list of metrics used. Some metrics were not reported because their definition could not be retrieved.

Code Smells. In our study, we considered ***5*** types of code smell namely *Complex Class*, *Large Class (or God Class)*, *Lazy Class*, *Refused Bequest* and *Spaghetti Code*. Table 1 reports a description for each of them.

Table 1: Code smells considered in our study

Name	Description
Complex Class	A class having at least one method having a high cyclomatic complexity.
Large Class	A class having huge dimension and implementing different responsibilities.
Lazy Class	A class that is under-used.
Refused Bequest	A derived class that doesn't honor the contract of the base class. It is based on LISKOV Substitution Principle, because it violates this principle.
Spaghetti Code	A class that implements complex methods interacting between them, with no parameters, using global variables.

Table 2: Projects considered in our study

Project Name	Release Tag	Classes	LOC
ant	rel-1.6.0	951	132 253
ant	rel-1.6.1	950	133 245
ant	rel-1.6.2	950	134 496
ant	rel-1.6.3	949	136 058
ant	rel-1.6.4	950	136 360
ant	rel-1.7.0	1157	160 762
ant	rel-1.7.1	1159	162 717
ant	rel-1.8.1	1083	161 879
ant	rel-1.8.2	1083	162 799
ant	rel-1.8.3	1082	163 201
argouml	VERSION_0_12	812	83 055
argouml	VERSION_0_14	1258	113 756
argouml	VERSION_0_18_1	1361	139 414
argouml	VERSION_0_20	1467	148 235
argouml	VERSION_0_22	1546	155 335
argouml	VERSION_0_24	1557	144 057
argouml	VERSION_0_26	1829	164 882
argouml	VERSION_0_30	2207	189 796
argouml	VERSION_0_30_1	2125	187 540
argouml	VERSION_0_30_2	2125	187 557
argouml	VERSION_0_32_1	2130	188 455
argouml	VERSION_0_32_2	2130	188 455
cassandra	cassandra-0.7.0	415	43 179
cassandra	cassandra-0.7.2	412	46 251
cassandra	cassandra-0.7.3	413	46 370
cassandra	cassandra-0.8.0	517	56 594
cassandra	cassandra-0.8.1	515	56 102
cassandra	cassandra-0.8.3	514	56 904
cassandra	cassandra-1.0.0	613	63 867
cassandra	cassandra-1.1.0	711	75 087
elasticsearch	v0.12.0	1713	118 458
elasticsearch	v0.13.0	1799	125 477
elasticsearch	v0.14.0	1880	132 150
elasticsearch	v0.15.0	1965	143 696
elasticsearch	v0.16.0	2047	155 344
elasticsearch	v0.17.0	2340	178 700
elasticsearch	v0.18.0	2427	189 187
elasticsearch	v0.19.0	2336	187 779
hadoop	release-0.1.0	115	15 580
hadoop	release-0.2.0	224	27 336
hadoop	release-0.3.0	224	28 383
hadoop	release-0.4.0	225	29 306
hadoop	release-0.5.0	223	29 087
hadoop	release-0.6.0	223	29 778

Table 3: Projects considered in our study

Project Name	Release Tag	Classes	LOC
hadoop	release-0.7.0	222	32 966
hadoop	release-0.8.0	223	33 778
hadoop	release-0.9.0	332	47 219
hsqldb	2.0.0	475	145 083
hsqldb	2.2.0	475	155 019
hsqldb	2.2.1	475	155 105
hsqldb	2.2.2	584	178 671
hsqldb	2.2.3	475	155 890
hsqldb	2.2.4	475	155 913
hsqldb	2.2.5	584	179 626
hsqldb	2.2.6	584	179 530
hsqldb	2.2.7	583	179 533
hsqldb	2.2.8	583	179 541
nutch	release-0.7	332	40 790
nutch	release-0.8	322	30 701
nutch	release-0.9	323	32 138
nutch	release-1.1	425	43 362
nutch	release-1.2	425	43 622
nutch	release-1.3	221	24 333
nutch	release-1.4	221	23 747
qpid	0.10	1730	155 438
qpid	0.12	1537	149 343
qpid	0.14	1541	148 456
qpid	0.16	1538	159 390
qpid	0.18	2088	188 996
xerces	Xerces-J_1_0_4	453	64 683
xerces	Xerces-J_1_2_0	447	68 064
xerces	Xerces-J_1_2_1	447	68 164
xerces	Xerces-J_1_2_2	447	68 822
xerces	Xerces-J_1_2_3	447	68 913
xerces	Xerces-J_1_3_0	441	72 181
xerces	Xerces-J_1_3_1	441	73 744
xerces	Xerces-J_1_4_0	434	76 289
xerces	Xerces-J_1_4_1	434	77 304
xerces	Xerces-J_1_4_2	434	77 425

