TEST DATA GENERATION USING GENETIC ALGORITHMS AND INFORMATION CONTENT

Ciprian-Ionut NUTESCU[1], Mariana MOCANU[2]

An important development cycle for a software product is represented by the quality assurance phase. In this phase, the software is testing to find out any defects or issue that might go unresolved into production. Testing usually needs data that simulates production scenarios. This data is generated manually by quality assurance engineers or taken from production and transformed into testing data samples. Even if the process of testing usually is automated using testing frameworks, data generation for these tests is done manually by quality assurance engineers. In this paper, we propose a solution to automatically generate testing data sets from small groups of samples using a modified machine learning genetic algorithm and information content fitness testing. We will describe the algorithm found, its hyper-parameters for tuning it for proper results and the performance of the algorithm using various scenarios. In the last chapter of this paper, we will discuss further improvements that can be done for this algorithm.

Keywords: Test data management, Genetic algorithms, Machine Learning, Information content

1. Introduction

Testing is a very important phase in a software development lifecycle because it checks if the functional and technical specifications of a software product are met after the development phase. A big challenge for software testing is gathering and preparing the necessary data for all the test cases. This process is called test data management. Data provided for testing is usually generated manually or cloned from production and cleaned manually. This process takes time and human resources and it is prone to errors due to the human factor. Even more, being a manual process, the testing data samples generated usually are not enough to cover the whole test case with all scenarios. The current solutions in the markets (TDM from Informatica[3]) are generating data using hand-made data schemas and dictionaries for the values of the data and randomizing various combinations without taking into consideration data inference (there is no inherence link between values of the data). In this process, we aim to create a new algorithm based on machine learning that could generate test case samples

[1] PhD student Eng., Department of Computer Science, Automatic and Computer Science Faculty, University POLITEHNICA of Bucharest, Romania, e-mail: cipinutescu@yahoo.com
[2] Prof., Department of Computer Science, Automatic and Computer Science Faculty, University POLITEHNICA of Bucharest, Romania, e-mail: mariana.mocanu@cs.pub.ro

without the intervention of the user. The algorithm is based on genetic algorithm and information theory. It automatically computes the schema of the data and its dictionary and generates relevant data from the smaller dataset taking into consideration data inherence. It uses information content as a fitness function for the genetic algorithm to determinate the relevancy and usefulness of the testing samples. [1]

2. Test data management

Test data management refers to all actions that are done to acquire and produce testing data for a software application. Even if the process of testing is automated (more and more companies are starting to use automatic testing software instead of manual testing) or manual (there are people who manual test the application to see if the requirements are meet and all issues are fixed), the provision of data is a manual process done by people. A study [2] conducted by IBM in 2016 shows that 64% of the data is created manually and 34% of the data is provided by cloning or manually cleansing data sets. The rest of 2% is done by other means. The Fig. 1[2] shows that it is undeniable evidence that data preparation is a time-consuming phase of software testing. As said before, even if the process of testing (we are referring here to the phase when the tests are run) is totally automated (using various software testing frameworks), the acquisition and preparation of data are done manually by quality assurance engineers and it is a time-consuming phase of the software testing.

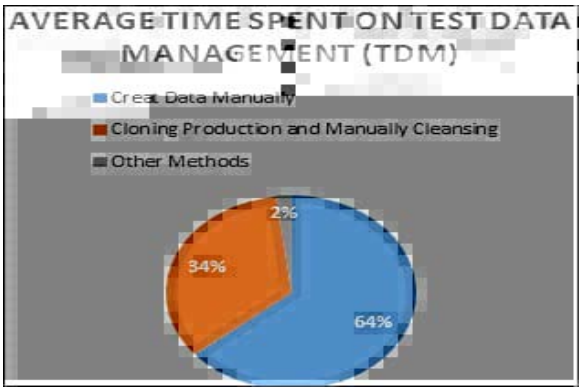

Fig. 1. Average time spent on test data management

3. The automatic generation of test samples

Nowadays most of the companies are using automatic testing frameworks to run tests each when needed, but the testing data is provided manually by the quality assurance engineers. The question is what if we can provide an automatic

method to generate the sample data for the tests that can be used in any situation regard-less of the testing data form. We propose to solve this problem with modified genetic algorithm having as input some testing sample (few in number) that are able to generate whole datasets from those samples without user interaction and without any knowledge of the domain in which the testing is done (we are refer-ring here to the domain where the software application resides: medicine, military, education, financials, etc.).

