ANALYSIS OF CODE SMELL AT CLASS AND METHOD LEVEL

Abhilasha Ojha

Banaras Hindu University

Department of Computational Science and Applications

Banaras, Uttar Pradesh

Abstract- *Any breach of fundamental design principles that degrades the overall quality of the code is referred to as a code smell. For developers, detecting code smells is difficult, and its informal characterization leads to the use of a variety of detection approaches and tools. Several tools have been developed over the years to detect code smells with different features. Using the iPlasma tool, this study examines and analyses code smells in 14 Java applications. This study aims to detect class level and method level code smell in an object-oriented paradigm. The accuracy of the datasets is discussed using Weka software with the application of several machine learning algorithms.*

Keywords: Code smells · iPlasma · Software metrics · Machine Learning · JRip · J48

INTRODUCTION

Code smells are considered to be a bad symptom of the code, or, as Martin Fowler (1999) puts it, "surface indications that usually correspond to deeper problems in the software system." They are neither defects in the code nor obstacles to the code's execution. They simply indicate a design flaw and may raise the likelihood of errors and program failure in the future.

Code smells is identified when improper programming methods lead to flaws in the design and implementation of the code, which can wreak havoc on software development, maintenance, and evolution (Lanza and Marinescu 2006). In software development, code smells are supposed to cause problems with comprehension and maintenance. As a result, detecting and eliminating them is essential. Detection of code smells in large software systems, has always been time-consuming, costing resources as well as fault-prone operation. Manual inspection of the systems is slow and ineffective. Thus, several tools were developed for the detection of the same.

There are an increasing number of software analysis tools available today that may discover programming faults, identify improper programming practises, highlight unique errors, and increase the software engineer's grasp of the structural properties of the programme under development. Some of these tools are used to detect bad code smells during the development phase, while others are used after the programming phase is over. Even if the tools are applied to the same source code, their results may differ due to the fact that they use different detection rules.

There are various tools available for the detection of code smell, namely, PMD, iPlasma, Jdeodorant and Stench Blossom. These tools are quite helpful in detecting problematic code smells and enhancing the software project's integrity and manageability. In this study, only iPlasma tool has been used because other tools were not compatible with the projects that were chosen for the paper. However, other tools can be used for verification and validation purposes.

The aim of the paper is to create datasets using iPlasma and validate it with the application of several Machine learning algorithms. With the creation of the dataset, various patterns and trends within it can be revealed for future works.

RELATED WORK

The previous works on code smell can be broadly classified into two categories, one is the traditional way of detecting code smells with the help of metrics and rules (manually and with the help of tools). The other category would be Machine Learning approach. Both the approaches have their benefits and drawbacks.

Research on comparison of several tools have been studied previously by many researchers (Paiva et al. 2017) (Marinescu et al. 2005) (Murphy-Hill and Black 2010). One may argue that Machine Learning approach is a better way of detecting code smells (Arcelli Fontana, Francesca, et al. 2016). But traditional way of detecting code smells, that is, detection of code smells on the basis of metrics and rules that rely on humans, yet not considered to be the best approach, by the researchers. Metric-based detection of code smell is made easier by application of tools. A literature of list of tools was reviewed by Fernandes et al. (2016). The constraint with using tools is that, they can detect only few of code smells. A lot of literature around detection of code smells with tools are available, namely, (Moha et al. 2010) (Zazworka and Ackermann 2010).

In this study, four datasets have been generated, using iPlasma, on code smells and accuracy, F-measure and Area under ROC were measured for different Machine learning algorithms. Similar approach was taken by Johansson et al. (2020) in his "*Ensemble approach to code smell identification: Evaluating ensemble machine learning techniques to identify code smells within a software system".* The difference in both the approaches is that the dataset in the later was generated using WekaNose. Both the works shows almost similar results.