4 DATA ENGINEERING

Before performing the experiments, some preprocessing steps are necessary to obtain reliable results.

Data imputation. Analyzing the dataset we noticed that some of the feature values were null (NaN). Table 5 shows these features. Datasets with missing values can cause problems for many machine learning algorithms. Therefore we applied KNN imputation to replace such values. *K-nearest neighbors* is an algorithm that employs feature similarity to predict the values of any new data points. This means that the new point is assigned a value based on how closely it resamples the points in the dataset.

Since our goal is to build ad-hoc classifiers for each type of code smell we used 5 datasets considering one smell for each dataset.

Data balancing. As can be seen in Figure 1, the original dataset is unbalanced. Figure 2 shows the number of smells present in our 5 datasets.

To avoid problems in the training phase, we need to apply the oversampling technique to obtain a balanced dataset with a more homogeneous distribution.

Table 4: List of metrics

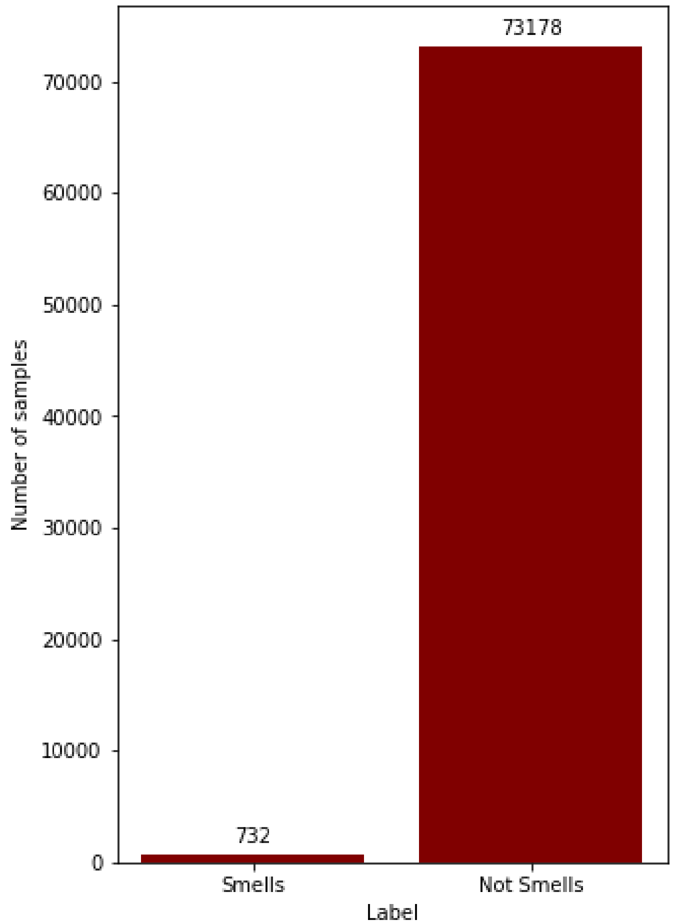
Size	Complexity	Cohesion
ELOC	CYCLO	LCOM
LOC	WMC	TexualCohesion
LOCNAMM	WMCNAMM	
NOA	TexualEntropy	
NOM		
NOMNAMM		

Coupling	Encapsulation	Inheritance
CBO	NOPA	DIT
		FanIn
		NOC

Table 5: Metrics with NaN value

Metric	Number of NaN
WLOCNAMM	1352
TextualCohesion	1
TextualEntropy	8

Figure 1: Distribution of target attributes in the original dataset



Before applying oversampling, we performed Stratified K-Fold obtaining 10 folds for each dataset.