We are able to propose this solution because the technology has advanced a lot during last year's regarding artificial intelligence and machine learning. We tried to find some solution of algorithms from existing machine learning ones, but we were not able to find anything that can solve our problem straight forward. The most adjacent regarding solution generation in all machine learning algorithms (supervised or unsupervised) are the genetic algorithms that converge to a solution for a problem by generating various solution of that problem and then testing the solution using a fitness function (that depends from problem to problem) and then regenerating the candidate solutions using cross-over and mutation if no solution was found in the current iteration. An iteration in genetic algorithms is called a generation and possible solution generated is called an individual. Each individual (possible solution to the problem) has a DNA-like representation called chromosome. The chromosome represents the solution parameters represented in an array or DNA like form (for the algorithm to able to apply cross-over and mutation). We choose the genetic algorithms as a starting point because of their ability to encapsulate various solution candidates in chromosome representation and use operate with this representation and the generation potential of candidates' solutions of the algorithms.

4. Genetic algorithms

As presented in the last chapter, genetic algorithms iterate sequentially through generations of individuals until the most fitted individual is found or the iterative process stops from various reasons (maximum number of iterations, maximum number or resources consumed, etc.), case in which a fitted individual is found that solves the problem with an accounted error. In fig. 2. we have the general genetic algorithms phases that we will use to build our own modified algorithm for testing samples generation.

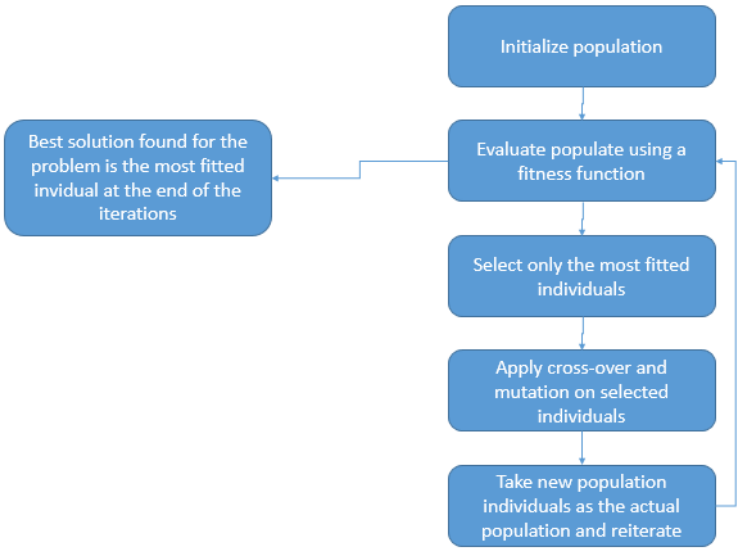

Fig. 2. Genetic algorithm phases

5. Information content

The information content or information entropy in our solution represents a fitness measurement to determinate how fitted are the individuals generated by the genetic algorithm. Let's consider the following:

$$O = \{o_1, o_2, o_3, ..., o_n\}, \text{ the initial object population} \quad (1)$$

$$A = \{a_1, a_2, a_3, ..., a_n\}, \text{ set of object attributes} \quad (2)$$

$$V = \{v_1, v_2, v_3, ..., v_n\}, \text{ set of values for attribute a} \quad (3)$$

$\theta(a,vi)$ = the function that counts the number of occurrences where attribute a has value vi in our initial population

The probability that an attribute will have the value v_i will be given by the following formula (which is the information content):

$$p(a, v_i) = \frac{\theta(a,v_i)}{\sum_{j=1}^{j=n} \theta(a,v_j)} \quad (4)$$

where p_{xi} is the probability of x_i ($X = \{x_1\ x_2, x_3, x_4, ...\}$).

