How Do Implementation Bugs Affect the Results of Machine Learning Algorithms?

Maurizio Leotta, Dario Olianas, Filippo Ricca, Nicoletta Noceti

Abstract:

Applications based on Machine learning (ML) are growing in popularity in a multitude of different contexts such as medicine, bioinformatics, and finance. However, there is a lack of established approaches and strategies able to assure the reliability of this category of software. This has a big impact since nowadays our society relies on (potentially) unreliable applications that could cause, in extreme cases, catastrophic events (e.g., loss of life due to a wrong diagnosis of an ML-based cancer classifier).

In this paper, as a preliminary step towards providing a solution to this big problem, we used automatic mutations to mimic realistic bugs in the code of two machine learning algorithms, Multilayer Perceptron and Logistic Regression, with the goal of studying the impact of implementation bugs on their behaviours.

Unexpectedly, our experiments show that about 2/3 of the injected bugs are silent since they does not influence the results of the algorithms, while the bugs emerge as runtime errors, exceptions, or modified accuracy of the predictions only in the remaining cases. Moreover, we also discovered that about 1% of the bugs are extremely dangerous since they drastically affect the quality of the prediction only in rare cases and with specific datasets increasing the possibility of going unnoticed.

Digital Object Identifier (DOI):

https://doi.org/10.1145/3297280.3297411

https://doi.org/10.1145/3297280.3297411

How Do Implementation Bugs Affect the Results of Machine Learning Algorithms?

Maurizio Leotta
Dip. di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Italy
maurizio.leotta@unige.it

Dario Olianas
Dip. di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Italy
S3717893@studenti.unige.it

Filippo Ricca
Dip. di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Italy
filippo.ricca@unige.it

Nicoletta Noceti
Dip. di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Italy
nicoletta.noceti@unige.it

ABSTRACT

Applications based on Machine learning (ML) are growing in popularity in a multitude of different contexts such as medicine, bioinformatics, and finance. However, there is a lack of established approaches and strategies able to assure the reliability of this category of software. This has a big impact since nowadays our society relies on (potentially) unreliable applications that could cause, in extreme cases, catastrophic events (e.g., loss of life due to a wrong diagnosis of an ML-based cancer classifier).

In this paper, as a preliminary step towards providing a solution to this big problem, we used automatic mutations to mimic realistic bugs in the code of two machine learning algorithms, Multilayer Perceptron and Logistic Regression, with the goal of studying the impact of implementation bugs on their behaviours.

Unexpectedly, our experiments show that about 2/3 of the injected bugs are *silent* since they does not influence the results of the algorithms, while the bugs emerge as runtime errors, exceptions, or modified accuracy of the predictions only in the remaining cases. Moreover, we also discovered that about 1% of the bugs are extremely *dangerous* since they drastically affect the quality of the prediction only in rare cases and with specific datasets increasing the possibility of going unnoticed.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; • **Theory of computation → Machine learning theory**;

KEYWORDS

Testing, Machine Learning, Bug, Accuracy, Oracle Problem, Software Quality Assurance

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297411

ACM Reference Format:
Maurizio Leotta, Dario Olianas, Filippo Ricca, and Nicoletta Noceti. 2019. How Do Implementation Bugs Affect the Results of Machine Learning Algorithms?. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3297280.3297411

1 INTRODUCTION

Machine Learning (ML) refers to a family of data-driven techniques able to *learn from* and *make predictions on* data instead of being explicitly programmed [1]. The considered problem may take the shape of classification (assign a sample to a category) or regression (estimate a value) tasks.

A typical ML task starts with the collection of a *training set* providing a sampling of the input space. The main goal of ML is to estimate the underlying model(s) ruling such data, and thus providing a good representation of the training set. At the same time, the *generalization* to new data must be ensured.

In recent years, ML emerged as one of the most relevant technology enabler within a multitude of different sectors such as medicine, bioinformatics and finance. ML allows artificial agents to find hidden insights from big volumes of data and for this reason, it plays a fundamental role in decision making tasks and helps to find good solutions to complex problems. However, the potential to apply ML to safety critical tasks strongly depends on the reliability of such software solutions. Thus, ***ensuring the correctness of machine learning based applications is of paramount importance***.

The relevance of the topic is also testified by influential scientists (e.g., several *Nature* articles pin point the problem of quality control in scientific computational software [2, 8, 14]), major companies (e.g., a recent Capgemini article [11] reports *"Software testing will be one of the most critical factors that determine the success of a machine learning system"*), and from the fact that the premier workshop in automated testing, AST 2018[1], has dedicated a special theme on the topic "Artificial Intelligence (AI) for Test Automation and Test Automation for AI/ML (ML) software". However, even if the problem is out in the open, there is still a lack of consolidated and sound approaches able to verify the reliability of ML software. Indeed, ML applications represent a peculiar class of software applications on which common software quality assurance techniques (e.g., traditional software testing) cannot be easily applied [19].

[1] http://ast2018.isti.cnr.it/topics.html

To the best of our knowledge, our work is the first that proposes a study having the goal of analysing the influence of implementation bugs on the behaviour of ML algorithms (i.e., on the results they provide) following a purely data-driven approach. We believe that the discussion and the results proposed in this paper constitute a first step towards the definition of novel strategies and techniques for assuring the quality of ML-based solutions. ***Understanding the behaviour of such systems in presence of bugs*** is a first, fundamental prerequisite for conceiving techniques able to detect bugs.

The paper is organised as follows: Section 2 describes the factors which hinder the quality assurance process of ML solutions; Section 3 reports the related works. The empirical study is described in Section 4, followed by conclusions and future work (Section 5).

2 MACHINE LEARNING QUALITY ASSURANCE: THE CURRENT SITUATION

Nowadays, the main research effort of ML scientists is in improving the effectiveness of models and algorithms to better automate the learning from real world data. On the contrary, very little effort and work has been done for assuring the correctness of ML-based solutions. It is well-known that formal proofs of algorithm's properties do not guarantee that the concrete implementation of such algorithm is correct, and thus software testing is necessary.

Software testing is a fairly straightforward activity. For every input, there is a defined and known output. The procedure used in software testing can be defined as follows. We define an input, make selections and interact with the target application; then we compare the actual result, provided by such an application, with the expected one (the oracle). If they match, we select another input, if they do not, we possibly have *identified a bug*. However, such an approach is not applicable to ML-based applications and algorithms since, in this specific context, the output cannot be simply computed for real samples outside the training-test sets, and, in some cases, it cannot be expected a priori: it will change over time as the model on which the ML system is built evolves.

The problem of defining the expected outputs is known as "the oracle problem" [19]. In literature, a technique called *metamorphic testing* [3] has been proposed to face it. This technique is based on the definition of metamorphic relations that such algorithm would be expected to satisfy (more details on it are reported in the related work section). Even if, nowadays, metamorphic testing is one of the best approaches to ML testing, it has limitations (see the related work section) making impossible to entirely solve the oracle problem. Thus, it cannot be actually seen as a complete solution to the oracle problem.