Then, we applied oversampling on the generated training sets while leaving the test sets intact. To create new artificial instances, we used the *Synthetic Minority Oversampling Technique (SMOTE)*. It employs the KNN algorithm to find neighbors of the minority class observations and then creates new artificial instances based on the distribution of data of the neighbors selected.

5 PRELIMINARY EXPERIMENTS

Before proceeding with the implementation of the Genetic Algorithm, we wanted to use a Decision Tree to better understand the problem. It also has been useful for us to understand how to build the Decision Tree for the Genetic Algorithm.

Figure 2: Number of smells present in the 5 datasets

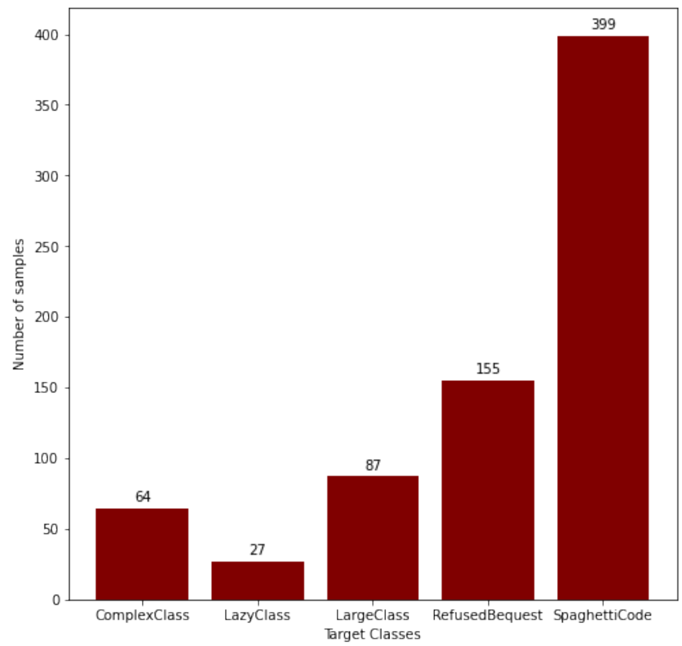


Table 6: Classification results with Decision Tree

Smell considered	F-measure
Complex Class	[0.9991, 0.6584]
Large Class	[0.9990, 0.6859]
Lazy Class	[0.9996, 0.6595]
Refused Bequest	[0.9986, 0.7252]
Spaghetti Code	[0.9961, 0.6021]

Decision Tree classifier. To perform the classification phase we used a *Decision Tree*, with maximum depth 10, 20 and 40, evaluating the performances obtained with the metric F-Measure. The results obtained from these experiments were disappointing, so we tried omitting the parameter related to maximum depth obtaining much better results which are shown in Table 6.

After training we verified whether the misclassified samples were false positives or false negatives, reported in Table 7.

Table 7: Number of wrong predictions, counting false positives and false negatives

Smell considered	Test size	True positive	True negative	Wrong Prediction	False positive	False negative
Complex Class	73 910	52	73 740	**118**	106	12
Large Class	73 910	70	73 704	**136**	119	17
Lazy Class	73 910	25	73 828	**57**	55	2
Refused Bequest	73 910	140	73 577	**193**	178	15
Spaghetti Code	73 910	336	73 014	**560**	497	63

6 EMPIRICAL STUDY DESIGN

The purpose of this study is to analyze the results that are obtained from the use of Evolutionary Decision Trees, constructed using a Genetic Algorithm, for code smell detection.

6.1 Genetic Algorithm

The previous experiments carried out using a Decision Tree confirm that it is possible to use this classifier for the problem related to the identification of code smells. We have therefore a confirmation of the feasibility of the experiment objective of our work, that is to create a version based on Genetic Algorithm in order to evaluate the performance that is able to guarantee.