Combining the two formulas, we get the information content of an object o from the initial population O:

$$I(o) = -\sum_{i=1}^{i=n} p(a_i, v_i) * \log p(a_i, v_i) \tag{5}$$

$$I(o) = -\sum_{i=1}^{i=n} \frac{\theta(a_i, v_i)}{\sum_{j=1}^{j=n} \theta(a_i, v_j)} * \log \frac{\theta(a_i, v_i)}{\sum_{j=1}^{j=n} \theta(a_i, v_j)} \tag{6}$$

Using this formula, we will determinate how relevant and how much information we can obtain from our generated individuals (test samples). If the information content of the individual is lower than a value that we are choosing called minimum information content threshold, then the individual is rejected from the future generation is the selection phase, otherwise, he is chosen to carry on.

6. The algorithm

In this chapter, we will present the final algorithm that will be used to produce testing samples. As it was presented before, we are using a modified genetic algorithm to generate the samples. The pseudocode of the algorithm is the following:

phase 1 initializations:

training_data_set = {first generation of the object received as input from the user featured as individuals for input}

current_generation = 0

probability_of_mutation = 0.3

minimum_information_content_threshold = 0.2

maximum_generation = 3

information_content_map = {<key1, value1> pairs, where key1 is the attribute of the object and value1 is another map <key2, value2> where key2 is on value of the attribute and value2 the information content for that value of the attribute}

phase 2 iterations:

```
testing_sample_generation (training_data_set, current_generation) {
        if (current_generation is equal to maximum_generation){
                return training_data_set
  }
  new_generation = empty_set
  for_each pair <p1, p2> in (training_data_set x training_data_set){
```

```
                    apply mutation for cross_over p1, p2
                compute information_content based on formula and
information_content_map

                if ( information_content of the individual >=
minimum_information_content_threshold ){
                        add above result in new_generation
            }
          }
          testing_sample_generation(new_generation, current_generation+1)
        }
```

The first phase initialization is where all the algorithm parameters are chosen and other auxiliary data structured is processed. The training data set is the input set of an object that will be used as archetypes for the new samples testing objects to be generated from. These objects are featured as a map of string – object, where the string is the name of the attribute and object is the value. This feature method is used because we want our algorithm to be able to process any kind of objects. The parameter current_generation is initialized with the value 0 and it will iterate with each recursive call of the algorithm until it is equal to the maximum_generation parameter. This parameter is used for maximum iterations of the algorithm for the training data set. The parameter probability_of_mutation is a genetic algorithm parameter that gives the probability of a mutation happening after the cross-over process is applied to an individual's pair. A very important auxiliary data structure processed in this step is the information_content_map. This map contains the information content for each value for each attribute. We need this kind of information to apply our formula on each generated individual to see how fitted it is. A very interesting question we ask ourselves when we de-sign the algorithm is why we compute this map at the very beginning and not at each iteration. The answer is justified by the nature of the information content computing method: by counting occurrences. We want to generate testing samples that will have the same value for the testing scenarios the same as the initial samples. If the information_content_map is recomputed at each step, then it will add the new occurrences from the newly created generations. This means that some values of attributes could appear often (we are using a probability of apparitions because the whole genetic algorithm process is based on random numbers generations) and will have a higher information content than others.

For example, if our proposed algorithm takes the attribute currency with the values USD, EUR, and RON. In our initial samples, we have USD 4 times, EUR 2 times and RON 1 time. We want that further generated testing samples to

have the same distribution of values as the initial ones. If at one step the value RON will appear 10 times, and by any changes the individuals holding this value will iterate into the new generation, at the next iteration of our algorithm by re-computing the information_content_map we will have value RON bringing more value than value USD, which is incorrect because we don't want to mess the initial distribu-tion and information content of the samples.