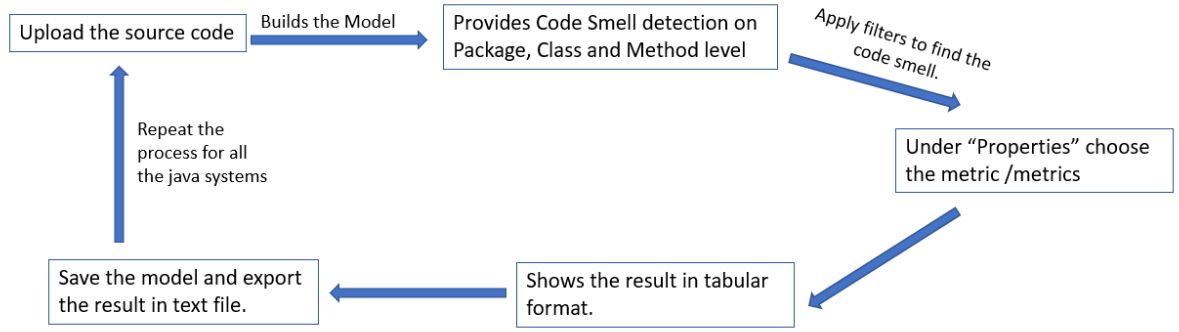
WORKING OF IPLASMA TOOL

iPlasma[1] provides an integrated environment for object-oriented software systems to get quality analysis done efficiently. This platform is also inclusive of all types of support needed for all the phases of analysis, from model extraction up to high-level metrics-based analysis, or detection of code duplication.

IPlasma starts with the source code (C++ or Java) and provides comprehensive assistance for all phases of the analysis process, from interpreting the code and developing a model to a simple definition of the desired analyses, including code redundancy detection, all of which are incorporated by a uniform front-end, insider.

The steps performed to generate the code smell dataset is mentioned below:

Fig 1: Flowchart of iPlasma.



[1] https://github.com/abhilashaojha/CodeSmells/blob/main/iPlasma6.zip

1. The source file has to be uploaded in the option "Load>Source file". Thereby, the model is extracted from the source code.

"The purpose of constructing the model is to retrieve information from the source program that is relevant to a given goal. Thus, understanding the types of the researched system, methods and variables, as well as have knowledge about their utilizations, inheritance ties between classes, the call-graph, and so on, is required for object-oriented model development"

1. Java-based systems iPlasma extracts all of the information in the form of an object-oriented meta-model, i.e., Memoria (2004), using the open-source parsing package 'Recoder[2]'.

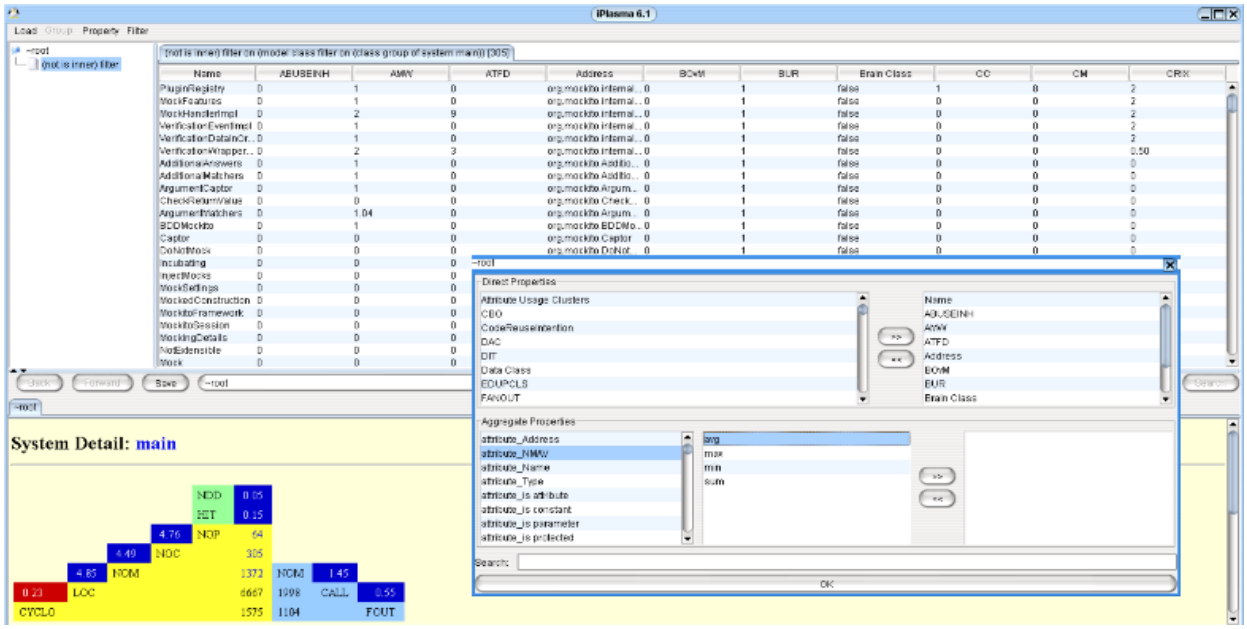
2. iPlasma allows the code smells in the uploaded Java (or C++) system to be detected on package, class and method level. Various metrics, associated with code smells on different levels are detected.