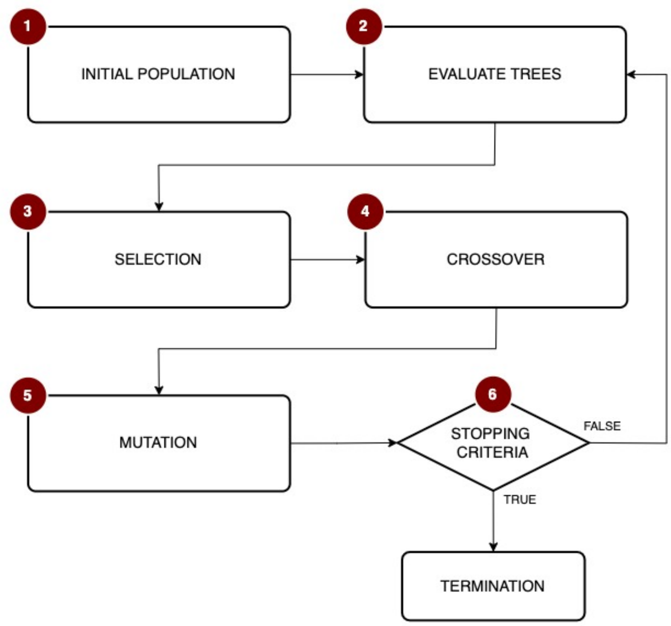
Before defining the steps of the Genetic Algorithm, shown in Figure 3, we need to specify the encoding of individuals.

Trees structure. The individuals of the population are trees and they are composed by two classes:

- ***Leaf***: represent the class to which the objects belongs (are the nodes that give us the classification);
- ***Decision***: represent the decision node and use a *Rule* to choose which of the two children will have the decision. It can have two types of childrens: Decision or Leaf.

❶ ***Initial population.*** To generate the starting population we created it randomly. For each tree, we first generate a Decision node as root, and then we random choose if the type of next node will be a Decision or Leaf. The last step will be repeated until the tree has reached the maximum depth or when all the nodes of the last level are leaves.

Figure 3: Executions steps of Genetic Algorithm



❷ ***Evaluate trees.*** In order to evaluate each tree, we defined the following fitness function:

$$(alpha1 * accuracy) + (alpha2 * height\_score)$$

- ***alpha1*** is penalty for misclassification;
- ***alpha2*** is penalty for large trees;
- ***height_score*** is calculated as (*1-height of the tree*).

We set *alpha1* to 0.99 and *alpha2* to 0.01. We defined this formula since a small tree would lead to underfitting problems and a complex tree to overfitting problems.

Figure 4: Crossover

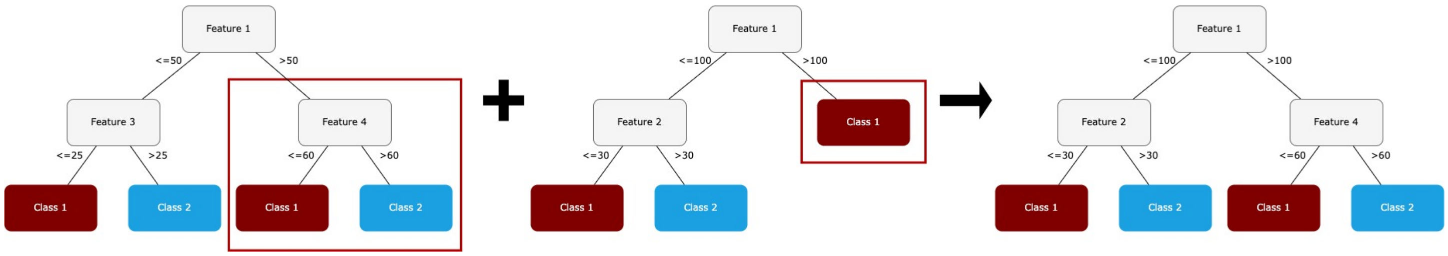
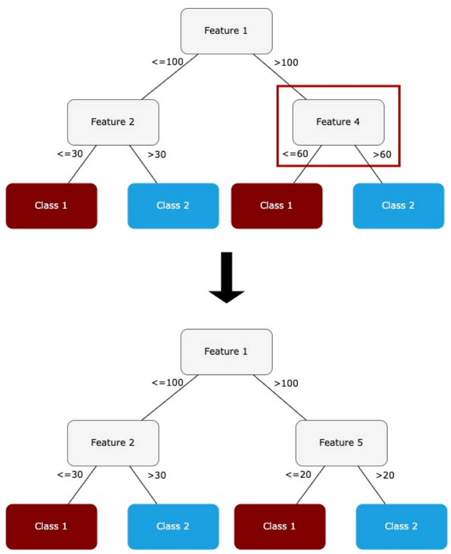