The second phase of the algorithm is the iterative genetic part that represents recursively calling testing_sample_generation method until the current_generation parameter is equal to the maximum_generations parameter. The stopping condition is checked, and if so then the generation received as a parameter is returned as a result because there is no need to iterate more. If the stopping condition is not met, then for each pair of individuals taken from the past generation (received as a parameter), we apply cross-over process and then we apply mutation with a probability of mutation_probability parameter. We apply cross-over for each pair instead of using others existing methods (tournament selection, roulette wheel selection, etc..) because we want to maximize the data combination between individuals (we don't want to lose any data combination because the result of cross-over can contain high information content values). A new individual is created from this process that will be testing in our fitness function. The fitness function is represented by computing the information content of the newly generated individual and comparing it with the minimum_information_content_threshold. If it passes, then we will be added in the new generation and it will help at generating new individuals.

7. Testing and experimental results

We implemented the testing samples generation algorithm in Java using only the SDK without any other external libraries. The software had the following inputs:

- input testing samples representing first generation
- information content threshold
- mutation probability
- maximum generations (iterations)

First, we are testing how the time of the processing is influenced by the input testing samples and the information content threshold.

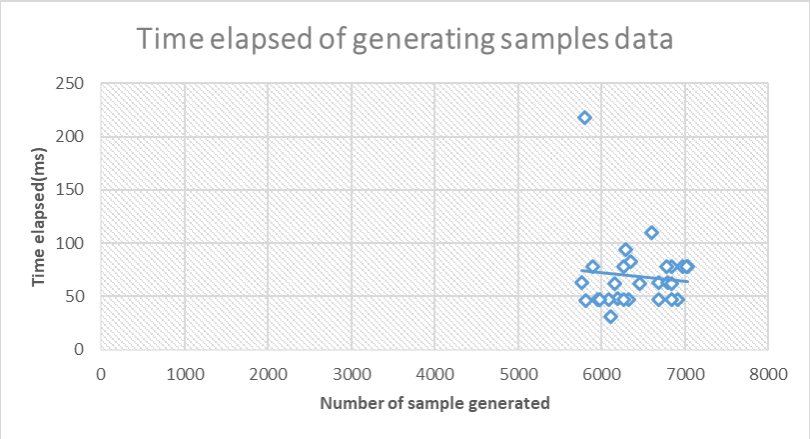

Fig. 3. Time elapsed of generating samples data for information content threshold 0.25, maximum generation 3, initial testing samples 10

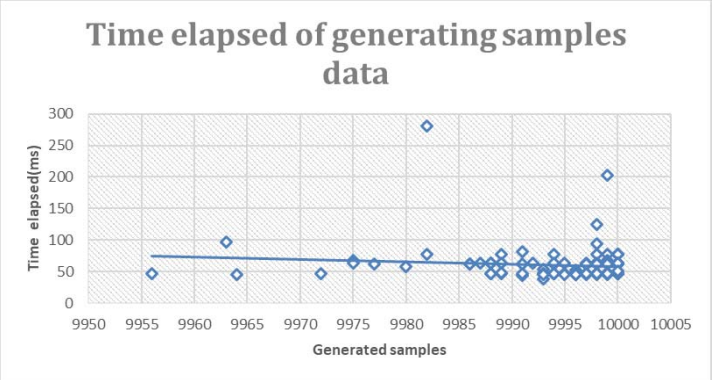

Fig. 4. Time elapsed of generating samples data for information content threshold 0.23, maximum generation 3, initial testing samples 10

Between Fig. 3 and Fig. 4, the only modified parameter is the information content threshold that is 0.25 for the first figure and 0.23 for the second figure. In other words, on the first run, we have been more restrictive with the algorithm having only testing samples with information content higher than 0.25 been generated, while in the second run we have been less restrictive. For the first case, we can see that the points on the graph (Fig. 3) are grouped in a cluster meaning that the algorithm on 100 repetitive calls has generated 6000-7000 samples of data (out of a maximum of 10000 samples of data, explained in table 1), having the processing time around 50-100 milliseconds. This gives a deterministic behavior to our algorithm. For the second case, we have done the same experiment having the information content threshold low-ered by 0.02.