Fig.1: Code smells detected by iPlasma

Package Level	Class Level	Method Level
SDP Violation	Data Class	Brain Method
Model Package	God Class	Shotgun Surgery
	Brain Class	Feature Envy
	Schizofrenic Class	Long Parameter list
	Refused Parent Bequest	Liskov Violator

3. In this step, after the successful detection of code smells, iplasma produces result in tabular format. An individual can export that and convert it to csv/xlsx as per his choice.

Fig 2: Screenshot of code smell generated for a java project along with its associated metrics



[2] https://sourceforge.net/projects/recoder/

METHODOLOGY

Code smells have been largely discussed by both the software engineering community and practitioners from the industry. Bad smell has a significant impact on the functionality and reliability of a software program. For example, we might see redundant code as a potential hazard to future software maintenance. There are a number of tools available to help detect code smells in programs written in various languages. Available methods for detecting code smells typically use one of the six strategies listed below:

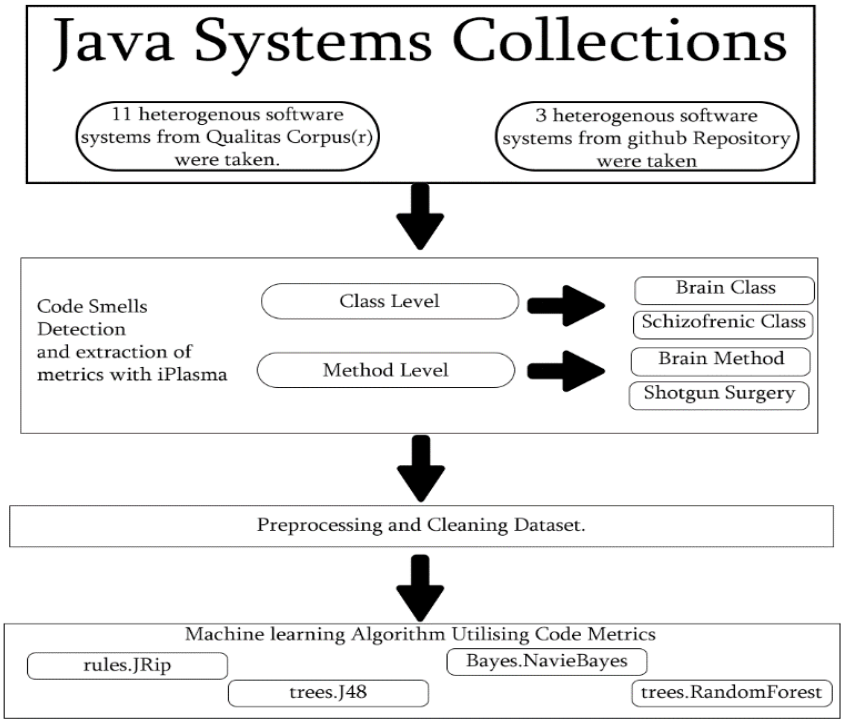
i. Manual,
ii. Symptom-based detection,
iii. Metric-based,
iv. Probabilistic,
v. Search-based, and
vi. Cooperative-based approaches.

