Automating Orthogonal Defect Classification using Machine Learning Algorithms

Fábio Lopes†, João Agnelo†, César Teixeira†, Nuno Laranjeiro†, Jorge Bernardino*†
†CISUC, Department of Informatics Engineering
University of Coimbra, Portugal
fadcl@student.dei.uc.pt, jagnelo@student.dei.uc.pt, cteixei@dei.uc.pt, cnl@dei.uc.pt,
*Polytechnic of Coimbra, Portugal
jorge@isec.pt

Abstract

Software systems are increasingly being used in business or mission critical scenarios, where the presence of certain types of software defects, i.e., bugs, may result in catastrophic consequences (e.g., financial losses or even the loss of human lives). To deploy systems in which we can rely on, it is vital to understand the types of defects that tend to affect such systems. This allows developers to take proper action, such as adapting the development process or redirecting testing efforts (e.g., using a certain set of testing techniques, or focusing on certain parts of the system). Orthogonal Defect Classification (ODC) has emerged as a popular method for classifying bugs, but it requires one or more experts to categorize each bug in a quite complex and time-consuming process. In this paper, we evaluate the use of machine learning algorithms (k-Nearest Neighbors, Support Vector Machines, Naïve Bayes, Nearest Centroid and Random Forest) for automatic classification of software defects using ODC, based on unstructured textual bug reports. Experimental results reveal the difficulties in automatically classifying certain ODC attributes solely using reports, but also suggest that the overall classification accuracy may be improved in most of the cases, if larger datasets are used.

Index Terms

Software Defects, Bug Reports, Orthogonal Defect Classification, Machine Learning, Text Classification

I. Introduction

Software systems are nowadays at the core of most businesses, where they are used to support critical operations, usually under the form of services [1]. In general, the size and complexity of critical software systems has grown throughout the years and the tendency is that this complexity and their importance will continue to increase. In this context, the presence of a software defect (i.e., a bug) may result in serious failures [2] that have several consequences on the business (e.g., lost business transactions, reputation losses, or even severe safety problems).

The development of dependable systems requires that developers carry out bug detection activities (e.g., code inspections, static analysis, black-box testing) where it is vital to detect bugs early in the process, especially before production, otherwise the impact on product cost can become quite high [3]. However, from a software engineering perspective, what really makes the whole process more efficient is the execution of preventive actions. As it is not possible to have preventive actions for all different bugs, it is better to understand their main types and then perform root cause analysis on the patterns found [4]. This allows for adjustments in the software engineering methodology, allowing to put more or less effort on certain bug detection techniques, changing the development methodology (e.g., to use test-driven development), or redistributing team effort, just to name a few.

Orthogonal Defect Classification (ODC) is a very popular framework for classifying software defects [4]. The main problem associated with it is that it requires the presence of one or more experts, the classification process is relatively complex and particularly lengthy, and does not fit well with current time-to-market pressure put on developers. Previous works have tried to automate the classification process (e.g., [5]–[7]), but most of them are limited to one or two ODC attributes (e.g., Defect Type, [6], [7], Impact [5], [8]), require the addition of further information to bug reports, or end up using a simpler classification, which is placed on top of ODC, and where higher level families serve to group ODC attributes, which impacts the usefulness of ODC.

In this paper, we evaluate the applicability of a set of popular machine learning algorithms for performing automatic classification of software defects purely based on ODC and using unstructured text bug reports as input. The classifiers considered were: k-Nearest Neighbors (k-NN), Support Vector Machines (SVM), Naïve Bayes (NB), Nearest Centroid (NC), and Random Forest (RF). We illustrate our approach using a set of 4096 bug reports referring to 'closed and resolved' bugs that were extracted from the issue tracking systems of three popular NoSQL systems: MongoDB, Cassandra, and HBase. At the time of writing, this number of bugs represents about one fourth of all 'closed and resolved' bugs for these systems and were previously manually classified (and partially verified) by a researcher according to ODC. Two additional independent classifications were also produced to understand the reliability of the dataset.

The experimental results show the difficulties in automatically classifying bugs by solely using bug reports and disregarding the actual changes in the code. This is evident in some ODC attributes, such as Defect Type and Qualifier, which are highly related to the code corrections associated with the bug report. On the other hand, other attributes (e.g., Target) show high accuracy values as they do not depend on the actions taken by the developer to correct the bug. It also became obvious that, with exceptions, a relatively small dataset (which typically carries low number of occurrences of certain ODC classes) negatively impacts the overall results. Results suggest that, in general, larger and richer datasets may be very helpful in effectively automating the whole classification procedure.

We must emphasize that we are making our dataset publicly available and are also providing a REST service for performing automatic classification of bugs according to ODC. The service receives, as input, an unstructured textual description of the bug (i.e., a bug report) and the machine learning algorithm to use. The dataset, Swagger API description, and REST service are available at *http://odc.dei.uc.pt*.

In summary, the main contributions of this paper are the following:

- A comprehensive evaluation of machine learning algorithms for automatic classification of bug reports using ODC;
- A machine learning system able to tackle the majority of ODC attributes, namely Activity, Trigger, Impact, Target, Defect Type, and Qualifier;
- The public availability of our dataset holding 4096 ODC annotated bug reports;
- A RESTful service allowing developers to effortlessly classify a bug report according to ODC.

The rest of this paper is organized as follows. Section II presents background concepts and related work on papers using ODC, with emphasis on research trying to automate bug classification using machine learning algorithms. Section III presents the study designed to evaluate the applicability of machine learning algorithms to ODC and Section IV discusses the results. Section V presents the threats to the validity of this work and finally Section VI concludes this paper.

II. Background and Related Work

Orthogonal Defect Classification is an analytical process used in software development and test process analysis to characterize software defects [9]. It allows extracting valuable information from defects and provides insights and aids diagnosis in software engineering processes. ODC is based on the definition of eight attributes grouped into two sections: open-report and closed-report. The *open-report* section refers to attributes for which there should be information at the moment of disclosure of a certain defect, in particular:

- Activity: refers to the activity being performed at the time the defect was disclosed (e.g., unit testing);
- Trigger: indicates what caused the defect to manifest (e.g., a blocked test);
- Impact: is the impact a user experienced when the defect surfaced.

The *closed-report* section includes the attributes that relate with the correction of a certain defect (and depend on that information). In particular:

- Target: the object that was the target of the correction (e.g., a set of build scripts);
- Defect Type: the type of change that was performed to correct the problem (e.g., changing a checking condition);
- Qualifier: describes the code point before before the correction was made (e.g., missing, incorrect, or extraneous);
- Age: the moment in which the defect was introduced (e.g., during the correction of another defect);
- Source: refers to the origin of the defect, e.g., if it was in an outsourced code are, in in-house developed code.

Several works have used ODC to classify sets of software defects to understand the distribution of the types of bugs found. Works such as [10] and [11] propose workflow strategies consisting of multiple steps which aim to improve the effectiveness of software development processes. In the midst of these steps, the authors include the use of ODC to classify bugs found in the software product being developed. The final steps are mostly focused on extracting knowledge and lessons learned from the whole process, mostly based on defect distributions according to ODC attributes. Some other works use the knowledge obtained from using ODC in rather more empirical settings (e.g., fault injection experiments), such as [12] and [13]. Christmansson and Chillarege ([12]) inject errors instead of faults (i.e., an error is the consequence of a fault [14]) so as to skip the unknown waiting period needed for a certain fault to be activated. This requires some form of mapping between the set of faults that should be injected and their respective errors, which is obtained via using ODC classified bugs. Durães and Madeira [13] use the ODC classification distributions from a set of software bugs to categorize common programmer mistakes. The emerging patterns among these common mistakes are then used to define a set of software fault emulation operators.

A common problem present in works that use ODC is that the bug classification is usually a time-consuming manual process that requires the presence of one (or ideally more than one) expert. In addition, the number of defects analyzed should be relatively large to enable further analysis, which aggravates the problem. This kind of task is not only time- and resource-consuming, but also error prone. Therefore, there is an obvious need to automate this process, either via rule-based approaches (e.g., as proposed in [15]) or through the use of machine learning algorithms.

In [16] the authors describe an automatic defect classification scheme based on feeding the Abstract Syntax Trees of the buggy code and its respective correction to a decision engine. This decision engine can classify defects according to four categories: i) Data; (ii) Computational; ii) Interface; and iv) Control/Logic. With a dataset of 1174 defects extracted from

defect-tracking repositories of three separate systems, the automatic defect classification scheme was able to classify defects with an accuracy ranging from 64% to 79%. Given the relatively low number of classes automating such procedure tends to be an easier task (i.e., when compared with a more complex scheme, such as ODC), but, at the same time, less informative.

In [6], the authors propose an SVM-based automatic defect classification scheme. This scheme classifies defects into three families that essentially aggregate ODC attributes: i) control and data flow; ii) structural; and iii) non-functional. The classification process applies a text mining solution, which is able to extract useful information from defect report descriptions and source code fixes, and uses this information as the main criteria to classify defects. To validate the proposal, 500 software defects were classified both manually by the authors and automatically by the resulting SVM algorithm. The authors reported being able to achieve a global accuracy of 77.8%. The fact that families of ODC attributes are used is a way of diminishing the error, which may be interesting in certain scenarios, but, at the same time, providing an automatic approach based purely in ODC is a quite different endeavour, as the diversity of attributes (and respective values) is much larger.

The authors of [7] propose an active learning and semi-supervised defect classification approach that targets the families of defects proposed in [6]. The approach resorts to minimal labeled data that includes diverse and informative examples (i.e., active learning) and uses both labeled and unlabeled defect examples in the learning process (i.e., semi-supervised learning). The main goal of this approach is to reduce the amount of time spent in categorizing defects to train machine learning algorithms. The authors validated the proposal by labelling only 50 ODC defects out of a set of 500 (unlabeled), and achieved a weighted precision of 65.1% (applied to the three mentioned families only and based on the special selection of the minimal set of defects).