Table 1

Maximum sample generated determined by current generation and number of initial samples

Number of initial samples	Current generation	Maximum samples generated
10	1	10
10	2	100
10	3	10000
100	1	100
100	2	10000
100	3	100000000

This modified the behavior of our algorithm, having most of the data centered around 9985 – 10000 point generated (the maximum being 10000) and with processing time around 50-100 milliseconds. The processing time has not been modified, but the number of generated points has increased from around 6000-7000 to 9000-10000. This means that the number of generated testing samples is correlated with the information content threshold. We will redo the same scenarios, but with 100 initial testing samples instead of 10.

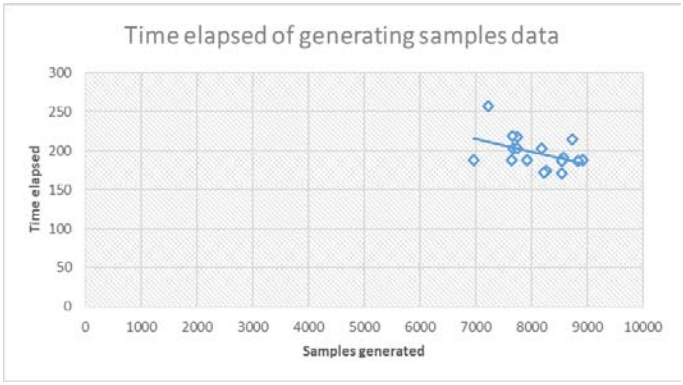

Fig. 1. Time elapsed of generating samples data for information content threshold 0.25, maximum generation 3, initial testing samples 100

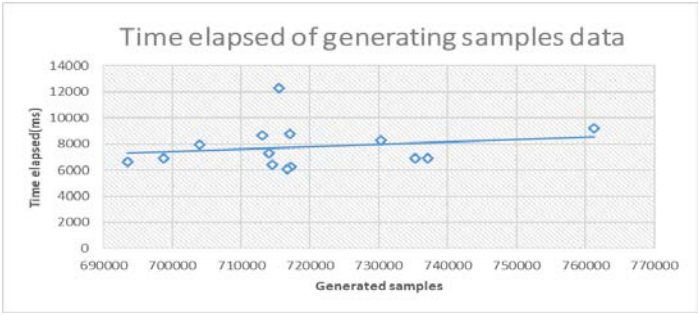

Fig. 6. Time elapsed of generating samples data for information content threshold 0.23, maximum generation 3, initial testing samples 100

In Fig. 5 and figure 6, the algorithm has been run 100 times, having an initial number of samples 100 and information content 0.25 and 0.23. In the first case with information content threshold 0.25, the algorithm has generated around 7000-9000 samples in about 150-250 milliseconds, while in the second case with information content 0.23 the algorithm has generated 700000-740000 samples in about 6-12 seconds. We can see that the time of processing has increased considerably from milliseconds to seconds if we need to generate thousands of samples data. For the Fig. 5, we have the same behavior as in Fig. 3, when the algo-rithm was used for the same parameters but with 10 initial testing samples: on repetitive runs, the algorithm has a deterministic behavior given by the cluster of runs of the graphic. For Fig. 4 and Fig. 6, where we used 10 and 100 initial testing samples, with information content 2.3, the algorithm tends to generate more testing samples approaching to the maximum number of the testing sample generated (given by the number of generations and number of initial samples, see table 1). This means that with the information content threshold we can decide if we need the algorithm to be more restrictive or not.

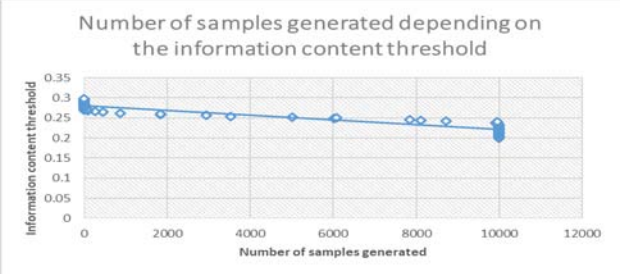

Fig. 7. The number of samples generated depending on the information content threshold with initial testing samples 10 and maximum generations 3

In Fig. 7, the algorithm has been tested using an information content threshold from 0.2 to 0.3. As seen from the testing results, if the information content threshold is chosen to be very restrictive 0.3, the algorithm starts generating 0 samples of data, and if the information content threshold is chosen to be less restrictive 0.2, the algorithm generated the maximum number of samples possible.