To detect code smells, many strategies use software metrics, some of which are metrics taken from third-party tools and then applied threshold values. Measuring code smells necessitates accurate quantification of design rules and practises, which presents a set of difficulties. In order to overcome these difficulties, we use the tool iPlasma which has a set of software metric defined for package, class and method level code smells.

EXPERIMENT SETUP

The principal steps of the workflow for detecting code smells, producing and analysing datasets, and verifying their accuracy are depicted in the diagram below.

Fig. 3: Workflow of the research paper



Let us discuss all the steps one by one:

1. **Collection of Repository of Java systems:**

For our analysis, we looked at the Qualitas Corpus (QC) of systems assembled by Evan Tempero (2010). The corpus, and in particular the collection we used, 20120401r, is composed of 111 systems written in Java, characterized by different sizes and belonging to different application domains. A total of 14 java projects were considered, 11 of which belong to Qualitas corpus (See table 8) and remaining are extracted from GitHub Repository [3 4 5]

Code Smell detection and Metric Extraction with iPlasma:

Code Smells, on both Class and method level, were detected using iPlasma tool which has been discussed earlier in the paper. Large set of object-oriented metrics associated with the code smells was extracted on class and method level. The metrics were computed on all 14 java systems. The code smells detected on class level are discussed below:

- *God Class (GC)*: God class refers to those classes that tend to centralize the intelligence of the system. It references a large number of unrelated or unorganized classes. A God Class has high complexity, low class cohesion, and has access to data of foreign classes.

- *Brain Class (BC)*: Similar to God class, Brain Class refers to complex functionality of the system. It manifests itself as a long and complex class that becomes difficult to understand and maintain.

- *Data Class (DC)*: Data classes refer to the classes that just store a lot of data and no method. Generally, a class should consist of both data and methods or they would not operate on their own. This leads to the definition and consideration of Data Class as a code smell.

- *Schizofrenic Class (SC)*: Schizofrenic Class code smell refers to a derived class that overrides behaviour of the base class and can be used as a template for objects whose behaviours are refined by the base classes.

- *Refused Parent Bequest (RPB)*: Refused parent Bequest refers to the subclasses that refuses some of the functionalities and methods provided by the parent class simply because they find it unnecessary. This leads to confusion and complexity and hence considered to

Table 1: Class level Code smells in selected Java systems.

Java Project	Brain Class	God Class	Data Class	Schizofrenic Class	Refused Parent Bequest
Freemind-0.9.0	2	19	14	39	2
Argouml-0.34	11	62	27	55	10
Sunflow-0.07.2	5	9	2	19	-
Retrofit	-	-	-	17	-
Apache-ant-1.8.4	12	74	68	117	71
ArtOfIllusion-2.8.1	27	41	43	44	4
Mockito	-	1	9	140	-
Fitjava-1.1	-	1	5	8	1
c-jdbc-2.0.2	12	41	43	44	4
Cobertura-1.9.4.1	1	10	17	10	-
Castor-1.3.1	24	55	327	58	5
ActionBarSherlock	-	43	3	85	-
Colt-1.2.0	16	2	-	4	4
Gnattproject-2.0.9	-	19	23	56	-

God Class and Data Class have both been the subject of a lot of previous research. The Brain Class and Schizofrenic Class have received the most attention in this study.

For method level 4 projects have been considered among the listed java systems. The code smells detected on method level are:

- *Brain Method*: Brain Methods are the code smells for concentrating a class's intelligence. It encapsulates a class's functionality in a single method. A method should avoid extremes in size. The issue with Brain Methods is that they are very extensive, making them difficult to understand and debug, as well as almost impossible to reuse.

- *Shotgun Surgery*: It is a concept where appending one method or class require changes to be made in coupled methods or classes. Because of this, it's simple to overlook important method adjustments, which makes system maintenance tough.