In [5], the authors describe AutoODC, which is an approach for automatically classifying software defects according to ODC and that views the case as a supervised text classification problem. AutoODC integrates ODC experience with domain knowledge and resorts to two text classification algorithms, namely Naïve Bayes and Support Vector Machines. The authors introduce the Relevance Annotation Framework, in which a person registers the reason why they believe a given defect should belong to a particular ODC attribute value, thus providing further information to the machine learning algorithm. The authors state that this additional approach is able to reduce the relative error of the classification by a factor between 14% and 29%. The accuracy of the proposed approach reaches 82.9% using the NB classifier and 80.7% using the SVM classifier on a total of 1653 industrial defect reports, and 77.5% and 75.2% using the NB and SVM classifiers respectively, on a set of open-source defect reports. Note, however that, trying to automate ODC exclusively relying on the typical information present in a bug report is a much harder task and, at the same time, a more realistic scenario.

Some works use machine learning algorithms to automate the process of bug triage, i.e., the process of deciding what to do with newly found software defects. These approaches do not focus on the classification of the defects in terms of their causes or corrective measures, but typically on properties like severity level, so that defects can be separated into specific categories, allowing developers to, for instance, prioritize their corrections. The works [17]–[20] apply this approach to automatically assign new bugs to specific developers. To this end, the authors of [17] propose the use of a supervised Bayesian machine learning algorithm. The algorithm correctly assigns 30% of bug reports to the expected developers. Xuan et al. [18] describe a semi-supervised NB text classification algorithm which uses ’expectation-maximization’ to take advantage of both categorized and uncategorized bug reports. The algorithm iteratively categorizes unlabeled data (which refers to the expectation step) and rebuilds a new classifier (which refers to the maximization step). This semi-supervised approach increases the overall accuracy by up to 6% when compared with its supervised counterpart.

Tian et al. [19] propose a framework, named DRONE, for predicting the severity of a bug based on a set of factors (i.e., temporal, textual, author, related-report, severity, and product) that are extracted from bug reports via feature extraction. The framework includes a classification engine named ThresholdinG and Linear Regression to ClAssifY Imbalanced Data (GRAY), which tries to enhance linear regression with a thresholding approach so that cases of imbalanced bug reports are handled. The approach is applied against a dataset of over 100.000 bug reports from Eclipse and it shows an improvement of 58.61% compared to results in Menzies and Marcus [21]. With the same bug triage goal, Dedík [20] proposes an algorithm based on SVM with Term Frequency–Inverse Document Frequency (TF–IDF). The achieved accuracy is of 57% for a dataset of Firefox bug reports, 53% for a Netbeans dataset and 53% for a defect dataset from a private company.

Finally, Hernández-González et al. [8] present an approach which focuses on automating the classification of bugs according to the ODC Impact attribute. The authors name this approach "learning from crowds" (it relies on the amount of data rather than quality), and it focuses on training a classifier with a large amount of potentially erroneous and noisy classifications from people with no particular expertise in ODC classification (i.e., partially reliable annotators). The work used the data produced by a group of 5 novice annotators, which classified 846 plus 598 bugs from the Compendium and Mozilla Firefox open-source projects, respectively. The authors resort to a Bayesian network classifier and conclude that learning from (unskilled) crowds can outperform the typical approach of using a reliable dataset to train the classifier. Still, the authors have used only 4 of the 13 values of the Impact attribute (Installability, Requirements, Usability, Other), reaching the whole 13 values with satisfactory accuracy is a complex task.

In summary, considering the prior developments on automatic defect classification, we essentially find two groups of works. One group uses machine learning to perform bug triage (including assigning bugs to developers and distinguishing severity). The second group specializes on ODC, but the different works either *focus on a single ODC attribute* (sometimes using a

reduced set of values for the single attribute), or they *focus in families of attributes*, where several attributes are used to form higher-level groups. From the perspective of ODC these kinds of approaches are less useful, as we are trading the information richness that is strictly associated with the detailed ODC process with the information that can be obtained from a higher granularity classification (which is less informative), although it potentially allows accuracy metrics to reach higher values.

In this work, we opted to not discard any attributes nor consider families of attributes. In this sense, this work diverges from related work and results cannot be directly compared using the typical metrics (e.g., accuracy). These two particular choices create greater difficulties to the machine learning algorithms, as: i) there is no additional information for each defect, other than what already exists in reality; ii) the number of attributes (and classes, i.e., attribute values) are higher, thus the chance of error tends to be higher. Despite the additional difficulties, we consider that this case represents reality in a more accurate way and allows developers and researchers to obtain full benefits of applying ODC.

III. Study Design

In this section, we describe the design of our study, which is based on the following set of steps:

1) Manual creation and preprocessing of the dataset;
2) Feature extraction;
3) Dimensionality reduction;
4) Configuration and execution of the machine learning classifiers;
5) Assessment of the classifiers performance.

The following subsections describe each of the above steps in detail, explaining the decisions taken throughout the work.

A. Dataset creation and preprocessing

The creation and initial preprocessing of the dataset (step 1 of the study) used in this work involved going through the following set of substeps, which we describe in the next paragraphs:

i. Selection of a set of NoSQL databases;
ii. Selection of a set of defect descriptions from each one of the databases online bug tracking platforms;
iii. Manual ODC classification of each software bug, carried out by one researcher (named *researcher1*);
iv. Manual verification of the bugs classified in the previous step in two different verification activities:
   a) Internal verification carried out by the same researcher responsible for the original classification;
   b) External verification produced by two external researchers (named *researcher2* and *researcher3*).

We started by *selecting three popular NoSQL databases*, MongoDB, Cassandra, and HBase (selected according to popularity rankings in db-engines.com, stackoverkill.com, and kdnuggets.com). One ODC-knowledgeable early stage researcher, named *researcher1* was selected to carry out the manual process of classifying 'closed and resolved' bug reports, according to ODC. *researcher1* first trained with a total of 300 bug descriptions (100 bug reports per each NoSQL database). During this process, and initially in an informal way, a few samples of the outcome of the classification were verified and discussed by two experienced researchers as a way of providing feedback to *researcher1*. These 300 bug descriptions were then discarded and, thus, are not part of the final dataset.

We then *randomly selected 4096 'closed and resolved' bugs* (1618, 1095 and 1383 bugs for MongoDB, Cassandra, and Hbase, respectively), which, at time of writing, accounted for around one fourth of the total closed and resolved defects in each database. We divided the 4096 bugs dataset in five batches, with the first four batches holding 2048 bugs (512 bugs each batch). *researcher1* then initiated with the *manual classification of each bug*.

The bug reports used are composed of a title, a description of the detected bug and several comments that end up describing what has been made to correct the bug. This information was interpreted by *researcher1* and each bug was manually labeled using the six different labels, one for each ODC attribute: Activity, Trigger, Impact, Target, Defect Type, and Qualifier. We excluded the 'Age' and 'Source' ODC attributes from this work, as the bug tracking systems of the three systems under analysis did not provide sufficient information to classify these attributes for all the bugs in our dataset.

To *verify the classification* of the bugs and, most of all, to have some insight regarding the overall quality of the annotated dataset, we performed two verification activities against 40% of the bugs (1640 out of 4096 bugs), which we name Internal and External and that are depicted in Figure 1.

During the *internal verification* activity, *researcher1* progressively re-inspected a total of 20% of the bugs (820 of the 4096 bugs) to check and correct any errors found. Thus, the internal verification was performed over the first four batches and immediately after the classification of a certain batch was concluded. In practice, by the end of each batch, the researcher double-checked a total of 40% of the already classified defects in that batch, using the following distributions: i) a sequential selection of the first 20% of the bugs in the batch; ii) a random selection of another 20% from the remaining bugs in the batch. After this was concluded and the dataset improved, we initiated the *external verification* activity, which involved two new and independent classifications carried out by two external researchers (designated *researcher2* and *researcher3*). These

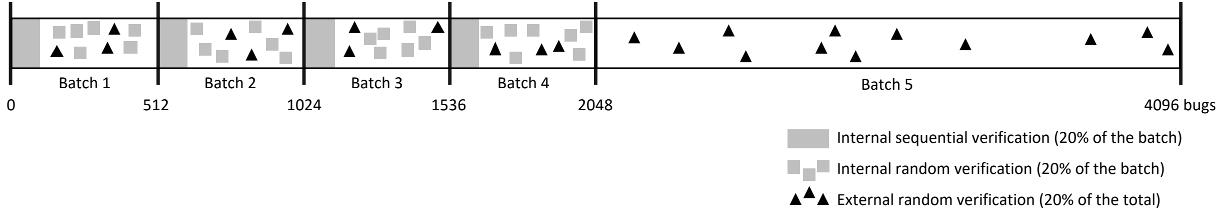


Fig. 1: Representation of the different verification tasks across the dataset.

two researchers were provided with two sets of randomly selected and non-overlapping bugs, which added up to 20% of the global dataset (i.e., 820 bugs in total, 410 bugs per each one of the two researchers).

Table I shows the detailed outcome of the verification procedure, which is composed of the previously mentioned verification tasks, named Internal (carried out by researcher1), External1 (carried out by researcher2) and External2 (carried out by researcher3). The table shows the details only for the Defect Type attribute, due to the fact that it is the most widely used ODC attribute in related work (e.g., [22]–[24]). Notice that the bug counts for Internal, External 1, and External 2 are below the 20% and 10% marks (the exact numbers are 746, 379, and 382 bugs, respectively), as we are considering only the bugs where the Target attribute was confirmed to be 'Code' and, as such, can be marked with a defect type.

In each of the matrices, each cell holds the total number of bugs marked with a certain attribute value, the values read in the rows represent the outcome of the verification procedure. In light blue, we highlight the true positives (the bugs in which there is agreement between the original classification and the respective verification task).