For this reason, differently from traditional software testing, the evaluation of the behaviour of ML algorithms is, in most of the cases, purely data-driven and conducted empirically. In fact, it is common practice in ML to evaluate the quality of the algorithm implementation by analysing its performances (e.g., measuring the accuracy of the results) on one or more data sets. A typical ML pipeline is based on the use of *training*, *validation* and *test* sets. In a supervised setting, training and validation sets are employed to learn the relation between input and output and to perform model selection (i.e., selecting the appropriate parameters). The estimated function is applied to the test set, as a final evaluation of its generalisation capability. When a validation set is not available, K-fold Cross Validation represents a viable alternative. Although effective and robust to evaluate the performance of the learned models on the available data, such pipeline provides only a partial view on the correctness of the algorithms, and is not designed to perform software verification. For example, cross-validation has been widely adopted as the main method for evaluating supervised classifier systems for decades [21]. The problem is that cross validation is not designed for performing verification. However, practitioners in ML often rely on it for verifying the correctness of their implementation of ML algorithms.

In case of poor performance of the implemented solution, the presence of bugs in the algorithm's implementation is only one of the possible causes. In fact, the results provided by ML algorithms could be negatively influenced by other, possibly interconnected, factors (this is well-known by ML experts): 1) an important fraction of the data is noisy or mislabelled, 2) the objective function is inappropriate, 3) the training set does not provide and appropriate characterisation of the input space, 4) test and training sets do not respond well to the same model, 5) the representation of data to feed into the model is not so appropriate, 6) the chosen ML algorithm is inappropriate.

Moreover, bugs in the algorithm's implementation can be sneaky to find since: "*codes may be riddled with tiny errors that do not cause the program to break down, but may drastically change the scientific results that it spits out*" [14].

Therefore, understanding if a low performance in terms of accuracy of a ML algorithm is due to an implementation bug or not is a difficult task. As a consequence, ***ensuring the correctness of machine learning applications presents huge challenges.***

3 RELATED WORK

The problem of assuring the quality of ML-based solution is perceived as very relevant in various sectors. For instance, concerning scientific computational software, several articles published on the prestigious *Nature* journal [2, 8, 14] pin point the problem of quality control in this category of software (of which ML software is undoubtedly a relevant fraction). Alden et al. [2] state that to unlock the full potential of the computer-based science, software engineering must be at peak quality throughout and that thus scientists must ensure that the research is relevant, but at the same time, software engineers must make sure that it is correct. Merali [14] highlights the problem that in computational software researchers conceive and implement novel algorithms without following the proper software engineering best practices that help to ensure the quality of the produced software. Hayden [8] reports that researchers increasingly rely on computation to perform tasks at every level of science, but most of them do not receive formal training in coding best practices. As a consequence, coding errors can be present in the algorithms implementations.

Assuring the quality of ML applications is an active research field. Indeed, only a few months ago Masuda et al. [13], performed a survey of the literature on software quality for ML applications considering about a hundred papers from academic conferences and journals including top venues of both the machine learning and

testing sectors like: ICML, NIPS, ICST, ISSTA and TSE. A relevant portion of the analysed papers have been published in the last 5 years.

Applying software testing in presence of the oracle problem is really complex (software having this problem is often called non-testable software). During the years, several techniques have been applied to face it: random testing [7], assertion checking [18], N-version programming [12], pseudo-oracle [6] and metamorphic testing [3]. In the following, for space reasons, we will focus only on the last two techniques. In particular, at date, metamorphic testing seems to be one of the most effective techniques for testing ML solutions.

In the case of pseudo-oracle [6], the same input is provided to multiple, independent, implementations of the same algorithm and the results are compared; if the output is not the same, then a defect/bug is present in at least one of the implementations; otherwise the implementations are likely to be correct. This approach is not always feasible, given that multiple implementations of the same algorithm may be difficult to find. When they are available, it is important to verify that they have not been created by the same developers since, in such case, they could contain the same mistakes [9].

In the literature, a different approach for non-testable software, that does not require multiple implementations, is available: it is named metamorphic testing [3]. The approach is based on the fact that often the algorithms exhibit properties such that if the input were modified in a certain way, it should be possible to predict (or estimate) the new output, given the original output. The steps to apply metamorphic testing are the following: (1) defining a set of properties called "metamorphic relations" (MRs) that relate multiple pairs of inputs and outputs of the target program; (2) defining pairs of source test cases and their corresponding follow-up test cases based on the MRs; (3) executing the test cases on the algorithm under test, and check whether the outputs of the source and follow-up test cases satisfy the corresponding MRs. As an example, let's suppose we want to test a web search engine. A possible way to test it is applying metamorphic testing providing a query Q (e.g., "Machine Learning") to the web search engine. An example of MR could be: to each query Q corresponds a number of results R; if, for example, we create a more strict query Q' by adding a predicate in conjunction to Q (e.g., "Machine Learning Algorithms") the number of results R' will be less or equal to R (the predicate is more selective). Thus, we could create arbitrary pairs of test inputs (i.e., in this case the queries Q and Q') and verify if the algorithm under test behave as expected even if we are not able to define R and R' in advance (in a web search engine the database is continuously updated, thus test cases with fixed values for R and R' would become quickly obsolete).

Although, several works recognized that metamorphic relations can effectively alleviate the oracle problem in testing, they can never completely solve it. Indeed, in a recent article published on ACM Computing Surveys [5], Chen et al. (the creator of metamorphic testing [3]) state: "*MRs are necessary properties of the target algorithm in relation to multiple inputs and their expected outputs, but because there are usually a huge number of these properties, it is almost impossible to obtain a complete set of MRs representing all of them. Even if it were possible to obtain such a complete set of MRs, they might still not be equivalent to a test oracle due to the necessary (but not sufficient) nature of the properties*". Moreover, another limitation of metamorphic testing is in the definition of the metamorphic relations. Even if they seems obvious on simple examples, they can be quite complex to define, from a practitioner point of view, in real cases. In particular, among the various possible MRs, it is important to select those providing "strict" constraints on the results ("weak" constraints can only identify macroscopic errors). As a consequence, testers expertise and experience is of paramount importance for identifying MRs [5].

Metamorphic testing has been applied for several years to test ML algorithms and in bioinformatics software. For instance, Xie et al. [21] applied this technique for testing and validating ML classifiers based on the Weka library[2] (in particular the k-Nearest Neighbors and the Naïve Bayes Classifiers). More in detail, they propose a set of metamorphic relations for classification algorithms and an associated technique that uses these relations to enable scientists to test and validate their implementations. Chen et al. [4] describe an approach based on metamorphic for testing bioinformatics programs. They report that *"the systematic testing of many bioinformatics programs is difficult due to the oracle problem"*. Moreover, they state that *"as biologists increasingly rely on the results produced by these bioinformatics programs, it is crucial to ensure that they are of high quality. We wish this paper can raise the awareness of proper software testing practice in the bioinformatics community"*.

4 EMPIRICAL STUDY

In this paper, we want to investigate how the presence of bugs in the implementation of ML algorithms affect their results (and more in general their behaviour). As preliminary step, we focused on supervised classification problems. Classification is the problem of identifying to which category a new observation belongs. Observations are represented as sets of features (i.e., attribute values). Supervised learning is the task of learning a function that maps an input to an output, given a set of known pairs input/output. Classification can be binary (only two classes, and elements must belong to one or another) or multi-class (there are more than two classes). We focused on this category of problems because (1) classification is a very common task that ML data scientists have to perform, and (2) it allows to quantitatively evaluate the performances of the selected algorithms in terms of accuracy of the predictions.

