Phenometric - Implementing Genetic Algorithms to Optimize Machine Learning Classifiers for Flaky Test Detection

Michelangelo Esposito
Department of Computer Science
University of Salerno
Italy
m.esposito253@studenti.unisa.it

Roberto Calabrese
Department of Computer Science
University of Salerno
Italy
r.calabrese21@studenti.unisa.it

Antonio De Luca
Department of Computer Science
University of Salerno
Italy
a.deluca72@studenti.unisa.it

1 CONTEXT OF THE PROJECT

Testing represents one of the most crucial steps of software development, especially with today's focus on quickly delivering new features and additions to existing software solutions.

Test cases allow developers to extensively examine and stress their code, but what happens when non-determinism becomes a factor in a test case's outcome? A flaky test is a test that both passes and fails periodically without any code changes. Even though flaky tests are often neglected by developers, the problem is very common, especially in large companies like Spotify [1]. Flaky tests can be quite costly since they often require engineers to retrigger entire builds in CI and often waste a lot of time waiting for new builds to complete successfully. The most common causes associated with test flakiness are the following:

- **Asynchronous wait** (45%): developers often write tests that need to wait for something else to complete. Many flaky tests use sleep statements to do the waiting. The problem with sleep statements, however, is that they are imprecise. If a test sleeps for 30 seconds, it may pass most of the times (if the processing takes less than 30s most of the time) but in other environments or under different circumstances, it may take longer, and the test will fail.
- **Concurrency** (20%): asynchronous wait is just an example of a broader category: concurrency. This category includes tests that are flaky because of other kinds of concurrency issues, such as data races, atomicity violations or deadlocks. These tests are usually flaky because the developer made an incorrect assumption about the ordering of operations being performed by different threads.
- **Test Order Dependency** (12%): a good test should be isolated and should set up the state it depends on in an explicit way. However, in practice, many tests make implicit assumptions about the shared state (i.e., memory, database, files, etc.) without doing anything to enforce those assumptions. There are two ways in which this kind of flaky tests can manifest. A test t1 can either depend on another test t2 not to execute before it (because t2 changes the state that t1 expects) or, conversely, t1 can depend on another test t3 to have to execute before it (because t3 sets up the state that t1 expects). If this order is not preserved then t1 will fail, otherwise it will succeed.

Besides these three main categories, the following one also contribute, although at a smaller scale, to test flakiness:

- **Resource leak**: tests can be flaky if the application or the tests do not properly acquire and release resources, such as memory, file streams or database connections). To avoid such issues developers should consider using resource pools and make sure that resources are returned to the pool when they are no longer needed.
- **Network**: test can be flaky because of dependencies on network operations. The best fix is to mock the remote services.
- **Time**: tests can be flaky if they depend on system time. System time is highly platform-dependent, and it is better to avoid it if possible.
- **I/O operations**: tests can be flaky because of I/O operations (besides network). This category is strongly related to concurrency and resource leak.
- **Randomness**: tests can be flaky if they use random number generators without accounting for the full range of possible results. In this case, the seeding uses for the generator should be controlled.
- **Floating point operations**: floating point operations are by nature imprecise and require careful attention to avoid underflows, overflows, or issues with non-associative addition.
- **Unordered collections**: tests can be flaky if they make assumptions about the order of elements in an unordered collection of objects. The solution is to simply check for the

existence of the element we are accessing, without making any assumption about its position.

Hence, identifying an approach to detect flakiness in test cases could potentially reduce the time wasted on failed builds and on test refactoring. An existing and popular solution to the flakiness problem is DeFlaker [2]. DeFlaker detects flaky tests in a three-phase process; in the first phase, differential coverage analysis, DeFlaker uses a syntactic diff from VCS and an Abstract Syntax Tree builder to identify a list of changes to track for each program source file. In the second phase, coverage collection, DeFlaker injects itself into the test-execution process, monitoring the coverage of each change identified in the prior phase. Finally, once tests have finished executing, DeFlaker analyses the coverage information and tests outcomes to determine the set of tests failures that are likely flaky.