TABLE I: Detailed view of the verification tasks results.

	Defect Type	A/I	C	A/M	F/C/O	T/S	I/OOM	R
Internal (746 bugs)	Assignment/Initialization (A/I)	34		4				
	Checking (C)		71	6				
	Algorithm/Method (A/M)	4	4	394	13	1	4	
	Function/Class/Object (F/C/O)			6	126		1	1
	Timing/Serialization (T/S)			1		4		
	Interface/O-O Messages (I/OOM)	1		3	1		65	1
	Relationship (R)							1
External 1 (379 bugs)	Assignment/Initialization	20		2				
	Checking		41	3				
	Algorithm/Method		2	216	5			
	Function/Class/Object			3	44			
	Timing/Serialization			1		6		
	Interface/O-O Messages		2	1	3		29	
	Relationship							1
External 2 (382 bugs)	Assignment/Initialization	14		8			9	
	Checking		34	8				
	Algorithm/Method	5		219				
	Function/Class/Object			8	51			
	Timing/Serialization					2		
	Interface/O-O Messages			2			22	
	Relationship							

We also analyzed the inter-rater agreement using Cohen's Kappa (k), which is able to measure the agreement between two raters that classify items in mutually exclusive categories [25]. In our case, we wanted to understand the agreement of *researcher2* (and *researcher3*) with *researcher1*. The definition of k is given by Equation 1:

$$k = \frac{p_o - p_c}{1 - p_c} \tag{1}$$

where p_o is the relative observed agreement between raters (i.e., accuracy) and p_c is the probability of agreement by chance. If both raters fully agree, then $k = 1$, if there is no agreement beyond what is expected by chance, then $k = 0$, and, finally, a negative value reflects the cases where agreement is actually worse than random choice. The following terms are typically used for the quantitative values of k: *poor* when $k < 0$, *slight* when $0 <= k <= 0.2$, *fair* when $0.21 <= k <= 0.40$, *moderate* when $0.41 <= k <= 0.60$, *substantial* when $0.61 <= k <= 0.80$, and finally *almost perfect* when $0.81 <= k <= 0.99$ [26]. The accuracy results of researcher2 are presented in Table II, along with the Kappa value (referring to the agreement with *researcher1*).

Table II shows that we obtained almost perfect agreements for the majority of the ODC attributes. The exception is Activity with substantial agreement and Trigger with moderate agreement. In the former case, the most visible case of disagreement between the researchers is the case where code inspection bugs were marked by *researcher2* as being one of the three possible

TABLE II: Accuracy and Cohen's Kappa Agreement for researcher2 external verification

ODC Attribute	Accuracy	k	Agreement
Activity	0.71	0.64	substantial
Trigger	0.61	0.59	moderate
Impact	0.82	0.81	almost perfect
Target	0.97	0.96	almost perfect
Defect Type	0.94	0.93	almost perfect
Qualifier	0.92	0.88	almost perfect

Testing cases (i.e., unit testing, function testing, system testing). Indeed, researcher2 classified roughly one third of the 229 bugs originally marked with code inspection by *researcher1* as being one of the three Testing activities (15 bugs attributed to unit testing, 30 to function testing, and 32 to system testing). This has direct impact in the Trigger classification, as, in this case, a wrong activity will lead to a wrong Trigger (this is due to the fact that some activities exclusively map into certain triggers). As such, we can expect that the accuracy values for Trigger are lower, which is actually the case. Due to time restrictions, *researcher3* only classified the defect type attribute, for which it reached an accuracy of 0.90 which corresponds to a Kappa value of 0.88, also achieving an almost perfect agreement. Based on this, we consider that the whole dataset is reliable, as we estimate the overall error is kept under similar values.

Table III holds a summary of the final outcome of the whole classification process and shows the details regarding the number of reports assigned to each label. As the Triggers directly map to the 5 Activities (i.e., Design Review/Code Inspection, Unit Test, Function Test, System Test), we also include the detailed values for the respective triggers. There is also a relation between the Defect Type and Qualifier attributes with the Target attribute, which only apply to reports classified with Target Code or Design (which means that not all of the 4096 defects qualify for being marked with a defect type / qualifier, in our case - 3846 defects are code defects). During the process, we had to exclude some classes (marked in gray in Table III) due to their absence or relatively low number of instances in the dataset. We also identify (light blue cells) the smallest number of reports per attribute in Table III (explained in the next paragraph).

TABLE III: Number of documents per class in the dataset. Gray cells indicate removed classes due to small or nonexistent instances. Light blue cells indicate the small number of instances per attribute.

	Attribute	Classes			
Opener Section	**Activity**	Design Review	1	System Test	658
		Code Inspection	2128	Unit Test	483
		Function Test	826	—	
	Trigger - Code Inspection	Backward Compatibility	86	Internal Document	26
		Concurrency	210	Language Dependency	63
		Design Conformance	273	Lateral Compatibility	85
		Logic/Flow	960	Rare Situation	14
		Side Effects	503	—	
	Trigger - Function Test	Coverage	535	Variation	233
		Sequencing	52	Interaction	6
	Trigger - System Test	Blocked Test	352	Software Configuration	88
		Recovery/Exception	26	Startup/Restart	68
		Workload/Stress	116	Hardware Configuration	7
	Trigger - Unit	Simple Path	413	Complex Path	70
	Impact	Capability	2165	Standards	98
		Integrity/Security	187	Usability	45
		Performance	199	Accessibility	0
		Reliability	988	Documentation	1
		Requirements	227	Installability	14
		Serviceability	156	Maintenance	1
		Migration	15	—	
Closer Section	**Target**	Code	3846	Information Development	20
		Build/Package	230	Requirements	0
		Design	0	National Language	0
	Defect Type	Algorithm/Method	2365	Function/Class/Object	543
		Assignment/Initializatio	169	Interface O-O Messages	312
		Checking	407	Relationship	8
		Timing/Serialization	42	—	
	Qualifier	Extraneous	158	Missing	1159
		Incorrect	2529	—	

After observing the classified bug reports in detail, we noticed that the dataset was very imbalanced. The classifiers chosen (SVM, k-NN, NB, NC and RF) tend to classify a certain report using the most prevalent class if the dataset is imbalanced. An obvious solution is to use a sampling technique to balance the dataset. We decided to use an undersampling technique rather

than oversampling because, in this context, oversampling would imply artificially generating bug reports (including mimicking conversations between developers), which is a very complex task and very prone to errors.

Undersampling consists of decreasing the number of reports so that their number will be the same for all classes. Thus, the class with less reports (marked in light blue in Table III) keep all information while in the others, reports are randomly removed until their number reaches the number of reports in the less popular classes.

B. Feature Extraction

The goal of this step is to extract representative values from the original text reports that are supposed to be informative and non-redundant (as a way to help the machine learning algorithms). Several feature extraction techniques, described in the next paragraphs, can be applied to non-structured text data, namely: *bag-of-words* [27]; term frequency (TF) [27]; term frequency combined with inverse document frequency (TF-IDF) [27]–[29].

The *bag-of-words* technique, where each word counts as 1 when it appears in a document and 0 otherwise, does not allow to distinguish uncommon words from words that appear in the entire dataset, since all of them would carry the same value.

The TF technique, where an array of numbers represents how many times each word appears in a document, gives a higher value to the words that appear most often in a document. This may be problematic, given that the words that appear most often in a document are usually words which are widely used in a given context. Therefore, the classifiers would very likely show poor performance since all the classes would be defined by the same best words.

The TF-IDF computes a weight that discriminates the occurrence frequency of a given word in the entire dataset and allows us to understand its importance for a certain document. The TF part counts how many times it appears in the document and the IDF part counts how many times it appears in the entire dataset. A higher number of occurrences of a word in the entire dataset decreases the word's importance for the classification problem [28][29]. In this work we use TF-IDF.

The IDF value for a certain term t in a dataset D is given by Equation 2:

$$\text{IDF}(t, D) = \log_{10}\left(\frac{N}{|\{d \in D : t \in d\}|}\right). \tag{2}$$

Where N is the number of documents in the data set D, and the denominator is the number of documents where the given term t appears. If the denominator is 0 it is added one unit to make the division possible. When we multiply the term counter values with the IDF values we obtain a value (TF-IDF) that allow us to know the importance of a word in each document. The words that are the most important are those that have larger values, i.e. those that appear more times in each document and less times in the entire data set. The TF-IDF is given by:

$$\text{TF-IDF}(t, d, D) = \text{TF}(t, d) \times \text{IDF}(t, D). \tag{3}$$

Where $\text{TF}(t, d)$ represents the number of times that term t appears in a document. Therefore, at the end each report is represented by a vector $\mathbf{x}$ with all the values that resulted from the application of TF-IDF.

The classification process included training (using part of the dataset) and testing (the execution of the classifiers using unknown data). Thus, we divided the dataset in these two parts (see Section III-E for further details). For performing feature extraction over the bug reports that belong to the training dataset, a dictionary was developed with all the words that appeared in the set and their respective number of occurrences. For this purpose, we relied on text feature extraction functions available in Scikit-learn: Machine Learning in Python [30]. For the testing set, the dictionary built with the training set was used to convert words into numbers using Equations 2 and 3. Although, the majority of state-of-art text classification models remove the words that appear too frequently, called stop words, we decided to keep them since these type of words are not useless in our case [5].

C. Dimensionality Reduction

Principal Component Analysis (PCA) and Linear Discriminant Analysis (LDA) are likely the most used Dimensionality Reduction methods in Machine Learning. On one hand, PCA is a non-supervised approach that tries to project features into a low-dimensional space, preserving the data variance as most as possible. On the other hand, LDA is supervised and the goal is to find a projection that maximizes data clusters separability and compactness [31].