Table 2: Method level code smells in selected Java Systems

Java projects	Brain Method	Shotgun Surgery
Freemind-0.9.0	34	98
Argouml-0.34	136	277
Sunflow-0.07.2	39	70
Retrofit	1	1

2. Creation of Dataset:

For each code smell type, a dataset has been created: two datasets for class-level smells (Brain Class and Schizofrenic Class) and two for method-level smells (Brain Method and Shotgun Surgery). In each dataset, each row represents class or method instances, and has one attribute for each metric. The labeled dataset at class level contains 8,066 instances and 53 columns whereas that at method level contains 10,097 instances and 47 columns.

The dataset can be accessed here: [Datasets]

After the creation of the datasets, it was observed that some of the metric column contained >90% zeroes. Thus, cleaning was done on all the datasets. This reduced the dimension of the datasets in the following way:

Table 3: Dimension of datasets created

Datasets	Dimension (rows * columns)
brain_class.csv	8026 * 47
schizophrenic_class.csv	8026 * 47
brain_method.csv	8872 * 45
shotgun_surgery.csv	8803*44

3. Applying Machine Learning Algorithm to the dataset:

We experimented our approach by selecting a set of suitable Machine Learning Algorithms and testing them on the generated datasets, by means of 10-fold cross validation using WEKA[6] software.

[3] https://github.com/JakeWharton/ActionBarSherlock
[4] https://github.com/mockito/mockito
[5] https://github.com/square/retrofit
[6]https://sourceforge.net/projects/weka/

In the set of supervised classification algorithms we selected, we can find algorithms used in the literature for software engineering tasks and algorithms able to produce human-readable models. Moreover, the set of selected algorithms covers many different machine learning approaches, i.e., decision trees, rule learners, support vector machines, Bayesian networks. The algorithms' implementation is the one available in Weka (Hall et al. 2009). In the following, we report the list of the selected algorithms, with a short description:

- J48 is an implementation of the C4.5 decision tree. This algorithm produces human understandable rules for the classification of new instances. The Weka implementation offers three different approaches to compute the decision tree, based on the types of pruning techniques: pruned, unpruned and reduced error pruning. In the experimentation, pruned type was considered.

- JRip is an implementation of a propositional rule learner. It is able to produce simple propositional logic rules for the classification, which are understandable to humans and can be simply translated into logic programming.

- Random Forest is a classifier that builds many classification trees as a forest of random decision trees, each one using a specific subset of the input features. Each tree gives a classification (vote) and the algorithm chooses the classification having most votes among all the trees in the forest.

- Naïve Bayes is the simplest probabilistic classifier based on applying Bayes' theorem. It makes strong assumptions on the input: the features are considered conditionally independent among each other.

The cross-validation experiment has the goal of finding the best algorithm, in its best setup, with respect to some performance measures. We applied three standard performance measures: accuracy, F-Measure, and Area Under ROC. These three measures express different point of views of the performances of a predictive model, and use a single number to express a whole performance measure. This is in contrast, e.g., with precision and recall, which must be considered together to better understand the performances of a model.

The following is a brief summary of these estimators:

- **Accuracy:** Accuracy is the ratio of number of correct predictions to the total number of input samples.

$$\text{Accuracy}=\frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions\ made}$$

  This performance index is one of the simplest performance measures for a classification task. Usually, accuracy is never reported alone because, when the positive and negative classes are unbalanced (small number of positive instances and a large number of negative instances) the result undergoes a severe distortion. That never occurs in our experiments, because all datasets are balanced.