There are several ways a *bug* can affect the behaviour of a ML algorithm running on a specific dataset: (1) it can leave the behaviour of the algorithm unchanged (for instance, when the line containing the bug is not executed or because the change is insignificant, like changing $var > 0$ in $var >= 0$ if var is always above 100 when the algorithm is executed on the specific dataset); (2) it can make the program to crash at runtime (these are the easier bugs to find), or (3) it can modify the results of the predictions, and thus the obtained accuracy. Bugs can be subtle, because of the oracle problem (in a real scenario, we do not know the expected accuracy) and because the incorrect behaviour may arise only with specific datasets and settings. The impact on accuracy can range from low to remarkable.

[2] https://www.cs.waikato.ac.nz/~ml/weka/

However, even low impacts must not be underestimated: in safety critical applications, e.g., a melanoma detector, a 1% accuracy reduction can bring to the loss of many lives.

The following of this section presents the considered algorithms and datasets, the automated procedure for generating the mutants mimicking realist bugs, the research questions, procedure and metrics, the results, and finally some considerations followed by the threats to validity of the empirical study. The ***goal*** of this paper **is to evaluate whether, and measure how much, bugs influence the behaviour of machine learning algorithms**. We follow the guidelines by Wohlin *et al.* [20] on designing and reporting empirical studies. The results of this study are interpreted according to the *perspective* of both practitioners and researchers interested to increase the quality of machine learning algorithms, but also of ML-based software libraries users since they may want to be aware of the possible problems that could arise if the chosen implementations contain bugs. In the experiment we used (1) two implementations of two common machine learning algorithms and (2) nine datasets.

4.1 Algorithms

We conducted our experiments employing two machine learning classification algorithms. In this study we selected: (1) Multilayer Perceptron as representative of the neural networks class: they are very common solutions in ML and recently gained new attention with the advent of Deep Learning; and (2) Logistic Regression that is among the most used algorithms by data scientists. The two algorithms are based on different ideas (their implementations are different) and both are available in the Scikit-learn library. Scikit-learn[3] [17] is a well-known free software machine learning library for the Python programming language. In the following, we briefly describe the two selected algorithms.

4.1.1 Multilayer Perceptron. The Multi-layer Perceptron[4] (*MLP* from hereafter) is a supervised learning algorithm based on a neural network composed of an input layer, an output layer and one or more hidden layers. Features are given to the input layer (which has one neuron per input feature) and passed through each hidden layer. Each neuron in the hidden layer transforms the m different values from the previous layer with a weighted linear summation $w_1x_1 + w_2x_2 + \ldots + w_mx_m$ followed by a non-linear activation function $g() : R \times R$. Globally a *MLP* learns a function $f() : R_i \times R_o$ where i is the number of input features (and thus, the number of neurons in the input layer) and o is the number of output class (and thus, the number of neurons in the output layer). We used the MLPClassifier class and built a network with hidden layers of size (14, 32, 18, 8), parameter *alpha* = 0.0001 and Rectified Linear Unit $f(x) = max(0, x)$ as activation function. We empirically selected the parameters values by maximizing the average accuracy across all the datasets. Note that since our goal is to evaluate the effect of bugs, the absolute performances of the algorithms on each dataset are not relevant since we compare the original results with the ones of the versions containing the bugs. The implementation of the *MLP* used in this study is provided by Scikit-learn library and can be found here[5]. As a preliminary step, starting from the original implementation, we removed the portions that are not used for classification tasks (indeed it supports also regression tasks). This is useful for avoiding the generation of mutants that will never be executed in our experiment. The considered implementation, after the preprocessing step, is 484 LoC long (comments are excluded) and divided in two object oriented classes: BaseMultilayerPerceptron and MLPClassifier.

4.1.2 Logistic Regression. The Logistic regression[6] (*LogReg* from hereafter), despite its name, is a linear model for classification which supports both binary and multiclass classification. Also for this algorithm, we used the implementation provided by the Scikit-learn library. For this algorithm, we selected the Newton-CG method solver with L2 penalization. The implementation of the considered algorithm can be found here[7]. Also in this case, we removed the portions that are not used for classification tasks. The considered implementation, after the preprocessing step, is 700 LoC long, some in standalone methods and some in the LogisticRegression class.

4.2 Datasets

In order to analyse the behaviour of the two algorithms in different circumstances, we considered nine datasets with heterogeneous characteristics: binary and multi-class datasets, high and low dimensionality, synthetic (i.e., where data are not obtained by direct measurement but artificially generated [16]) and real (i.e., where data are from the real world), overlapping and non-overlapping categories. Synthetic datasets have been considered in order to have a complete control on the quality of the data (completely separable or not; in the latter case amount of overlapping) while the three real datasets has been selected among free datasets having a reasonable size in order to limit the computational effort required to complete the experiments. The datasets Gaussian Single, Gaussian Double (variants: 2, 4, 6, 8), and Body Mass Index are synthetic datasets, while the datasets Iris, Cleveland Heart Disease, and Pulsar contain real data.

In the Gaussian Clouds based dataset the feature are the x,y coordinates of the points while the label can assume two values: purple and green. Fig. 1 shows the plots of the five Gaussian datasets used in our experimentation. More in detail, *Gaussian Single* (*SG* from hereafter) is a gaussian cloud of points (μ=(5,5) and σ=2) divided in two classes by the line $y = x$. The cloud is composed of 1000 points.

The four *Gaussian Double* datasets are created by using two gaussian distributions (one with μ=(0,10) and the other μ=(10,0)) with different levels of overlapping. To this end, we changed the standard deviation of the two distributions (σ=2, 4, 6, 8). Each cloud is composed of 1000 points. The four dataset are called DG_2, DG_4, DG_6, and DG_8.

In the case of *Body Mass Index*[8] dataset (*BMI* from hereafter), each point has two features (mass, height) and a label representing a BMI class (8 classes from 0 to 7 categorizing a person from underweight

[3] http://scikit-learn.org/
[4] http://scikit-learn.org/stable/modules/neural_networks_supervised.html
[5] http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html
[6] http://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
[7] http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
[8] https://en.wikipedia.org/wiki/Body_mass_index

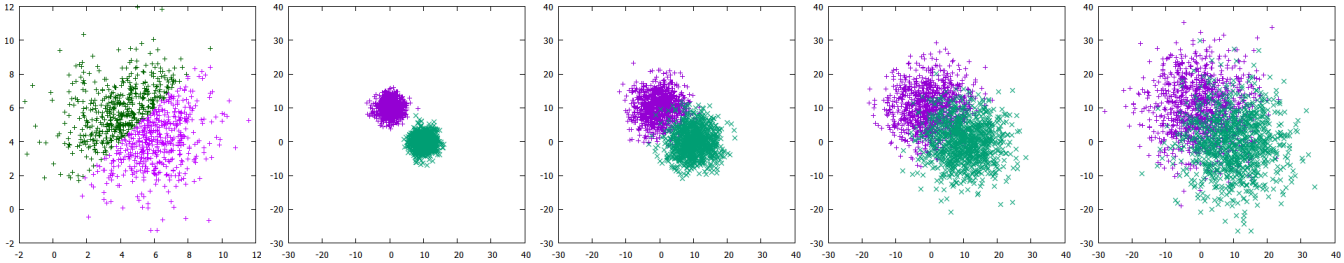

Figure 1: Left to right: Gaussian Single with $\mu = (5, 5)$ and $\sigma = 2$, and Gaussian Double with $\mu_a = (0, 10)$, $\mu_b = (10, 0)$ and $\sigma = 2, 4, 6, 8$