Due to its simplicity, PCA was used to diminish the ratio between the number of samples and the number of features, known as Dimensionality Ratio [32]. High dimensionality ratios reduce the risk of overfitting (training data specialization), i.e., tend to improve the capacity to classify correctly new data. For this purpose, it was necessary to select a dimensionality ratio that would allows us to keep as much information as possible, while keeping the abovementioned ratio higher than one (values below one mean that we have more features than samples and accuracy in new data tend to be very low). In the case of this work, the value that we used was half of the number of documents in the balanced dataset for each attribute. This leads to a dimensionality ratio of two, preserving around 70% of the data variance for all ODC categories, which is pointed as the minimum value of preserved variance in the literature [33]. After applying PCA for Dimensionality Reduction, we obtained new vectors $\mathbf{x}$ with new features created based on the old best features, i.e, based on the best discriminative words.

D. Classifiers

Five classifiers that fit into five different types were considered: the k-Nearest Neighbor classifier (k-NN), known to be a lazy learner; a Bayesian classifier (Naïve Bayes), i.e., a parametric approach; the Support Vector Machine (SVM), which is a maximal-margin classifier; the Random Forest (RF), an ensemble classifier; and the Nearest Centroid (NC), a minimum distance classifier.

The k-NN classifier is named lazy learner because its training is just the storage of labeled data [34]. A new pattern is classified by finding the k closest neighbors, and the class of the new pattern will be the most prevalent class among the k closest neighbors. Thus, the k parameter should be odd to not allow the occurrence of draws. The best k value is usually obtained by experimentation, as we did in this work. Concerning the determination of the k closest neighbors, we opted to use the Euclidean distance, which is known to be the most used metric.

A Bayesian classifier is based on the Bayes rule, which is given by:

$$P(\omega_i|\mathbf{x}) = \frac{P(\omega_i)p(\mathbf{x}|\omega_i)}{p(\mathbf{x})}. \tag{4}$$

where $\mathbf{x}$ is a feature vector corresponding to a given bug report that we want to classify, ω_i is the i-th class, $p(\mathbf{x}|\omega_i)$ is the conditional probability density function (*pdf*) that describes the occurrence of a given vector $\mathbf{x}$ when the class ω_i is known, $P(\omega_i)$ is the probability of occurrence of objects of class ω_i, $p(\mathbf{x})$ is the *pdf* that define the probability of occurrence of a given pattern $\mathbf{x}$ independently of the class, and $P(\omega_i|\mathbf{x})$ is the probability that knowing a pattern $\mathbf{x}$ its class is ω_i. Thus in a classification problem we assign the class that corresponds to the biggest $P(\omega_i|\mathbf{x})$ to $\mathbf{x}$. The classification rule is:

$$\mathbf{x} \in \omega_i \text{ if } P(\omega_i|\mathbf{x}) > P(\omega_j|\mathbf{x}) \; \forall i \neq j. \tag{5}$$

The training of a Bayesian classifier encompasses the determination of $P(\omega_i)$ and $p(\mathbf{x}|\omega_i)$ for all the classes. The former is estimated by computing the prevalence of patterns from a given class in the training set, for example, for a given class ω_i: $P(\omega_i) = n_i/N$, being N the total number of patterns in the training set and n_i the number of patterns that belong to ω_i. The latter involves the estimation of a *pdf* from data. Naïve Bayes (NB) is a particular case of the general Bayesian classifier, where independence between every pair of features is assumed, thus the estimation of $p(\mathbf{x}|\omega_i)$ is simply given by:

$$p(\mathbf{x}|\omega_i) \approx \prod_{k=1}^{F} p(x_k|\omega_i), \tag{6}$$

where x_k is a given value of the k-th feature and F the total number of features. Concerning the distribution type assumed for each $p(x_k|\omega_i)$, in this paper we considered the multinomial distribution, which is known to be a good choice for text classification [35].

In its native formulation SVM finds a decision hyperplane that maximizes the margin that separates two different classes [36]. Considering non-linearly separable training data pairs $\{\mathbf{x}_i, y_i\}$ (with $i = 1, ..., N$), where $y_i \in \{-1, 1\}$ are the class labels and $\mathbf{x}_i \in R^d$ are corresponding features vectors, the function that defines the hyperplane is:

$$y_i \left(\mathbf{w}^T \mathbf{x}_i + b\right) \geq 1 - \xi_i; \quad y_i \in \{-1, 1\} \tag{7}$$

where $\mathbf{w}$ is the vector normal to the hyperplane, $b/(||\mathbf{w}||)$ is the hyperplane offset to the origin and ξ is a quantification of the degree of misclassification. $\mathbf{w}$ and b are obtained by minimizing:

$$\Psi(\mathbf{w}) = \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{N} \xi_i, \;\; \text{subject to} \tag{8}$$
$$y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 - \xi_i; \quad i = 1, .., N.$$

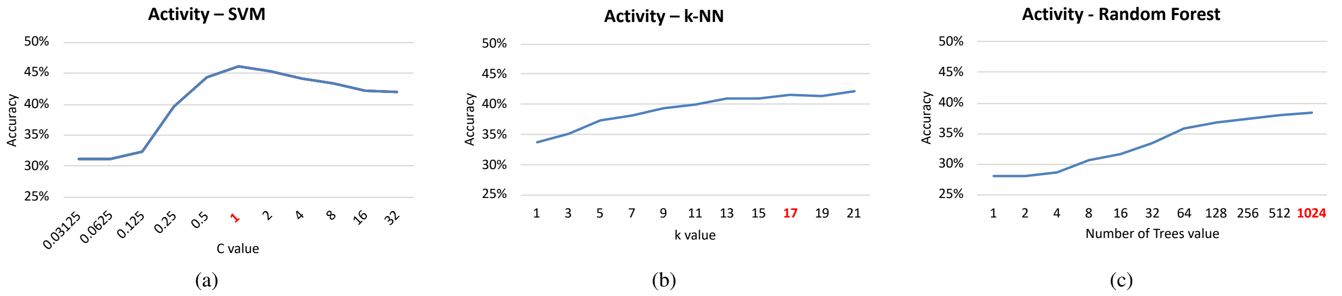
C defines the influence of ξ on the minimization criterion Ψ. The SVM is defined as a two-class classifier, but ODC classification is a multi-class problem. The usual approach for multi-class SVM classification is to use a combination of several binary SVM classifiers. Different methods exist but, in this work, we used the one-against-all multi-class approach [37]. This method transforms the multi-class problem, with c classes, into a series of c binary sub problems that can be solved by the binary SVM. The i-th classifier output function ρ_i is trained taking the examples from a given class as 1 and the examples from all other classes as -1. For a new example $\mathbf{x}$, this method assigns to $\mathbf{x}$ the class that were assigned in majority by all the binary classifiers [37].

The Random Forest classification is made by considering a set of random decision trees [38]. The trees training is usually performed with the "bagging" (Bootstrap aggregating) method [39], with the ambition that a combination of simple learning models increases the overall classification performance. The "random" designation comes from the fact that individual trees are based on a random selection of the available features. RF is less prone to overfitting than single decision trees because of its random nature. When compared to non-ensemble approaches, this type of algorithms tends to return more stable results, i.e., they present a reduced variance.

The first step in a NC classifier is the computation of the centroids based on a training dataset, i.e., the mean vector for each class. In the testing phase a new sample is classified by computing the distance between it and the centroids, related to the different classes, then the class of the sample will be the class of the nearest centroid. In the text classification domain using TF-IDF feature vectors NC is also known as the Rocchio classifier [27].

Configuring the classifiers mostly involves finding the best hyperparameters for our SVM, k-NN and RF. Thus, we performed a search, usually referred as Grid-Search, for the parameters that lead to the best classification, i.e, that leads to the best accuracy. The Bayes classifier and the NC have no free parameters and thus it was not necessary to perform this search. For SVM we searched for the C values, for k-NN we searched for the number of neighbors (k), and for RF for the appropriate number of trees.

An application of the Grid-Search method is represented in Figure 2, where the goal is to find C, k and the number of trees that provides high accuracy (the best values, in the case of the example shown in Figure 2, are marked in red). As presented in the Figure, we varied C in powers of two between 2^{-5} and 2^5, k assumed odd values between 1 and 21, and the number of trees tested were $[2^0, 2^1, \cdots, 2^{10}]$. For SVM, we should choose a C value that leads to high accuracies but that at the same time is small. Larger C values lead to the definition of smaller separation margins based on the training data and the probability of overfitting is high. On the other hand, for k-NN we have the reverse situation, i.e., the number of neighbors should not be so small in order to prevent overfitting. RF is known to be very resistant to overfitting, even with a higher number of trees. The number of trees can result in a problem of complexity, and we should use a good compromise between performance and complexity. Although Figure 2 only shows the values for the Activity attribute, the procedure was exactly the same for all other ODC attributes and therefore we omit the graphical view of the remaining searches (all detailed values and graphics may be found at *http://odc.dei.uc.pt*).



(a) (b) (c)

Fig. 2: Grid Search for hyperparameters using Activity

Table IV presents all the best values for C, k and number of trees. As previously mentioned, these values were selected by examining the best accuracies found with lower values of C, k and number of trees. We found out that the best C values were confined to the range $[2^{-2}, 2^4]$, the best k values to the range $[5, 21]$, and the best number of trees to the range $[2^7, 2^{10}]$.

TABLE IV: Parameters that lead to the best accuracy for each classifier and ODC attribute

	SVM	k-NN	Random Forest
ODC Attribute	***Best C***	***Best k***	***Best Number of Trees***
Activity	1	17	1024
Trigger - Code Inspection	2	17	512
Trigger - Function Test	2	5	512
Trigger - System Test	1	11	256
Trigger - Unit Test	0.25	19	128
Impact	1	21	512
Target	16	15	512
Defect Type	8	7	256
Qualifier	16	17	256

E. Classifier assessment

As mentioned earlier, the classification process includes training (using part of the dataset) and testing (using unknown data, also known as out-of-sample data). It is important to mention that, for training the classifiers related to 'open section' ODC attributes we just used the title and description of the bug. Thus, for these cases, we do not use the posterior developer comments, which many times discuss how a fix should be performed and would not fit the open section definition. For training the classifiers related with the 'close section' ODC attributes, we use all available information regarding each bug.