The number of samples the algorithm is able to generate is given by the initial testing samples number, the number of generation we want to run (it will give the iterations of the algorithm) and the information content threshold. Even if we chose to have a big set of initial testing samples, having a very restrictive thresh-old of information content could make the algorithm output 0 samples of testing data. This means that the following parameters: number of generations and in-formation content threshold must be chosen very carefully after a series of testing and hyper-parameter tuning sessions. We deduced that a 0.25 information

content can be restrictive but it can still generate testing samples, while 0.3 generated 0 samples and 0.2 generated all possible testing samples without any fitness testing. So the recommendation is to take the information content 0.23-0.25, while the maximum generations should be taken into consideration depending on a number of initial testing samples. As presented in table 1, the maximum number of the sample generated by the algorithm S is:

$$S = m^{2^n} \tag{1}$$

where *m* is a number of initial samples and n is the number of maximum generations. The testing platform used for the algorithm was Windows 10 Pro 64-bit, Intel i7-6500 CPU, 8 GB Ram.

8. Comparison with existing TDM solutions

The only solutions we found in the market comparable with ours (generating datasets) were Informatica TDM[3] and Broadcom Test Data Manager[4]. Both solutions required manual interaction and creation of the data schema (they mainly integrate only to databases, not having the ability to generate testing data from a json, xml or csv file). The user has to complete the dictionary of what values can a column have in the Informatica case (they generate from scratch the dataset, randomizing combinations between data values), but in the case of Broadcom TDM the software creates automatically the dictionary where you can add more values (they generate from existing dataset only, but randomizing all combinations between data values). Both solutions don't take into consideration data inherence (what information links can exist between various values of data, generating irrelevant data for testing). There is no state-of-art in the existing solutions (not using machine learning or big data techniques, only full random combination between values in the dictionaries).

We tried to overcome those problems in our paper by minimizing the user interaction with the software, automatically computing data schema and encapsulation in chromosome for the genetic algorithms (the solution proposed can take any data into consideration, from SQL tables to json/xml/csv files). Another problem we wanted is taking in consideration data inherence; in our solution data generated which is not relevant for the testing scenarios is dropped.

The state-of-art of this paper is using an existing machine learning algorithm used for a certain purpose and using its logic and inherence computation to serve another purpose.

9. Conclusions and future improvements

The algorithm presented in this paper can be used to generated relevant testing data for different (research, financial, medical, etc...) applications and can

be implemented in various programming languages because it is not depending on certain libraries or platforms. The most important parameters of the algorithm are a number of generations and information content threshold as demonstrated in the section above because with these two parameters the algorithm can be controlled to generated more testing samples or less testing samples with relevancy proximally equal to the input testing samples. The algorithm is capable of generating in 10 seconds approximately 760000 testing samples, but this performance can be improved by migrating the algorithm from a serial approach using on a single thread to a multithreaded approach. The section that can be very easily parallelized is the new individual cross-over, mutation and fitness testing, while for each generation iteration is needed a synchronization barrier so all the elements have been generated and tested because the data dependency, in this case, is between individuals from different generations, not the same generation

R E F E R E N C E S

[1]. Test Data Management in the New Era of Computing, Vinod Khader IBM InfoSphere Optim Development, IBM Technical Summit

[2]. 2. A Comprehensive Test Data Design and Management Guide, June 7, 2018, https://www.softwaretestinghelp.com/tips-to-design-test-data-before-executing-your-test-cases.

[3].https://docs.informatica.com/data-security-group/test-data-management/10-2-0-hotfix-2/getting-started-guide/glossary/glossary-of-terms/test-data-management--tdm-.html

[4]. https://www.broadcom.com/products/software/continuous-testing/test-data-manager