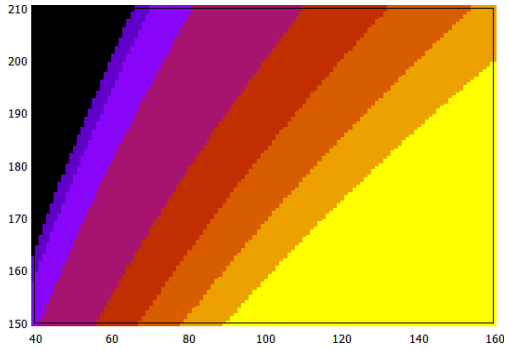

Figure 2: BMI dataset with 8 classes (e.g., yellow is obese)

to obese based on that value, see Fig. 2). We first sampled the possible area of BMI: x ranges from 40 to 160kg (sample size 1kg) and y from 150 to 210cm (sample size 0.5cm). In this way, we obtained $120x120 = 14400$ points. Then, we extracted 200 points per class using a script that uniformly selects the points from the area of each class.

The *Iris*[9] dataset (*Iris* from hereafter) is a biological multi-class dataset that classifies 3 different species of Iris flowers (Setosa, Versicolour, Virginica) using 4 features (sepal length, sepal width, petal length, and petal width). The dataset contains 50 instances for each class. It is one of the datasets used by the Scikit Learn official unit tests.

The *Cleveland Heart Disease*[10] dataset (*Cleveland* from hereafter) is a medical multi-class dataset with 14 features (representing various patient's attribute like age, sex, cholesterol and more specific ones) and 5 classes, that represent the health status of the patient (0 = healthy, while 1,2,3,4 = different levels of disease). It has 297 entries overall and in particular 160, 54, 35, 35, 13 entries respectively for the classes 0, 1, 2, 3, 4.

The *Pulsar HTRU2*[11] dataset (*Pulsar* from hereafter) is a two class astronomical dataset. Pulsars are a rare type of neutron stars that produce radio emission detectable from Earth. Samples are classified into pulsar and non-pulsar classes based on the values of 9 features (8 continuous variables and a single class variable). The dataset contains 17898 total examples: 1639 positive and 16259 negative.

For all the considered datasets, except *Cleveland*, the points in each class have been equally distributed between training and test sets (50-50 subdivision). For the *Cleveland* dataset, which has few elements with an unbalanced class distribution, we used a 5-fold cross-validation. The final accuracy value is the mean of the five combinations.

4.3 Mutants Generation

A mutant is a slight variation of the original code simulating typical errors a developer could make during development and maintenance activities. For this reason each mutated line can be seen as a possible bug. Mutants are traditionally used in the context of testing for evaluating the quality of the produced test suites (mutation testing [15]). Indeed, the mutants can be used to identify the weaknesses in the verification artefacts by determining the parts of a software that are badly or never checked [10]. If a test is able to detect the mutants is likely to have good chances of detecting bugs.

In this work, we generated mutants of the two selected algorithms (i.e., *MLP* and *LogReg*). The mutation phase is usually driven by mutation operators which affect small portions of code, exploiting some typical programming mistakes, like a change in a logical/-mathematical operator (e.g., AND/+ instead of OR/-), a boolean substitution (e.g., from true to false), or a conditional removal (e.g., an IF condition statement is set to true). Our goal is understanding how the mutants change the behaviour of machine learning algorithms. Manually generating the mutants in a realistic scenario is clearly infeasible (and, in the context of an experiment, possibly biased), but there exist automated tools (used in the context of mutation testing) providing operators for generating a large number of mutants starting from the original code.

In this work we used Mut.py[12], a tool for mutation testing that generates mutants of a given Python source code and run a test suite against them. It inserts various kinds of mutations, like operators replacement and deletions, slice index removal and constant replacement. We modified the tool in order to save the full source code of the mutants to files. In this way, we can run them independently and record interesting data for our experimentation, e.g., the coverage of the mutated line, behaviour of the mutant, possible runtime errors, accuracy of the result computed by the mutant (if any). Overall the tool generated 576 mutants for *MLP* and 732 for *LogReg*.

In this work, a *mutant* is a new version of the original algorithm implementation generated by applying a *mutation*. Only one mutation is applied for each mutant. Thus all the mutants differ from the original version only for a single mutated line. Thus, after the

[9] https://archive.ics.uci.edu/ml/datasets/iris
[10] https://archive.ics.uci.edu/ml/datasets/heart+Disease
[11] https://archive.ics.uci.edu/ml/datasets/HTRU2
[12] https://pypi.org/project/MutPy/

mutation, in the resulting mutant it is present a *bug*, i.e., the error introduced by the mutation.

4.4 Research Questions, Procedure and Metrics

Our study aims at answering the following four research questions.
RQ1: *What is the effect of a bug on the behaviour of a machine learning algorithm?*

The goal of the first research question is to analyse the effect of a bug on the behaviour of a ML algorithm implementation. The possible cases are: same behaviour or different behaviour (i.e., accuracy different from the original or exception raised/error). The metrics used to answer RQ1 is the percentage of mutants having a different behaviour, w.r.t. the original implementation, out of the total number of mutants.
RQ2: *What is the effect of a bug on the accuracy of machine learning algorithms?*

The goal of the second research question is to measure the variation of the accuracy in presence of bugs in the implementation of ML algorithms. The metrics used to answer RQ2 is the accuracy.

The definition of *accuracy* we used is the number of correct predictions out of the total numbers of predictions on the test set (e.g., a 0.91 accuracy means that the 91% of the predictions are correct). It is a good metrics for measuring the performances of ML algorithms since it gives a clear insight of their performance.

To answer RQ1 and RQ2 we: (1) considered the two classification algorithms described in Section 4.1 (i.e., *MLP* and *LogReg*), (2) considered the nine datasets described in Section 4.2, (3) computed the accuracy of the prediction for the original version of each algorithm on each dataset, (4) mutated the algorithms as described in Section 4.3.

Then, for answering RQ1, we executed each mutant on each dataset noting down the observed behaviour. For each mutant execution, we verified whether the mutated instruction is actually *executed* (we logged the number of time each mutated line is executed using the Trace module[13] available in the Python library). If the mutated line is not executed, the algorithm obviously provides the same behaviour of the original one. Otherwise, if the mutated line is executed, we can have three different behaviours:

- *same behaviour*: the algorithm provides exactly the same results (i.e., the accuracy is the same);
- *accuracy different from the original*: the algorithm runs without errors but provides a different result;
- *exception raised/error*: the algorithm does not work correctly, raises an exception or an error occurs, and consequently it is not able to provide a result (this category includes also the mutants that must be killed since do not terminate even after waiting a multiple of the time required by the original version).