In order to validate our classifiers, we used the K-Fold Cross Validation (CV) approach, with ten folds (i.e., K=10) [40], which is known to be the most classic validation strategy. Supposing that the training data is composed by M bug reports (i.e.,

samples), the algorithm will split the data in 10 parts, where each part contains M/10 samples. Afterwards the classifiers are trained 10 times for the same data. In each iteration, the classifiers are trained with nine parts and the part that is “left out" is used for testing. The algorithm output is compared to the real labeling by filling confusion matrices. Different ODC attributes present different numbers of classes, thus, confusion matrices with different sizes were filled. These can be represented by the generic confusion matrix illustrated in Table V.

TABLE V: Generic confusion matrix, where “$\hat{c_k}$” indicates predictions provided by the algorithms

		Predicted			
		$\hat{c}_1$	$\hat{c}_2$	...	$\hat{c}_K$
True	c_1	$n_{1,1}$	$n_{1,2}$	...	$n_{1,K}$
	c_2	$n_{2,1}$	$n_{2,2}$	...	$n_{2,K}$
	⋮	...	...	...	...
	c_K	$n_{K,1}$	$n_{K,2}$	...	$n_{K,K}$

The diagonal terms $n_{i,j}$, where $i = j$, correspond to the instances where the algorithm's output was consistent with the real class label, i.e. the true positive patterns. The values $n_{i,j}$, where $i \neq j$, are the number of instances misclassified by the algorithm, which can be considered as false positives or false negatives depending on the class under analysis. Considering a given class k, four measurements can be taken:

- True positives (TP_k): number of instances correctly classified as class k;
- False positives (FP_k): number of instances classified as k when in fact they belong to other class;
- True negatives (TN_k): number of instances correctly not classified as k;
- False negatives (FN_k): number of instances assigned to other classes when in fact they belong to class k;

These measurements for a problem with K classes can be computed from the confusion matrix as follows:

$$\begin{aligned} n_{+,k} &= \sum_{i=1}^{K} n_{i,k}; \\ n_{k,+} &= \sum_{j=1}^{K} n_{k,j}; \\ TP_k &= n_{k,k}; \\ FP_k &= n_{+,k} - n_{k,k}; \\ FN_k &= n_{k,+} - n_{k,k}; \\ TN_k &= N - TP_k - FP_k - FN_k. \end{aligned} \tag{9}$$

where N is the number of patterns that do not belong to class k. We evaluated the algorithm's according to three metrics: accuracy, recall (or sensitivity), and precision. These metrics are respectively given by:

$$\text{accuracy}_k = \frac{TP_k + TN_k}{TP_k + FP_k + TN_k + FN_k} \tag{10}$$

$$\text{recall}_k = \frac{TP_k}{TP_k + FN_k} \tag{11}$$

$$\text{precision}_k = \frac{TP_k}{TP_k + FP_k} \tag{12}$$

In the next section we present the results obtained with each classifier and discuss results, based on the concepts presented in this section.

IV. Results and Discussion

This section discusses the results obtained during this work in three different views. We first analyse how the size of the dataset affects the accuracy of the classifiers. We then overview the classification results according to the previously mentioned metrics (i.e., accuracy, recall, and precision) and using the whole dataset. Finally, we go through the detailed results obtained for each ODC attribute. The numbers shown in this section refer to the average of several runs, namely 25 runs using all

algorithms, so that we could identify the best one per each ODC attribute. We report average performance indicators, as well as their variability in the testing data in the format *average value±standard deviation*.

We begin by analyzing how the dataset size affects the overall accuracy of the classification. For this purpose, we carried out a set of tests using SVM, which was the algorithm that revealed the best accuracy values while classifying the complete dataset (see next paragraphs for details on the accuracy values obtained by each algorithm). We varied the size of the dataset using increments of 10% and computed the relative change of the accuracy across the different dataset sizes and using the smallest dataset size (i.e., 10%) as reference. The relative change is calculated as follows:

$$RC\left(x, x_{ref}\right) \frac{x_i - x_{ref}}{x_{ref}} \tag{13}$$

where x_{ref} represents the reference accuracy value for the smallest dataset size (in this case, 10%) and x_i represents the accuracy obtained for a dataset with a certain size i.

Figure 3 presents a graphical view of the relative change values for a single ODC attribute – defect type, which confirms our general expectation that a larger dataset size would lead to better accuracy values. Table VI presents a more detailed vision of the relative change of accuracy for all ODC attributes per varying dataset sizes.

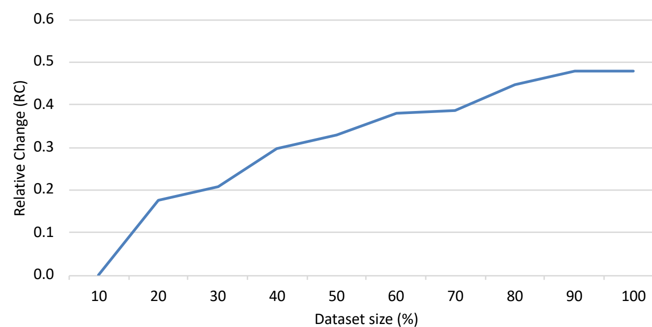


Fig. 3: Relative Change for the Defect Type attribute.

TABLE VI: Relative change values for all ODC attributes

	Dataset Size										# Documents
ODC Attribute	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%	after balancing
Activity	0.00	0.10	0.15	0.19	0.23	0.24	0.25	0.26	0.28	0.29	1932
Code Inspection	0.00	0.23	0.40	0.54	0.65	0.68	0.74	0.78	0.81	0.89	441
Function Test	0.00	-0.02	-0.05	-0.03	-0.08	-0.04	-0.04	0.00	-0.03	-0.04	156
System Test	0.00	-0.03	0.20	0.38	0.36	0.55	0.50	0.68	0.71	0.69	130
Unit Test	0.00	0.12	0.14	0.16	0.13	0.16	0.20	0.13	0.19	0.17	140
Impact	0.00	0.44	0.52	0.47	0.60	0.71	0.80	0.87	0.92	0.92	360
Target	0.00	0.08	0.12	0.17	0.20	0.20	0.22	0.23	0.24	0.25	460
Defect Type	0.00	0.18	0.21	0.30	0.33	0.38	0.39	0.45	0.48	0.48	845
Qualifier	0.00	0.04	0.01	0.05	0.10	0.09	0.10	0.12	0.10	0.10	474

In general, and as we can see in Table VI, all attributes (the exception is Function Test) benefit from increasing the dataset size. This is quite expectable, since machine learning algorithms tend to perform better and its performance is well estimated with increasing ammounts of (good quality) information used during training [41]. In the particular case of the Function Test triggers, and after analyzing the data in further detail, we observed that the increasing amounts of data present in the different dataset sizes do not contribute towards the definition of a pattern that would allow distinguishing among the different triggers. It may be the case that a different dataset is required, or that the current sizes used are simply not sufficient for this particular case.

Additionally, we can observe that, for most of the attributes, the relative change of accuracy does not always progressively change with the increase of the dataset size. More specifically, Unit Test, System Test, Impact, Defect Type and Qualifier do not show the highest relative change value for the dataset size of 100%. Still, the differences between the highest values and those at 100% dataset size are relatively subtle.

Table VII presents the ***overall results*** for all classifiers and per each ODC attribute, with the best values per attribute marked in orange (using the average of 25 runs). Figure 4 visually shows the overall results for the best classifier and per each ODC attribute.

As we can see in Table VII the SVM classifier seems to outperforms all the others. In fact, this was expected since the data is sparse (a considerable number of features values are zero) and this type of classifier is known to handle these cases quite well considering it has a linear kernel, as discussed in [29]. Still, we wanted to know if the observed differences among the