Figure 5: Mutation



3 ***Selection.*** To select the parents that will be used to create the next generation we used *Roulette Wheel Selection*, assigning at each individual a probability of selection based on their fitness score. The size of mating pool is 20% of population size.

4 ***Crossover.*** To perform the crossover we combined pairs of parents. To obtain a new child, as can be seen in Figure 4, we select a random node of first parent and we copy the entire subtree starting from this node. Next, we randomly select a leaf of second parent and we replace the leaf with subtree copied.

5 ***Mutation.*** To perform the mutation we randomly select a node. If the type of the node is Leaf we change the result class. If the type of the node is Decision we change the Rule. Figure 5 shows the mutation of Decision node.

6 ***Stopping criteria.*** The algorithm stops when it arrives to the last iteration, or when for a determined number of iterations do not happen improvements.

Table 8: Results of two executions of Genetic Algorithm

Smell considered	F-measure
Complex Class	[0.9652, 0.0693]
Large Class	[0.9543, 0.1228]
Lazy Class	[0.7525, 0.0032]
Refused Bequest	[0.7413, 0.0210]
Spaghetti Code	[0.8152, 0.0570]
Complex Class	[0.9735, 0.1817]
Large Class	[0.9596, 0.0710]
Lazy Class	[0.7441, 0.0014]
Refused Bequest	[0.7255, 0.0137]
Spaghetti Code	[0.5977, 0.0234]

7 ANALYSIS OF RESULTS

Table 8 shows the results of executions of the Genetic Algorithm. The parameters used for the experiments are the following.

Execution N.1

population_size: 20;
epoch: 10;
minimum_tree_depth: 3;
maximum_tree_depth: 10.

Execution N.2

population_size: 20;
epoch: 10;
minimum_tree_depth: 3;
maximum_tree_depth: 15.

As can be seen, the results obtained are very low. This is probably related to the number of executions performed since, with the above parameters, the training phase lasted about 4 hours. Given this, since we do not have enough results, we cannot affirm that Evolutionary Decision Trees can be used for code smell detection. We think that by performing more executions, even using different parameters, better results could be obtained, hypothetically on par with those obtained using the Decision Tree.

8 CONCLUSION AND FUTURE WORK

In this paper we have shown the various steps required to use Evolutionary Decision Trees, constructed using a Genetic Algorithm, for code smell detection.

Due to problems related to the timing of each execution of the Genetic Algorithm, we do not have enough results to affirm that Evolutionary Decision Trees are a suitable solution for code smell detection. The results obtained from the preliminary experiments, described in the chapter 5, show the feasibility of using the Decision Tree as a solution to the problem.

The next study could focus on conducting multiple experiments, using different parameters and performing an appropriate number of experiments to confirm whether Evolutionary Decision Trees could prove to be a good solution to the problem.

In addition, different types of crossover and mutation could be used.

References

[1] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2017. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering* 43, 11 (2017), 1063–1088.

[2] Salerno Daniele, Simone Benedetto. 2022. Development of a Genetic Algorithm for the evolution of a Decision Trees for Code Smell Detection - Git-Hub Repository. https://tinyurl.com/3eexcpnm.

[3] De Stefano, M., Pecorelli, F., Palomba, F., & De Lucia, A. (2021, August). Comparing within-and cross-project machine learning algorithms for code smell detection. In Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution (pp. 1-6).

[4] Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, and Andrea De Lucia. 2019. Comparing heuristic and machine learning approaches for metric-based code smell detection. In 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC). IEEE, 93–104. *J. Chemical Physics* Vol. 374 (2003) p. 201-205.

[5] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* 23, 3 (2018), 1188–1221.