On the contrary, for answering RQ2, we analysed the accuracy distributions only for the mutants belonging to the category *accuracy different from the original*, i.e., for those mutants whose mutated line is executed at least once and that provide an accuracy different from the original.
RQ3: *Does a bug consistently affect the behaviour of a ML algorithm varying the considered dataset?*

The goal of the third research question is to investigate whether a bug modifies the behaviour of a ML algorithm similarly when we consider different datasets.

To answer this research question, for each mutant and dataset we computed a value, representing its behaviour, in the following way: if the mutant executes without problem we noted down the computed accuracy of the prediction in the range [0,1] (i.e., in the cases: *same behaviour*, *accuracy different from the original*, and *mutated statement not executed*); if the mutant crashes or raises an exception we noted down the value -1. In this way, for each of the eight datasets, we defined a vector with size equal to the number of mutants (respectively 576 for *MLP* and 732 for *LogReg*) that we call "behaviour distribution".

The metrics used to answer RQ3 is the Pearson correlation coefficient[14] computed among the various behaviour distributions. It is a measure of the linear correlation between two variables X and Y. Values are in $[-1, +1]$, where 1 is total positive linear correlation, 0 is no linear correlation, and -1 is total negative linear correlation.
RQ4: *How many are dangerous silent bugs?*

We define a dangerous silent bug as a bug (i.e., a mutation applied to the original algorithm) that drastically affects the accuracy of the predictions only in rare cases while in general allows the algorithm to execute without errors. It can be defined *silent* because it is not easy to pin point (it runs smoothly on several datasets), and, at the same time, it is *dangerous* because, when emerges, it drastically affects the results on a dataset without rising errors or runtime exceptions. Thus, it is very difficult to understand that the poor performance is due to the presence of a bug (indeed, when the algorithm is used on different datasets everything works fine).

The goal of the fourth research question is to understand how many dangerous silent bugs are present among the generated mutants. The metrics used to answer this research question is the percentage of dangerous silent bugs out of the total number of mutants.

4.5 Results

4.5.1 RQ1. Table 1 reports the data used to answer RQ1. Data concerning the *MLP* algorithm is reported in the upper half, while the *LogReg* one is in the lower half. Each column represents a dataset (6 synthetic and 3 real) and the last column shows the aggregate values on all the datasets.

From the table it is evident that a relevant portion of the mutants has the mutated statement not executed, respectively about 39% for MLP and 51% for LogReg. This is due to the fact that in several cases the mutants are in Python methods or in if-then-else branches that are respectively never called or executed. This is reasonable since the mutator distributed the mutants in each portion of the code of the two algorithms. Some of these portions are used to manage: (1) datasets having different characteristics (e.g., some portions of the code of the algorithm can be used to manage multi-class classification tasks and thus they are not executed if the current dataset contains only two classes, i.e., in a binary classification task), (2) peculiar cases or error that in our datasets never occurred, or (3) algorithm settings different from the ones we adopted in this experiment.

[13] https://docs.python.org/3/library/trace.html

[14] https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

		Synthetic Datasets						Real Datasets			Total	
		Single Gaussian	*Double Gaussian*				*BMI*	*Iris*	*Cleveland*	*Pulsar*		
			2	4	6	8						
Multilayer Perceptron	Mutants: Total	576	576	576	576	576	576	576	576	576	5184	
	Not Executed	227	225	225	225	227	227	227	227	225	2035	39,26%
	Executed	349	351	351	351	349	349	349	349	351	3149	60,74%
	Mutants: Same Behaviour	167	165	150	147	134	134	143	130	143	1313	25,33%
	Mutants: Different Behaviour	182	186	201	204	215	215	206	219	208	1836	35,42%
	of which raise an exception	151	151	151	151	154	154	157	157	150	1376	26,54%
	of which show a different accuracy	35	35	50	53	61	61	49	62	58	464	8,95%
Logistic Regression	Mutants: Total	732	732	732	732	732	732	732	732	732	6588	
	Not Executed	375	375	375	375	375	380	380	380	375	3390	51,46%
	Executed	357	357	357	357	357	352	352	352	357	3198	48,54%
	Mutants: Same Behaviour	204	205	189	190	189	161	174	165	174	1651	25,06%
	Mutants: Different Behaviour	153	152	168	167	168	191	178	187	183	1547	23,48%
	of which raise an exception	133	133	133	133	133	138	138	138	133	1212	18,40%
	of which show a different accuracy	20	19	35	34	35	53	40	49	50	335	5,09%

Table 1: Effect of bugs on the behaviour of machine learning algorithms

In the remaining cases, the mutated statement is executed at least once. This brings the mutants to show a different behaviour, w.r.t. the original implementation, in the 35% and 23% of the cases respectively for *MLP* and *LogReg* while in the 25% of the cases, for both datasets, the results are identical (i.e., the mutated algorithm provides exactly the same results of the original one). Among the ones that show a different behaviour, about 2/3 raise an exception/error (precisely 27% and 18% of the total for *MLP* and *LogReg* respectively) while 1/3 execute without problem but the accuracy of the prediction changes (9% and 5% of the total for *MLP* and *LogReg* respectively).

Summary: A bug in the implementation has a relevant (from 2/3 to 3/4) probability of not influencing the behaviour of a ML algorithm (i.e., it is a *silent bug*). This is due to the complexity of the code (i.e., the bug may never be performed) or that the bug has an irrelevant impact (its effect is too small to modify the provided accuracy). In the remaining of the cases (from 1/3 to 1/4) the bug emerges in the form of: runtime error or exception, or modified accuracy of the predictions.

4.5.2 RQ2. Concerning the *MLP* algorithm, Fig. 3 shows the distributions of the accuracy of the mutants that changed the accuracy of the predictions (464 cases in total considering the nine datasets). Concerning the DG_4, DG_6, and DG_8 datasets, the values are quite close to the original accuracy. On the contrary, in the case of *BMI* and of the real datasets, the values are more variable and in several cases the accuracy is drastically affected by the bug introduced in the mutant. It is interesting to note that in 64 cases out of 464 (14%) the accuracy results (slightly) improved by the mutation and that in about an half of the cases (215 out of 464) the accuracy is only slightly affected (i.e., the accuracy of the mutants is between the 67% and the 100% of the original value).

Fig. 4 shows the same kind of data for the *LogReg* algorithm. In this cases, concerning the synthetic datasets, differently from the *MLP* case, the values are lower than the original accuracy. Note that the boxplots in Fig. 3 and Fig. 4 report only the accuracy values for the mutants for which the accuracy of the prediction changes w.r.t.