TABLE VII: Overall classifier results per ODC Attribute

Attribute	Algorithm	Accuracy (%)	Recall (%)	Precision (%)
Activity	k-NN	41.3 ± 3.5	41.3 ± 3.5	41.4 ± 3.8
	NB	28.3 ± 2.7	28.3 ± 2.7	34.5 ± 12.6
	NC	40.5 ± 3.0	40.5 ± 3.0	45.8 ± 5.2
	RF	38.3 ± 3.0	38.3 ± 3.0	44.2 ± 7.4
	SVM	45.9 ± 3.3	45.9 ± 3.3	46.2 ± 3.6
Code Inspection	k-NN	38.6 ± 7.2	38.6 ± 7.2	39.0 ± 8.9
	NB	22.9 ± 4.4	22.9 ± 4.4	18.6 ± 10.3
	NC	37.4 ± 7.6	37.4 ± 7.6	40.4 ± 9.4
	RF	34.0 ± 6.8	34.0 ± 6.8	33.8 ± 10.0
	SVM	39.5 ± 7.0	39.5 ± 7.0	43.7 ± 7.9
Function Test	k-NN	35.7 ± 12.0	35.7 ± 12.0	35.0 ± 14.9
	NB	35.2 ± 8.9	35.2 ± 8.9	27.9 ± 17.3
	NC	35.7 ± 11.8	35.7 ± 11.8	35.7 ± 14.1
	RF	33.8 ± 10.9	33.8 ± 10.9	31.4 ± 16.9
	SVM	36.2 ± 11.4	36.2 ± 11.4	36.0 ± 13.7
System Test	k-NN	35.3 ± 12.3	35.3 ± 12.3	34.0 ± 15.2
	NB	26.2 ± 8.5	26.2 ± 8.5	17.1 ± 13.2
	NC	38.2 ± 12.2	38.2 ± 12.2	35.8 ± 15.9
	RF	29.0 ± 12.5	29.0 ± 12.5	25.6 ± 16.2
	SVM	39.0 ± 13.4	39.0 ± 13.4	37.5 ± 17.1
Unit Test	k-NN	57.0 ± 12.6	57.0 ± 12.6	57.6 ± 14.4
	NB	52.7 ± 9.3	52.7 ± 9.3	50.2 ± 22.2
	NC	57.0 ± 13.1	57.0 ± 13.1	57.7 ± 14.4
	RF	55.0 ± 12.5	55.0 ± 12.5	55.6 ± 15.2
	SVM	57.5 ± 12.3	57.5 ± 12.3	58.5 ± 14.7
Impact	k-NN	28.7 ± 7.1	28.7 ± 7.1	29.1 ± 8.9
	NB	13.8 ± 2.5	13.8 ± 2.5	6.6 ± 7.3
	NC	31.5 ± 7.3	31.5 ± 7.3	31.0 ± 9.1
	RF	20.3 ± 5.6	20.3 ± 5.6	21.4 ± 10.9
	SVM	31.6 ± 7.1	31.6 ± 7.1	34.2 ± 9.1
Target	k-NN	75.3 ± 6.3	75.3 ± 6.3	78.2 ± 6.2
	NB	51.9 ± 2.4	51.9 ± 2.4	55.1 ± 24.1
	NC	65.2 ± 8.1	65.2 ± 8.1	67.1 ± 8.5
	RF	74.0 ± 6.4	74.0 ± 6.4	79.7 ± 5.4
	SVM	85.6 ± 5.1	85.6 ± 5.1	86.0 ± 5.1
Defect Type	k-NN	28.3 ± 4.8	28.3 ± 4.8	27.8 ± 6.0
	NB	20.8 ± 2.0	20.8 ± 2.0	11.2 ± 9.8
	NC	28.8 ± 4.8	28.8 ± 4.8	30.3 ± 7.1
	RF	30.2 ± 4.4	30.2 ± 4.4	29.6 ± 6.1
	SVM	34.7 ± 5.1	34.7 ± 5.1	34.8 ± 5.5
Qualifier	k-NN	37.3 ± 7.4	37.3 ± 7.4	37.9 ± 7.8
	NB	34.1 ± 3.4	34.1 ± 3.4	22.6 ± 14.0
	NC	36.6 ± 7.2	36.6 ± 7.2	37.5 ± 8.5
	RF	37.7 ± 7.2	37.7 ± 7.2	39.2 ± 9.4
	SVM	39.7 ± 7.3	39.7 ± 7.3	40.0 ± 7.5

classifiers performance were statistically significant. Thus, we performed statistical tests which ended up by comparing SVM's performance with all the remaining classifiers. For each run we balance data, thus testing data is different between runs, which led us to consider classifier's performance as independent realizations. This allows us to use independent statistical tests. We decided to use t-test or the Mann-Whitney test, depending on the data following a normal distribution or not (respectively). To evaluate normality, we used three tests: Kolmogorov-Sminorv, Shapiro-Wilk and Levene. The t-test is only applied if all three tests determine that data is normal.

Table VIII shows the cases where SVM outperforms the remaining in a statistically significant manner (with an α value of 0.05, i.e., 95% confidence). The cases marked with X refer to those where there are no statistical differences between SVM and a certain algorithm under analysis.

TABLE VIII: Statistical difference of SVM against the remaining algorithms

Algorithm	Activity	Trigger Code Inspection	Trigger Function Test	Trigger System Test	Trigger Unit Test	Impact	Target	Defect Type	Qualifier
k-Nearest Neighbours			X		X				
Naïve Bayes			X						
Nearest Centroid			X	X	X	X			
Random Forest									

As we can see in Table VIII, there are statistically significant differences between the performance of SVM and the remaining classifiers for most of ODC attributes. Figure 4 shows the best results obtained with SVM for each ODC attribute in a graphical manner.

As we can see in Figure 4, we were able to reach an accuracy of 45.9%±3.3% for the Activity attribute, which essentially means that **the 4 classes for the Activity attribute seem to be a little bit hard to separate using our data**. For the Trigger attribute, which we have previously divided in parts according to the respective Activity (according to the Trigger-Activity mapping in [9]), we obtained accuracy values of 39.5%±7.0% for Code Inspection, 36.2%±11.4% for Function Test, 39.0%±13.4% for System Test and 57.5%±12.3% for Unit Test. Since the Trigger attribute is directly related with the Activity ([9]), these results were expected since the Activity results were not high. In Function Test, System Test and Unit Test, the

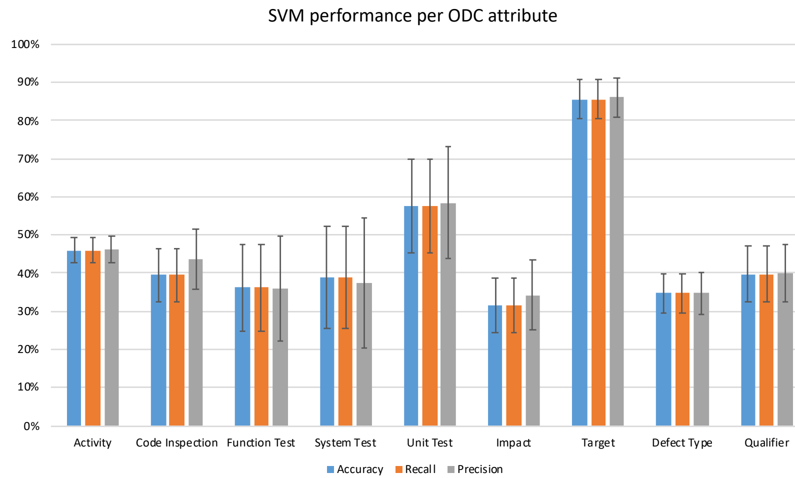


Fig. 4: Best Classifier (SVM) Results per ODC Attribute

standard deviation values are relatively high, suggesting that there is still high variability in the data in what concerns these three attributes.

The accuracy obtained for the Impact attribute was relatively low (31.6%±7.1%), which represents the worst classification performance since it has about half of its classes with no data and still has low accuracy. However, considering that we are still using a relatively large number of classes (i.e., 8 classes) and not a very large quantity of data, results certainly have space for improvement.

Regarding the Target attribute, we were able to obtain 85.6%±5.1% of accuracy (the highest of all attributes). Still, it is important to mention that we just used 2 out of the 6 classes (i.e., we used Code and also Build/Package), as we did not have enough data for the remaining 4 classes. **The high accuracy for Target was expected** since, in the case of this attribute, it is really easy to separate the respective classes just by looking for some keywords, as bugs which are not associated with Code (e.g., bugs associated with build scripts) usually have that information explicitly mentioned in the bug reports.

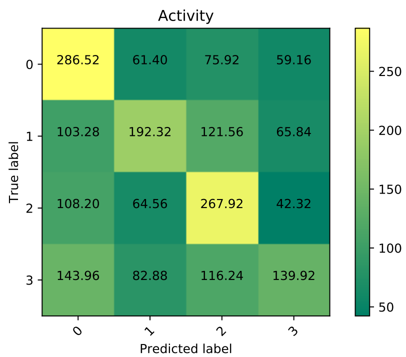
Finally, for the Defect Type and Qualifier attributes we obtained 34.7%±5.1% and 39.7%±7.3% of accuracy, respectively. These **low values obtained for Defect Type and Qualifier are excepted, as these attributes are fully related to what is changed in the code**, thus being the most difficult cases to classify solely based on the text of the bug reports. Thus, to improve the results, it is important to also analyze the code changes (i.e., using other means), which is something that falls outside of the scope of this paper and is left for future work.

Overall, we were able to easily automate the classification regarding the ODC Target attribute. The remaining cases revealed to be more difficult to handle. For these, and in general, the algorithm associated with the best performance showed to be correct between one third and half of the times. Although the values obtained are not high, we find them to be acceptable, mostly due to the relatively low number of reports available (and especially as a consequence of the data balancing performed during this study). As it is presented in Table VI, the classifiers tend to perform better if an adequate quantity of data is used for training, thus gathering more good quality data is a crucial step for a more effective classification procedure.

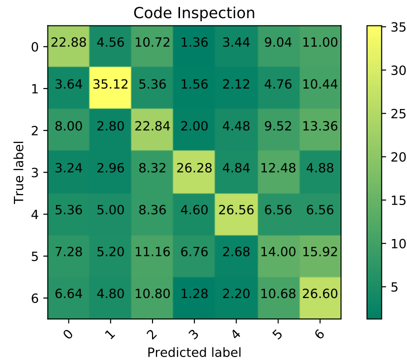
We now ***discuss the detailed results per ODC attribute***, to understand in which classes the algorithms tend to perform better and in which ones the difficulties are higher. For this purpose, we analyze the 9 confusion matrices (one per ODC attribute, including the 4 groups of triggers), as previously explained in Section III-E. It should be mentioned that the values shown in the confusion matrices are not integers since they represent an average of 25 runs.

Figure 5a holds the detailed results for the **Activity** attribute. As we can see, for this classifier **it is relatively easy to identify the Code Inspection and System Test classes**. It was expectable that code inspection could be easily distinguishable from the remaining. Conceptually, System Test (which refers to testing whole systems) should also be relatively easy to identify. On the other hand, we were expecting to see the classifier struggling with Unit Test and Function Test. Unit Test refers to white-box testing of small units of code, while Function Test refers to testing functions and this separation is not always very clear (e.g., in some cases, a unit may be a function, the bug report may not be clear about this). In fact, the independent classification carried out by *researcher2* (see Section III) resulted only in *moderate agreement*, which means that identifying the right activity used in the disclosure of a bug is a somewhat difficult task.