- **F-measure:** It is the Harmonic Mean between precision and recall. The range of F-measure is [0,1]. It gives the precision and robustness of your model's classifier. High accuracy but low recall offers you an incredibly accurate result, but it also misses a big number of difficult-to-classify cases. The higher the F-measure, the better our model's performance. Mathematically, it is expressed by:

$$\text{F-measure}=2*\frac{1}{\frac{1}{precision}+\frac{1}{recall}}$$

- **Precision:** It is the ratio of number of correct positive results by the number of positive results predicted by the classifier.

$$\text{Precision}=\frac{True\ Positives}{True\ Positives+False\ Positives}$$

- **Recall:** It is the ratio of number of correct positive results by the number of ***all*** relevant samples where all the samples have been identified positive.

$$\text{Recall} = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

- **Area under ROC (Receiver Operating Characteristic):** The last performance index is the area under ROC. The ROC chart shows the true positive rate versus the false positive rate as the classifier's discrimination threshold is changed. While the discrimination performs well, the area under the ROC approaches 1; when the categorization performs poorly, the area under the ROC approaches 0.5.

RESULT OF EXPERIMENTATION

Every algorithm was applied in different configurations to each dataset, by means of 10- fold cross validation, and the performances obtained on each fold were recorded. Given below are the tables with recorded values.

Table 4: Brain Class cross-validation results

Classifier	Accuracy	F-measure	Area under ROC
JRip_BC	99.9887 %	1.000	1.000
J48-Pruned_BC	99.9662 %	1.000	0.997
Naïve Bayes_BC	94.296 %	0.970	0.986
Random Forest_BC	98.6924 %	0.993	1.000

All the selected algorithms show exceptional accuracy. J48 records the highest accuracy and hence performs better whereas Naïve Bayes, in comparison to other algorithms gives less accuracy.

Table 5: Schizofrenic Class cross-validation results

Classifier	Accuracy	F-measure	Area under ROC
JRip_SC	100 %	1.000	1.000
J48-Pruned_SC	100 %	1.000	1.000
NaïveBayes_SC	88.9983 %	0.939	0.860
Random Forest_SC	92.7361 %	0.962	1.000

JRip and J48, both perform equally good under the considered metrics. Random Forest records good accuracy. Lowest performance is observed in Naïve Bayes.

Table 6: Brain method cross-validation results

Classifier	Accuracy	F-measure	Area under ROC
JRip_BM	99.9546 %	1.000	1.000
J48-Pruned_BM	100 %	1.000	1.000
NaïveBayes_BM	85.8441 %	0.920	0.967
Random Forest_BM	99.5456 %	0.998	1.000

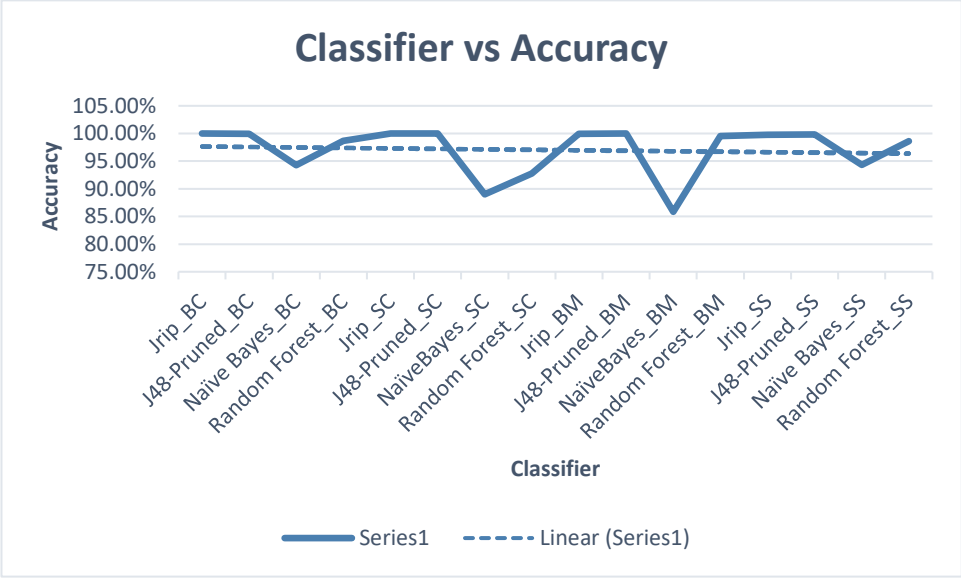
JRip records the maximum accuracy. J48 and Random Forest perform at par, J48 tends to record slight rise in its accuracy. Naïve Bayes records the lowest accuracy.