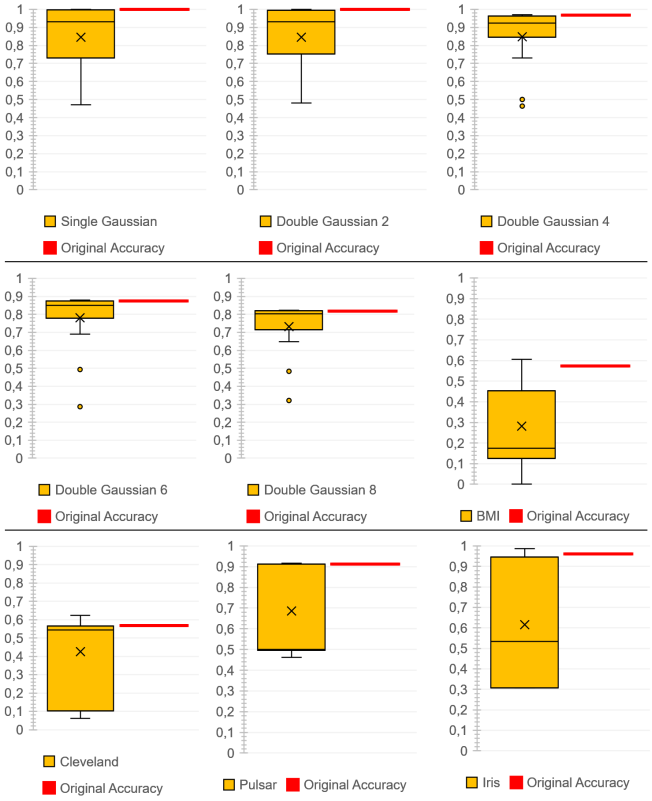

Figure 3: Accuracy distributions of mutants providing different results: *MLP* algorithm

the original algorithm. Thus, in the case of *LogReg*, by comparing the boxplot for DG_2 and DG_4 it could appear that mutants provided better results for DG_4. This is true only considering the mutants that provided different accuracy values (only 19 in DG_2 and 35 in DG_4) and it does not mean that globally all the mutants performed better on DG_4 rather than on DG_2. In the case of the real datasets, the distribution are quite similar to the one obtained for *MLP* except for

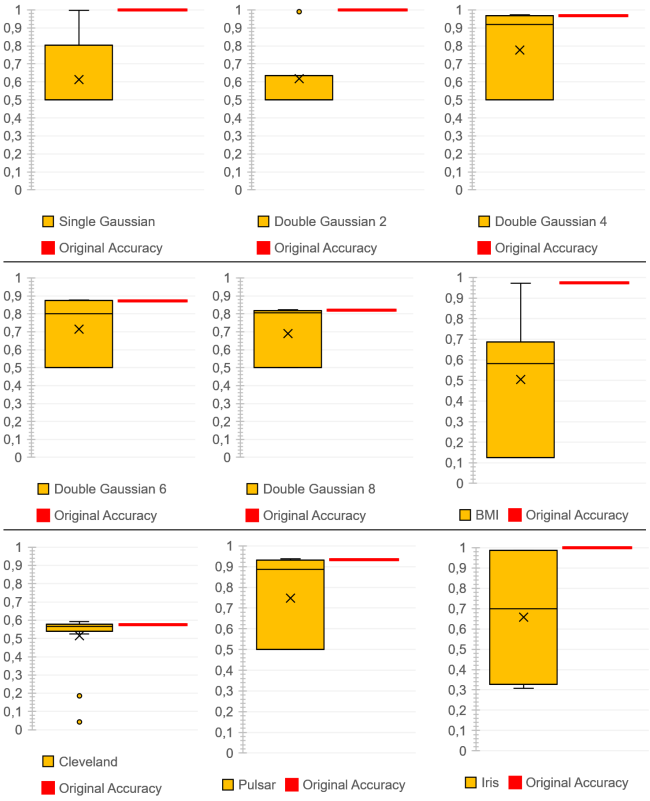

Figure 4: Accuracy distributions of mutants providing different results: *LogReg* algorithm

the *Cleveland* dataset where mutants consistently provided accuracy predictions very close to the original one (except for two outliers). Overall, 48 mutants out of 335 improved the accuracy of the results with respect to the original implementation. In more than one third of the cases (134 out of 335) the accuracy is only slightly affected (i.e., the accuracy of the mutants is between the 67% and the 100% of the original value).

Summary: Focusing on the bugs that modify the accuracy, the interesting fact is that a big portion reduces the accuracy of prediction of less than 1/3. Therefore, on real datasets they can be very difficult to detect by considering only the achieved accuracy level.

4.5.3 RQ3. Table 2 reports, for the *MLP* and *LogReg* algorithms, the Pearson correlation coefficients computed between all the 81 "behaviour distributions" pairs (9x9 matrix with ones on the main diagonal). For instance in Table 2 (left) the cell (BMI,Iris) reports 0.9901. This value represents the Pearson correlation coefficient computed between the distributions of the mutants for the *BMI* and *Iris* datasets (a 576 points array for each dataset). The coefficients have a greener background when the correlation is strong (i.e., close to one), redder when the correlation is weaker (i.e., close to the min value in the table), and white when it is close to the median value. By looking at the tables, it is evident that, in absolute, the correlation is always strong: indeed the min values are 0.9284 for *MLP* and 0.9311 for *LogReg*, where 1 represents a total positive correlation and 0 no correlation. The results can be clustered into two sets: each mutant behaves similarly when working on binary datasets (i.e., *SG*, DG_2, DG_4, DG_6, DG_8, *Pulsar*), and the same holds for multi-class classification tasks (*BMI*, *Iris*, *Cleveland*). This is probably due to the fact that for similar kind of datasets, similar execution paths are performed in the algorithm code.

However, this coarse grain analysis does not allow to pin point possible interesting rare cases. Indeed, even if in general there is a high correlation among the behaviour distributions (one per dataset), we observed that some mutants provide very different results when executed on the various datasets. In particular, 23 and 22 mutants of *MLP* and *LogReg* have a variation, in terms of accuracy, greater than 0.5 (i.e., on some datasets the prediction is close to the original one and in others lower of more than the 50%). The worst case is represented by mutant id=206 at line 129 of the *LogReg* algorithm: on *BMI* the accuracy is reduced of 0.8488 w.r.t. the original algorithm while in the case of *Cleveland* the accuracy is reduced of only the 0.0368.

Summary: In general, there is a high probability that a bug influences the behaviour of an algorithm similarly when working with different datasets. In particular, this holds when the type of classification is similar (i.e., binary or multi-class classifications). Focusing on the provided accuracy values, about a tenth of the bugs provided different results depending on the considered dataset and in about the 4% of the cases the difference can exceed the 50% ranging from excellent to poor results.