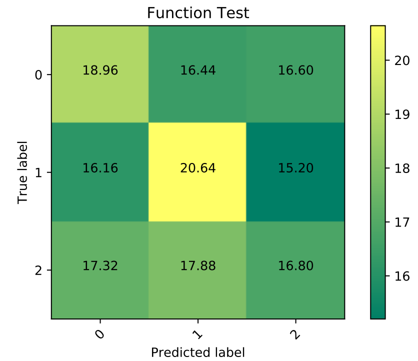
Figure 5b presents the **Trigger - Code Inspection** classifier results. Overall, **the classifier experienced more difficulties when identifying Logic/Flow, which is also the one with the most generic definition** [9] and, possibly more prone to being confused with some other. We can observe that quite often our classifier confuses this class with Design Conformance and even more so with Side Effects.



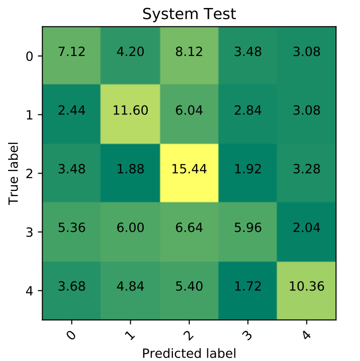
(a) Labels: 0: Code Inspection, 1: Function Test, 2: System Test, 3: Unit Test



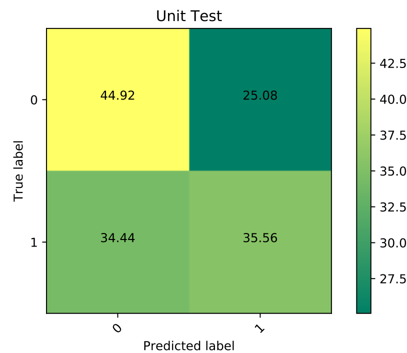
(b) Labels: 0: Backward Compatibility, 1: Concurrency, 2: Design Conformance, 3: Language Dependency, 4: Lateral Compatibility, 5: Logic/Flow, 6: Side Effects



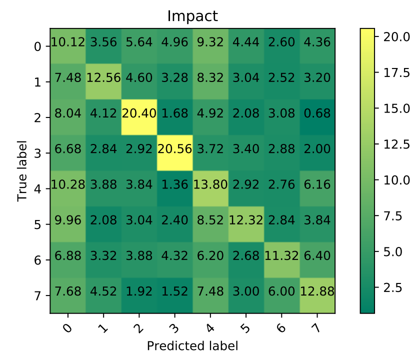
(c) Labels: 0: Coverage, 1: Sequencing, 2: Variation



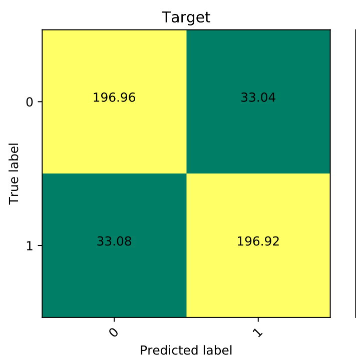
(d) Labels: 0: Blocked Test, 1: Recovery/Exception, 2: Software Configuration, 3: Startup/Restart, 4: Workload/Stress



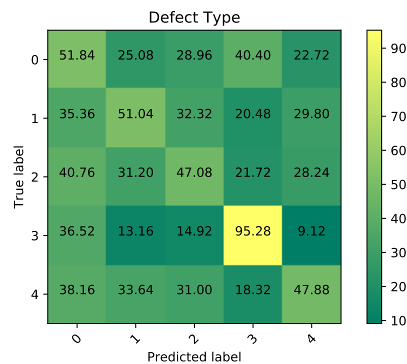
(e) Labels: 0: Complex Path, 1: Simple Path



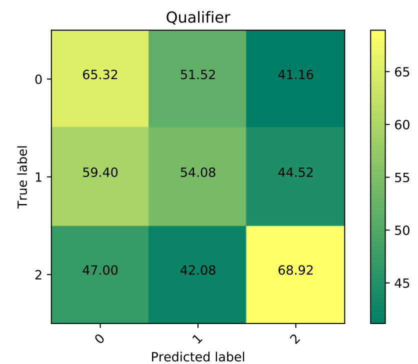
(f) Labels: 0: Capability, 1: Integrity/Security, 2: Performance, 3: Reliability, 4: Requirements, 5: Standards, 6: Serviceability 7: Usability



(g) Labels: 0: Build/Package, 1: Code



(h) Labels: 0: Algorithm/Method, 1: Assignment/Initialization, 2: Checking, 3: Function/Class/Object, 4: Interface O-O Messages



(i) Labels: 0: Extraneous, 1: Incorrect, 2: Missing

Fig. 5: Detailed classification results per ODC attribute: a) Activity; b) Trigger - Code Inspection; c) Trigger - Function Test; d) Trigger - System Test; e) Trigger - Unit Test; f) Impact; g) Target; h) Defect Type; i) Qualifier

On the opposite side, the easiest class to identify is Concurrency. Although there are no overlaps in the definition of any of the classes, is it true that, conceptually, concurrency problems are quite easy cases to identify due to their own specificity. This leaves little error margin for human classifiers which in the end reflects on the performance of machine learning classifiers.

Regarding the **Trigger - Function Test** results depicted in Figure 5c, we find that **all three classes are confused quite often**, although the Sequencing class shows the best true positive results. Function Test-labelled bugs are those which could have been caught through functionality testing (as suggested by the name), and although to the human reader there may be clear differences between the three presented classes (i.e., coverage, sequencing, variation), it is possible that the main keywords referring to each class end up by being quite similar. This was, in fact, reported by *researcher1* that pointed out a general difficulty in distinguishing among the three classes.

The results for the **Trigger - System Test**, presented in Figure 5d, were not quite what we expected since we find the definitions of the System Test triggers to be quite separate, which would make them relatively easy to tell apart, and we can see that **several pairs of Triggers were confused by the classifier**. A possible explanation for the confusion cases (.e.g, mislabeling Blocked Test with Startup/Restart) can be the relatively low number of samples (i.e., 130 samples) and, indeed, we have previously observed a general tendency for better results with larger datasets (see Table VI). Software Configuration showed to be the easiest to identify class, mostly due to the specificities of the keywords related to this class.

In the case of **Trigger - Unit Test the classifier shows difficulties in distinguishing both classes involved** (see Figure 5e). The definitions of these two classes (complex path and simple path) both refer to white-box testing and, although they are conceptually different, they do not largely differ from each other [9]). It is easy to imagine a bug report not being sufficiently clear about one of these particular cases. Despite the relatively similar output, the results show that **it is slightly easier to identify Complex Path Unit Tests than Simple Path**.

Regarding the **Impact** attribute, shown in Figure 5f, we can see that the **Performance and Reliability classes are the easiest to identify** and that, **in general, the amount of bugs that are confused are low, when compared to the amount of correctly labelled bugs**. The Capability class is associated with the worst Impact results, which may be explained by the fact that this is a generic class (i.e., it represents the ability of the software to perform its intended functions and satisfy known requirements) that is select when the user is not affected by any of the other Impact types [9]. We have seen that the human classifier used Capability far more than all other Impact classes (i.e., more than 50% of the bugs in the dataset were marked with Capability), and, when this happens, it is common for the machine classifier to be confused (a possible reason is that the keywords associated with more generic classes overlap with some of the keywords in more concrete classes). Curiously, it seems that it is more frequent for other classes to be confused with Capability (left-most column), than the opposite (top-most row). Perhaps the exception is the confusion between Capability and Requirements, which is relatively high in both ways.

As previously mentioned, **the best results were obtained with the Target attribute**, due to the fact that the respective classes Code and Build/Package are easily separable. Figure 5g shows the results in detail.

As for the **Defect Type** classifier results, which are shown in Figure 5h, **we were actually not expecting good results** because, by definition, this attribute is about the nature of the change that was performed to fix the defect, thus mostly (if not completely) referring to the actual changes in the code. However, we can see that the **results for Function/Class/Object are quite good**, with the classifier seldom confusing this class with the remaining ones. This Defect Type class refers to large changes in the code, that would require formal design changes [9]. We can also see that, the higher number of errors tends to be related with the erroneous classification of a certain type of defect with Algorithm/method (leftmost column). This can be explained by the fact that this is, by definition, the most generic class of the Defect Type attribute.

Finally, Figure 5i shows the detailed results for the **Qualifier** attribute. **The classes associated with the best results are Extraneous and Missing**. It was expected that the three classes had similar results since their definitions do not overlap, with each Qualifier referring to a particular state in which the code was (i.e., present but wrong - Incorrect; present but unnecessary - Extraneous). The Incorrect class was the one which the algorithm confused the most, which we know to be a consequence of (as opposed to the other Qualifiers) this particular value having been used for most bugs (nearly two thirds of the code bugs). In the end, this results in the classifier not easily finding a distinguishable pattern for this particular class.

Overall, regardless of the accuracy values reported, we noted that there are a number of resemblances between the difficulties encountered by the classifier and what the main researcher involved in the creation of the dataset informally reported. The human difficulties would be mostly related with classes that, in practice, become not clearly separable (e.g., simple path and complex path), although they are different by definition. Also, the low quality of some of the reports creates difficulties to the human creation of the dataset, which ends up reflecting in the performance of the classifiers (which have to learn from an erroneous, many times inconsistent, classification). Thus, and although the focus of this work is not on the process taken to manually classify the set of bug reports, we would like to emphasize that the path to achieve higher accuracy values using machine learning classifiers starts with a high quality dataset, which implies a set of actions that, in a pragmatic way, allow us to reach such highly reliable dataset. The difficulties observed simply reflect the reality of handling such complex problem and indicate that there is still space for improvement, namely in the definition of new techniques (e.g., using a composition of different algorithms, mixing machine-learning with other techniques such as rule-based approaches) for automatic classification of bug reports using ODC.