Table 7: Shotgun surgery cross-validation results

Classifier	Accuracy	F-measure	Area under ROC
JRip_SS	99.7882 %	0.999	0.978
J48-Pruned_SS	99.838 %	0.999	0.986
Naïve Bayes_SS	94.306 %	0.970	0.960
Random Forest_SS	98.6295 %	0.993	0.999

J48 records the maximum accuracy whereas NaïveBayes has the minimum accuracy in comparison to other algorithms.

In a nut shell, J48 and JRip has the best performance considering all the estimators Accuracy, F-measure and Area under ROC. Random Forest performs excellent in all the datasets except Schizofrenic Class. NaïveBayes, being a probabilistic classifier shows relatively lower accuracy in all datasets.



Above graph shows the difference in accuracies of all classifiers. One can observe that the highest accuracy is achieved by JRip and J48-pruned in all categories (Brain Class, Schizofrenic Class, Brain Method, Shotgun Surgery) whereas the lowest accuracy is observed by NaïveBayes_BM which is in Brain Method.

THREATS TO VALIDITY

Incompleteness of the investigation, process imbalance, research distribution disparity, and inaccuracy in information gathering and synthesis could all affect the conclusions of this study. This section reviews the threats to validity to our study.

It is possible that during the formation of the dataset, some metrics were missed that might have been useful. Since this study relies on one tool iplasma for detection of code smell, the number of code smell detected might vary with some other tool.

The results of both the parts of this study --- Creation of Dataset and Analysing or testing the prevalence of code smells in the dataset, rely on a specific set of open-source Java projects that was selected and might not generalize to all projects. However, the analysis in all the four datasets was done with the help of the algorithms suggested in the study.

It is possible to have some unnoticed error in the metrics of Machine Learning Algorithms. However, the dataset was tested rigorously using WEKA-3.8.5 to minimise the risk of getting incorrect metrics as much as possible.

CONCLUSION AND FUTURE WORKS

In this study, code smell detection and analysis of the resulting dataset are discussed. Supervised Machine learning algorithms, that are able to evolve with new data, is taken into consideration.

All of the algorithms produced similar performances (see Tables 4, 5, 6 and 7). The performances were also good or very good with respect to the machine learning measures used (Accuracy, F-measure, ROC). This demonstrates that employing machine learning techniques to analyse the dataset created, is a fine decision, or at least it is for the four code smells that were analysed. The winning machine learning algorithms have significant differences with the remaining ones, with a medium effect size on average.

In the future developments, we would like to consider more metrics to the dataset and validate the same with the help of other tools in order to make it more accurate and precise. Including more Java systems in the code smell detection, thus generating the dataset considering those systems along with its associated metrics, would be another development in the future. Other future works would include the automation of the different processes discussed in the paper, in the form of a GUI application.

REFERENCES

D. Rat¸iu; Memoria: A Unified Meta-Model for Java and C++, Master thesis, "Politehnica" University of Timi¸soara, 2004.

Proceedings of the 20th international conference on automated software engineering. ACM, pp 214–223 Lanza M, Marinescu R (2006) Object-oriented metrics in practice. Springer, Heidelberg

Arcelli Fontana, Francesca, et al. "Comparing and experimenting machine learning techniques for code smell detection." *Empirical Software Engineering* 21.3 (2016): 1143-1191.

Fontana, Francesca Arcelli, et al. "An experience report on using code smells detection tools." *2011 IEEE fourth international conference on software testing, verification and validation workshops*. IEEE (2011).

Marinescu C, Marinescu R, Mihancea PF, Ratiu D, Wettel R (2005) *"iPlasma: an integrated platform for quality assessment of object-oriented design."* In: Proceedings of the 21st IEEE international conference on software maintenance. IEEE, pp 25–30

Paiva, Thanis, et al. "On the evaluation of code smells and detection tools." *Journal of Software Engineering Research and Development* 5.1 (2017): 1-28.

Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton and James Noble 'Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies' *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pp336–345, December 2010.

Nuñez-Varela, A.S., Pérez-Gonzalez, H.G., Martínez-Perez, F.E. and Soubervielle-Montalvo, C., 2017. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, *128*, pp.164-197.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T. and Figueiredo, E., 2010. A Review-based Comparative Study of Bad Smell Detection Tools.

Murphy-Hill E, Black A (2010) An interactive ambient visualization for code smells. In: Proceedings of the 5th international symposium on software visualization. ACM, pp 5–14

Zazworka N, Ackermann C (2010) *CodeVizard: a tool to aid the analysis of software evolution.* In: Proceedings of the 4th international symposium on empirical software engineering and measurement. ACM, article 63

Moha N, Gueheneuc Y, Duchien L, Le Meur A (2010) *DECOR: a method for the specification and detection of code and design smells.* Softw Eng IEEE Trans 36:20–36. doi: 10.1109/tse.2009.50

Johansson, Alfred. *"Ensemble approach to code smell identification: Evaluating ensemble machine learning techniques to identify code smells within a software system."* (2020).

Hall, Mark, et al. "The WEKA data mining software: an update." *ACM SIGKDD explorations newsletter* 11.1 (2009): 10-18.

Table 8 : List of Java Systems Selected

System	Source	LOC	NOP	NOC	NOM
Freemind-0.9.0	Qualitas Corpus	65,687	43	849	5788
argouml-0.34	Qualitas Corpus	284,934	125	2361	18,015
Sunflow-0.07.2	Qualitas Corpus	24,319	21	191	1447
Retrofit	GitHub Repository	6006	23	151	683
Apache Ant-1.8.4	Qualitas Corpus	119486	80	1172	13223
ArtOfIllusion-2.8.1	Qualitas Corpus	136,533	22	799	688
Mockito	GitHub Repository	25758	103	648	3857
Fitjava-1.1	Qualitas Corpus	2453	2	60	254
c-jdbc-2.0.2	Qualitas Corpus	72408	101	543	5193
Cobertura-1.9.4.1	Qualitas Corpus	58,364	19	107	3309
Castor-1.3.1	Qualitas Corpus	213,479	149	1542	11,967
ActionBarSherlock	GitHub Repository	20108	24	158	2584
Colt-1.2.0	Qualitas Corpus	41872	25	297	3960
Gnattproject-2.0.9	Qualitas Corpus	58,718	54	959	5518

Table 9: Metrics and its abbreviations used in all datasets

Abbreviations	Metrics
ALD	Access to Local Data
ATFD	Access to Foreign Data
AMW	Average Method Weight
CC	Cyclomatic Complexity
ATFD	Access To Foreign Data
CDISP	Coupling Dispersion
DAC	Data Abstraction Coupling
DIT	Depth Inheritance Hierarchy
CINT	Coupling intensity
CM	Changing Methods
CYCLO	Average Cyclomatic Number
FANOUT	Number of Classes Called
FDP	Foreign Data Providers
FANOUT	Number of called classes
LOCC	Lines of Code in a Class;
FDP	Foreign Data Providers
NAS	Number of Associations per Class
LAA	Locality of Attribute Access.
LOC	Lines of Code
NOA	Number of Attributes
MAXNESTING	Maximum Nesting level
NOAV	Number Of Accessed Variables
NOAM	Number Of Accessor Methods
NOLV	Number of Local Variables
NOP	Number of Parameters
BUR	Base-Object Usage Ratio
CBO	Coupling Between Objects
CC	Changing Classes
CM	Changing Methods
NOD	Number of Descendants
NOM	Number of Methods
NOPA	Number of Public Attributes
NProtM	Number of Protected Members
TCC	Tight Class Cohesion
WMC	Weighted Methods per Class
WOC	Weight of a Class