4.5.4 RQ4. Starting from the data used to answer RQ3 we focussed our analysis in order to find the candidate *dangerous silent bugs*. Thus, among the mutants that always returned a results (i.e., that

		Synthetic Datasets						Real Datasets		
		Single Gaussian	Double Gaussian 2	Double Gaussian 4	Double Gaussian 6	Double Gaussian 8	BMI	Iris	Cleveland	Pulsar
Synthetic Datasets	Single Gaussian	1,0000	1,0000	0,9999	0,9999	0,9996	0,9285	0,9284	0,9325	0,9911
	Double Gaussian 2	1,0000	1,0000	0,9999	0,9999	0,9996	0,9285	0,9284	0,9325	0,9911
	Double Gaussian 4	0,9999	0,9999	1,0000	0,9999	0,9996	0,9291	0,9294	0,9331	0,9916
	Double Gaussian 6	0,9999	0,9999	0,9999	1,0000	0,9997	0,9287	0,9293	0,9333	0,9912
	Double Gaussian 8	0,9996	0,9996	0,9996	0,9997	1,0000	0,9293	0,9296	0,9334	0,9919
	BMI	0,9285	0,9285	0,9291	0,9287	0,9293	1,0000	0,9901	0,9886	0,9317
Real Datasets	Iris	0,9284	0,9284	0,9294	0,9293	0,9296	0,9901	1,0000	0,9972	0,9322
	Cleveland	0,9325	0,9325	0,9331	0,9333	0,9334	0,9886	0,9972	1,0000	0,9351
	Pulsar	0,9911	0,9911	0,9916	0,9912	0,9919	0,9317	0,9322	0,9351	1,0000

		Synthetic Datasets						Real Datasets		
		Single Gaussian	Double Gaussian 2	Double Gaussian 4	Double Gaussian 6	Double Gaussian 8	BMI	Iris	Cleveland	Pulsar
Synthetic Datasets	Single Gaussian	1,0000	0,9999	0,9999	0,9996	0,9991	0,9329	0,9382	0,9382	0,9988
	Double Gaussian 2	0,9999	1,0000	0,9999	0,9998	0,9993	0,9325	0,9384	0,9384	0,9988
	Double Gaussian 4	0,9999	0,9999	1,0000	0,9999	0,9995	0,9323	0,9385	0,9388	0,9987
	Double Gaussian 6	0,9996	0,9998	0,9999	1,0000	0,9997	0,9319	0,9386	0,9397	0,9986
	Double Gaussian 8	0,9991	0,9993	0,9995	0,9997	1,0000	0,9311	0,9382	0,9401	0,9985
	BMI	0,9328	0,9325	0,9323	0,9319	0,9311	1,0000	0,9938	0,9869	0,9349
Real Datasets	Iris	0,9381	0,9384	0,9385	0,9386	0,9382	0,9938	1,0000	0,9943	0,9391
	Cleveland	0,9382	0,9384	0,9388	0,9397	0,9401	0,9869	0,9943	1,0000	0,9396
	Pulsar	0,9988	0,9988	0,9987	0,9986	0,9985	0,9349	0,9391	0,9396	1,0000

Table 2: Correlation coefficients among behaviour distributions: *MLP* (left) and *LogReg* (right) algorithm

executed without errors on all the datasets) we selected the ones that provided: (1) on only one dataset an accuracy reduction, w.r.t. the original one, of at least the 20% (>= −0.2000, i.e., the problem is, in our opinion, substantial); and (2) on all the other eight datasets an accuracy reduction smaller or equal than the 5% (<= −0.0500, we believe that on these dataset the problem could go unnoticed).

In the case of *MLP*, we found four mutants with these characteristics (4 out of 576, 0.69%), while in *LogReg* they were nine (9 out of 732, 1.23%). An example of mutant with little or no impact on the accuracy for eight datasets, but with a catastrophic impact for the remaining one is the mutant id=300 at line 437 of the *LogReg* algorithm. It changes the line of code containing `if self.solver == 'liblinear'` in `if not self.solver == 'liblinear'`, causing the unexpected execution of some lines (we always used newton-cg as solver). This mutant was unnoticed on the gaussian clouds datasets (i.e., null accuracy variation), had a slight negative impact on the *Iris*, *Cleveland* and *Pulsar* datasets (respectively -0.0267, -0.0003, and -0.0040 accuracy), but had a consistent accuracy decrease (-0.3825) for the BMI dataset.

The two algorithms, *MLP* and *LogReg*, are equipped of unit tests that can be used to validate the correctness of the implementation when a developer commits a new version. We executed them against the mutants to understand if they were able to detect them. In the case of *MLP* all the candidate *dangerous silent bugs* were detected, while in the case of *LogReg* the test cases were not able to detect two of them. The first mutant is id=357 at line 164. It changes an assignment `inter_terms = v[:, -1]` in `inter_terms = v[:, -2]`. This mutant was unnoticed on gaussian clouds datasets and on the *Iris* dataset (i.e., null accuracy variation), had a slight positive impact on Cleveland (+0.0033 accuracy) and a slight negative impact on Pulsar (-0.0030 accuracy), but caused a significant accuracy decrease (-0.3763) for *BMI*. The second mutant is id=209 at line 169. It changes an assignment `r_yhat -= inter_terms` in `r_yhat += inter_terms`. This mutant was unnoticed on the *SG*, DG_2 and DG_6 datasets (i.e., null accuracy variation), had a slight positive impact on *Cleveland* (+0.0067 accuracy), a slight negative impact on the DG_4, DG_8, *Iris*, and *Pulsar* datasets (respectively -0.0010, -0.0010, -0.0133, and -0.0090 accuracy), but had a significant accuracy decrease (-0.3838) for the *BMI* dataset.

Summary: About 1% of the bugs introduced can be defined as *dangerous silent bugs*. Since they manifest themselves on only a dataset (i.e., in rare cases), they are very difficult to detect but potentially, in real cases, they can drastically affect the quality of the predictions. In a few cases, even the official test suite used to check the quality of the algorithm implementation was not able to detect these dangerous silent bugs.

4.6 Considerations

At a first sight, it could appear that in general bugs probably do not drastically affect the quality of the predictions or that they can be easily detected thanks to runtime errors or exceptions. However, the existence of the dangerous silent bugs is a relevant problem. Few of them emerged in our study but it is important to highlight that other bugs of this kind can be present among the remaining mutants. In fact, each bug that does not clearly manifest itself with strong effects on multiple datasets could be a potential dangerous silent bug on other datasets. In our study, *dangerous silent bugs can be still present* among the mutants that: (1) contain a mutated line that is not executed using our datasets (602 out of 1308 overall); (2) do not modify the accuracy of the considered datasets (371 out of 1308), or (3) that modify the accuracy only slightly (e.g., 25 out of 1308 mutants affect the accuracy of no more of 5% in the worst case on all the eight datasets). They could be found only using datasets with different characteristics or different algorithms' settings. Note that in general 279 out of 1308 mutants are also not killed by the official algorithm's test suites.

Finally, the definition we adopted of dangerous silent bugs is quite stringent (they should clearly manifest on only one dataset). We have several additional cases in which a bug drastically affects the results of only two datasets. If we had not considered in our experiment one of the two datasets, suddenly they would have classified as dangerous silent bugs.

Summary: The fact that a considerable percentage of the injected bugs basically does not affect the behaviour of the algorithms, poses a considerable problem: among them several other *dangerous silent bugs* could be present. In a real scenario, these bugs could emerge only when the corresponding implementation is adopted on a new dataset, for instance as a part of a safety critical system. This poses a ***serious problem*** about which techniques should be used to detect them and thus to ensure the quality of ML-based software.

4.7 Threats to Validity

In this section we briefly sketch some of the major threats of this study.