V. Threats to Validity

In this section, we briefly present threats to the validity of this work and discuss mitigation strategies. An obvious first observation refers to the relatively *small dataset size obtained after balancing*. In fact, the difficulties in obtaining large datasets (and especially considering that some classes will be rare) stem from the lengthy and arduous manual labeling of bugs according to the several ODC attributes. We are currently working on extending the dataset, so that we are able to further improve the classification results. The reduced number of reports lead to the *low dimensionality ratio* problem. We tried to control this issue by performing dimensionality reduction using Principal Component Analysis, while preserving as most as possible the retained variance.

An aspect that will be tackled in future work is related with the fact that some ODC attributes refer to code changes that are many times not properly described in the bug reports. As such, the fact that we are *only using the unstructured textual bug reports* is something that may influence the outcome of the classification (i.e., a code change is objective data, whereas a description of a code change may not be). Thus, in future work, we intend to use techniques that directly analyze the code (and especially, the code changes) as a way to provide additional information to the classifiers. Related with the previous threat, we must also mention that, the outcome of the classification process *depends on the quality of the bug reports* and especially on the consistency of the text across reports. But the main aspect to retain here is that we are using what can be found in the field, and correcting or individually changing bug reports would not represent a realistic context.

Our *dataset was originally manually annotated (labeled) by one researcher*. Obviously, this activity may have introduced some errors in the process, especially inconsistent labelling of the same type of bugs. To reduce the likelihood of this threat, *researcher1*, the main researcher responsible for the classification, was first trained with 300 bug reports (which actually touched most of the ODC classes used later). In addition, throughout the classification process, we verified 40% of the classified bugs. 20% of the bugs were incrementally verified by *researcher1* and the remaining 20% by two independent researchers (10% each). The goal was to achieve a more consistent classification and also to gather some insight regarding the reliability of the dataset. Due to the amount of effort involved, in practice it becomes inviable to verify the complete dataset. Obviously, this does not remove the mislabeling problem (as two persons may make the same mistake), but it allows to mitigate it.

VI. Conclusion

This study evaluated the applicability of a set of popular machine learning algorithms (k-Nearest Neighbors, Support Vector Machines, Naïve Bayes, Nearest Centroid, and Random Forest) for performing automatic classification of software defects based on ODC and using unstructured text bug reports as input. The experimental results show the difficulties in automatically classifying certain ODC attributes solely using unstructured textual reports, but also suggest that the overall classification accuracy may be improved if larger datasets are used. Based on our work, we are providing developers and researchers with a public and free RESTful service (available at *http://odc.dei.uc.pt*) for performing automatic classification of bugs according to ODC. The service receives the bug report and the machine learning algorithm to use as input and classifies the bug according the ODC scale. As future work, we intend to extend the dataset used, integrate code changes in the whole process, and also consider a combined approach using different machine learning algorithms in the definition of a new automatic classification process, particularly tailored to handle the ODC specificities.

References

[1] A. Bouguettaya, M. Singh, M. Huhns, Q. Z. Sheng, H. Dong, Q. Yu, A. G. Neiat, S. Mistry, B. Benatallah, B. Medjahed, M. Ouzzani, F. Casati, X. Liu, H. Wang, D. Georgakopoulos, L. Chen, S. Nepal, Z. Malik, A. Erradi, Y. Wang, B. Blake, S. Dustdar, F. Leymann, and M. Papazoglou, "A Service Computing Manifesto: The Next 10 Years," *Commun. ACM*, vol. 60, no. 4, pp. 64–72, Mar. 2017. [Online]. Available: http://doi.acm.org/10.1145/2983528

[2] M. Grottke, D. S. Kim, R. Mansharamani, M. Nambiar, R. Natella, and K. S. Trivedi, "Recovery From Software Failures Caused by Mandelbugs," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 70–87, Mar. 2016.

[3] M. Grottke and K. S. Trivedi, "Fighting bugs: remove, retry, replicate, and rejuvenate," *Computer*, vol. 40, no. 2, pp. 107–109, Feb. 2007.

[4] R. Chillarege, "Orthogonal defect classification," in *Handbook of Software Reliability Engineering*, M. R. Lyu, Ed. IEEE CS Press, 1996, ch. 9, pp. 359–399.

[5] L. Huang, V. Ng, I. Persing, M. Chen, Z. Li, R. Geng, and J. Tian, "Autoodc: Automated generation of orthogonal defect classifications," *Automated Software Engineering*, vol. 22, no. 1, pp. 3–46, 2015.

[6] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012, pp. 205–214.

[7] F. Thung, X.-B. D. Le, and D. Lo, "Active semi-supervised defect categorization," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. IEEE Press, 2015, pp. 60–70.

[8] J. Hernández-González, D. Rodriguez, I. Inza, R. Harrison, and J. A. Lozano, "Learning to classify software defects from crowds: a novel approach," *Applied Soft Computing*, vol. 62, pp. 579–591, 2018.

[9] IBM. (2013, Sep.) Orthogonal Defect Classification v 5.2 for Software Design and Code. [Online]. Available: https://researcher.watson.ibm.com/researcher/files/us-pasanth/ODC-5-2.pdf

[10] L. Zhi-bo, H. Xue-mei, Y. Lei, D. Zhu-ping, and X. Bing, "Analysis of software process effectiveness based on orthogonal defect classification," *Procedia Environmental Sciences*, vol. 10, pp. 765–770, 2011.

[11] S. Kumaresh and R. Baskaran, "Defect analysis and prevention for software process quality improvement," *International Journal of Computer Applications*, vol. 8, 2010.

[12] J. Christmansson and R. Chillarege, "Generation of an error set that emulates software faults based on field data," in *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*. IEEE, 1996, pp. 304–313.

[13] J. Durães and H. Madeira, "Definition of software fault emulation operators: a field data study," in *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, San Francisco, CA, USA, 2003, pp. 105–114.

[14] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[15] S. Bellucci and B. Portaluri, "Automatic calculation of orthogonal defect classification (ODC) fields," U.S. Patent 8 214 798, Jul., 2012. [Online]. Available: http://www.freepatentsonline.com/8214798.html

[16] C. Liu, Y. Zhao, Y. Yang, H. Lu, Y. Zhou, and B. Xu, "An ast-based approach to classifying defects," in *Software Quality, Reliability and Security-Companion (QRS-C), 2015 IEEE International Conference on*. IEEE, 2015, pp. 14–21.

[17] D. Cubranic and G. Murphy, "Automatic bug triage using text categorization," in *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. Citeseer, 2004.

[18] J. Xuan, H. Jiang, Z. Ren, J. Yan, and Z. Luo, "Automatic bug triage using semi-supervised text classification," *CoRR*, vol. abs/1704.04769, 2017.

[19] Y. Tian, D. Lo, and C. Sun, "Drone: Predicting priority of reported bugs by multi-factor analysis," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 2013, pp. 200–209.

[20] B. V. Dedık, "Automatic ticket triage using supervised text classification," Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2015.

[21] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. IEEE, 2008, pp. 346–355.

[22] J. A. Durães and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, 2006.

[23] T. Sotiropoulos, H. Waeselynck, J. Guiochet, and F. Ingrand, "Can Robot Navigation Bugs Be Found in Simulation? An Exploratory Study," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Jul. 2017, pp. 150–159.

[24] A. Rahman, S. Elder, F. H. Shezan, V. Frost, J. Stallings, and L. Williams, "Categorizing Defects in Infrastructure as Code," *arXiv:1809.07937 [cs]*, Sep. 2018, arXiv: 1809.07937. [Online]. Available: http://arxiv.org/abs/1809.07937

[25] J. Cohen, "A Coefficient of Agreement for Nominal Scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, Apr. 1960. [Online]. Available: https://doi.org/10.1177/001316446002000104

[26] J. R. Landis and G. G. Koch, "The Measurement of Observer Agreement for Categorical Data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977. [Online]. Available: https://www.jstor.org/stable/2529310

[27] C. D. Manning, P. Raghavan, and H. Schütze, *Scoring, term weighting, and the vector space model*. Cambridge University Press, 2008, p. 100–123.

[28] A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets*. New York, NY, USA: Cambridge University Press, 2011.

[29] T. Joachims, "Text categorization with suport vector machines: Learning with many relevant features," in *Proceedings of the 10th European Conference on Machine Learning*, ser. ECML '98. London, UK, UK: Springer-Verlag, 1998, pp. 137–142. [Online]. Available: http://dl.acm.org/citation.cfm?id=645326.649721

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[31] T. Archanah. and D. Sachin, "Dimensionality reduction and classification through pca and lda," *International Journal of Computer Applications*, vol. 122, no. 17, p. 4–8, 2015.

[32] J. de Sa, *Pattern Recognition: Concepts, Methods, and Applications*. Springer, 2001. [Online]. Available: https://books.google.pt/books?id=O5vwppJQQwIC

[33] A. Rea and W. Rea, "How Many Components should be Retained from a Multivariate Time Series PCA?" oct 2016. [Online]. Available: http://arxiv.org/abs/1610.03588

[34] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[35] A. McCallum, K. Nigam *et al.*, "A comparison of event models for naive bayes text classification," in *AAAI-98 workshop on learning for text categorization*, vol. 752, no. 1. Citeseer, 1998, pp. 41–48.

[36] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[37] K.-B. Duan and S. S. Keerthi, "Which is the best multiclass svm method? an empirical study," in *International workshop on multiple classifier systems*. Springer, 2005, pp. 278–285.

[38] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[39] ——, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[40] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI'95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 1137–1143. [Online]. Available: http://dl.acm.org/citation.cfm?id=1643031.1643047

[41] C. Beleites, U. Neugebauer, T. Bocklitz, C. Krafft, and J. Popp, "Sample size planning for classification models," *Analytica Chimica Acta*, vol. 760, pp. 25 – 33, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0003267012016479