Differently from many empirical studies, we are not interested to obtain the best performance from a proposed algorithm or technique, but to understand how ML algorithms behave in presence of bugs. Thus, we are aware that the settings used to run the algorithms can be considered suboptimal and in some cases affect the absolute performance of the predictions. Adopting better parameters of the algorithms could have modified the observed behaviour, however we believe that: (1) in general the obtained accuracies are quite good (see the red lines in Fig. 3 and 4), so the choice of the parameters can be considered acceptable; and (2) ML library users are often not super-expert of the selected algorithms so our choices could be similar to the one of a typical user.

Concerning the generalization of results, we selected two real open source machine learning algorithms and three real datasets, which makes the context realistic, even though further studies with other algorithms/datasets are necessary to corroborate the obtained results and to understand whether the observed behaviour is similar also in other cases. Also the kind of mutator operations applied to the original algorithms cannot fully represent the plethora of possible bugs that a developer can insert in the implementation. Extending the set of considered bugs with additional, more complex errors could change the distribution of the behaviours among the various classes but could also help to find even more dangerous silent bugs and so to strengthen our results concerning the strong need of novel techniques for software quality assurance of ML algorithms.

The thresholds (20% and 5%, see the results of RQ4) used for selecting the bugs that we defined both as dangerous and silent, from the complete list of bugs are arbitrary. Changing these thresholds would obviously lead to find a different number of dangerous silent bugs. However, in our opinion the thresholds are reasonable and the found bugs are potentially really dangerous (in several cases the selected bugs show an accuracy reductions close to the 40%).

5 CONCLUSIONS AND FUTURE WORK

In this paper, we analysed the influence of implementation bugs on the behaviour of ML algorithms. We believe that the results of our study can be interesting for two categories of people. (1) Final users of ML algorithms since they should be aware of the possible problems that could arise if the chosen implementation is not carefully selected. The problem is becoming more and more relevant since the proliferation of ML implementations freely available on the web is increasing. (2) Researchers working on quality assurance of ML-based software since understanding the behaviour of such systems in presence of bugs is a fundamental prerequisite for conceiving techniques for detecting and then removing such quality problems. For this reason, our work can be considered a first useful step towards the definition of novel strategies and techniques for assuring the quality of ML-based solutions.

Our study clearly shows that a small fraction of the considered bugs exhibits a behaviour that can be defined as both *dangerous* and *silent*. Indeed, such bugs manifest themselves, with strong negative effects (depending on the context of usage of the algorithm this could have dangerous consequences) in rare cases, and thus they are very difficult to detect. The fact that a considerable percentage of the bugs basically does not affect the behaviour of the algorithms, considering the nine datasets, poses a considerable problem: indeed, among them, several other *dangerous silent bugs* could be present. They could emerge when the algorithms implementations are used on a novel dataset.

In our future work, we plan to replicate this study by: (1) considering more complex mutations in order to include other categories of bugs; in this way we will be able to simulate other categories of errors that ML developers can commit (e.g., calling a wrong function, exchange the value of variables, etc.); (2) considering other ML algorithms; (3) considering other real datasets with different characteristics. Moreover, since in the case of open source implementations of ML algorithms all the versions are available on the online software repositories, we plan to select the versions preceding the various documented bugs fixes in order to analyse the effects of real bugs on the behaviour of the algorithms (i.e., using implementations including real recognized bugs instead of generated mutants).

ACKNOWLEDGMENTS

This research was partially supported by Actelion Pharmaceuticals Italia and DIBRIS SEED 2018 grants.

REFERENCES

[1] 1998. Glossary of Terms. *Mach. Learn.* 30, 2-3 (Feb. 1998), 271–274. http://dl.acm.org/citation.cfm?id=288808.288815
[2] Kieran Alden and Mark Read. 2013. Scientific software needs quality control. *Nature* 502 (2013), 448. https://doi.org/10.1038/502448d
[3] Tsong Yueh Chen, Samson Cheung, and Siu Ming Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases.* Technical Report HKUST-CS98-01. Department of Computer Science, The Hong Kong University of Science and Technology.
[4] Tsong Yueh Chen, Joshua WK Ho, Huai Liu, and Xiaoyuan Xie. 2009. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics* 10, 1 (2009), 24. https://doi.org/10.1186/1471-2105-10-24
[5] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *Comput. Surveys* 51, 1, Article 4 (2018), 27 pages. https://doi.org/10.1145/3143561
[6] Martin D. Davis and Elaine J. Weyuker. 1981. Pseudo-oracles for Non-testable Programs. In *Proceedings of ACM 1981 Conference (ACM 1981)*. ACM, New York, NY, USA, 254–257. https://doi.org/10.1145/800175.809889
[7] Richard Hamlet. 2002. Random Testing. In *Encyclopedia of Software Engineering*. John Wiley & Sons. https://doi.org/10.1002/0471028959.sof268
[8] Erika Check Hayden et al. 2013. Mozilla plan seeks to debug scientific code. *Nature* 501 (2013), 472. https://doi.org/10.1038/501472a
[9] J. C. Knight and N. G. Leveson. 1986. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering* SE-12, 1 (1986), 96–109. https://doi.org/10.1109/TSE.1986.6312924
[10] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. 2015. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *Proceedings of 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER 2015)*. IEEE, 560–564. https://doi.org/10.1109/SANER.2015.7081877
[11] Deepika Mamnani. 2017. *Testing of machine learning systems – The new must have skill in 2018.* Technical Report. CapGemini. https://www.capgemini.com/2017/12/must-have-testing-skills-in-2018.
[12] L. I. Manolache and D. G. Kourie. 2001. Software Testing Using Model Programs. *Software: Practice and Experience* 31, 13 (2001), 1211–1236. https://doi.org/10.1002/spe.409
[13] S. Masuda, K. Ono, T. Yasue, and N. Hosokawa. 2018. A Survey of Software Quality for Machine Learning Applications. In *Proceedings of International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2018)*. 279–284. https://doi.org/10.1109/ICSTW.2018.00061
[14] Z. Merali. 2010. Computational science: ...Error ... why scientific programming does not compute. *Nature* 467 (2010), 775 – 777. https://doi.org/10.1038/467775a
[15] A Jefferson Offutt and Roland H Untch. 2001. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*. Advances in Database Systems, Vol. 24. Springer, 34–44. https://doi.org/10.1007/978-1-4757-5939-6_7
[16] S.P. Parker. 2003. *McGraw-Hill Dictionary of Scientific and Technical Terms*. McGraw-Hill Education.
[17] 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. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[18] David S. Rosenblum. 1995. A Practical Approach to Programming With Assertions. *IEEE Transactions on Software Engineering* 21, 1 (1995), 19–31. https://doi.org/10.1109/32.341844
[19] Elaine J. Weyuker. 1982. On Testing Non-Testable Programs. *Comput. J.* 25, 4 (1982), 465–470. https://doi.org/10.1093/comjnl/25.4.465
[20] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. 2012. *Experimentation in Software Engineering*. Springer. https://doi.org/10.1007/978-3-642-29044-2
[21] Xiaoyuan Xie, Joshua W. K. Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. 2011. Testing and Validating Machine Learning Classifiers by Metamorphic Testing. *Journal of Systems and Software* 84, 4 (2011), 544–558. https://doi.org/10.1016/j.jss.2010.11.920