2 GOALS OF THE PROJECT

The main goal of this project is to develop techniques to identify the presence of flaky tests in source code by employing various machine learning classifiers on a dataset of test case characteristics. Secondly, we are interested in determining whether the usage of genetic algorithms can enhance the performances of these predictors by a non-ignorable margin, and also when compared with traditional statistical feature selection methods, such as information gain. To achieve this, we focused on developing the following two approaches:

- A machine learning model capable of identifying flaky test cases based on a series of parameters, including lines of code of the test, java keywords and a series of text identifiers. During this phase we followed the results obtained in [3]
- A genetic algorithm that, given an initial set of different feature configurations for a classifier cases, can find the optimal subset of attributes to strengthen the performances of the predictor by using selection-inspired search operations.

3 METHODOLOGICAL STEPS CONDUCTED TO ADDRESS THE GOALS

3.1 Data pre-processing

We constructed our dataset based on the DeFlaker benchmark [4] with 1402 flaky test cases and 44819 non-flaky test cases. Table 1 shows the 24 projects from which we retrieved the test cases.

Table 1: Projects used

project	GitHub ID
achilles	doanduyhai/Achilles.git
alluxio	Alluxio/alluxio.git
ambari	apache/ambari.git
assertj-core	joel-costigliola/assertj-core.git
checkstyle	checkstyle/checkstyle.git
commons-exec	apache/commons-exec.git
dropwizard	dropwizard/dropwizard.git
hadoop	apache/hadoop.git
handlebars	jknack/handlebars.java.git
hbase	apache/hbase.git
hector	hector-client/hector.git
httpcore	apache/httpcore.git
jackrabbit-oak	apache/jackrabbit-oak.git
jimfs	google/jimfs.git
logback	qos-ch/logback.git
ninja	ninjaframework/ninja.git
okhttp	square/okhttp.git
oozie	apache/oozie.git
oryx	OryxProject/oryx.git
spring-boot	spring-projects/spring-boot.git
togglz	togglz/togglz.git
undertow	undertow-io/undertow.git
wro4j	wro4j/wro4j.git
zxing	zxing/zxing.git

First, we made sure that the non-flaky tests were non-flaky, so we consolidated the results of the 100 reruns done for each test case using the rerun log files.

At this stage, the balance of the two classes was way off; this would cause any machine learning model to develop a heavy bias towards non-flaky test cases. To balance the number of flaky and non-flaky tests we decided to choose only 1402 non-flaky tests out of the available ones. Choosing them randomly would still not guarantee a proper balance among the test case contents, so we considered the median size of the flaky test cases, then we randomly picked a non-flaky test with size above

the median and another with size below it. This process was repeated until we reached a balance in the two classes.

Once these two lists of files were available, mining operations were performed to extract from the test cases the various features on which to build our dataset for the classification. First, various tokens were extracted from each test case by using several natural language processing techniques, such as word stemming, lower case reduction, term separation and camel-case separation. Then, the number of lines of code was computed as a metric for each test case, as well as the number of occurrences for each java keyword present in the file. Afterwards, these metrics were placed in both a CSV file and an ARFF file, to be ready for the input of the scikit-learn and Weka classifiers respectively.

3.2 Machine Learning

Machine learning is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. Machine learning focuses on the development of computer programs that can access data and use it to learn for themselves.

Machine learning algorithms are often categorized as supervised or unsupervised.

- **Supervised machine learning** algorithms can apply what has been learned in the past to new data using labeled examples to predict future events. Starting from the analysis of a known training dataset, the learning algorithm produces an inferred function to make predictions about the output values. The system can provide targets for any new input after sufficient training. The learning algorithm can also compare its output with the correct, intended output and find errors to modify the model accordingly.
- **Unsupervised machine learning** algorithms are used when the information used to train is neither classified nor labeled. Unsupervised learning studies how systems can infer a function to describe a hidden structure from unlabeled data. The system does not figure out the right output, but it explores the data and can draw inferences from datasets to describe hidden structures from unlabeled data.
- **Semi-supervised machine learning** algorithms fall somewhere in between supervised and unsupervised learning, since they use both labeled and unlabeled data for training – typically a small amount of labeled data and a large amount of unlabeled data. The systems that use this method can considerably improve learning accuracy. Usually, semi-supervised learning is chosen when the acquired labeled data requires skilled and relevant resources to train it or learn from it. Otherwise, acquiring unlabeled data generally does not require additional resources.
- **Reinforcement machine learning** algorithms is a learning method that interacts with its environment by producing actions and discovers errors or rewards. Trial and error search and delayed reward are the most relevant characteristics of reinforcement learning. This method allows machines and software agents to automatically determine the ideal behavior within a specific context to maximize its performance. Simple reward feedback is required for the agent to learn which action is best; this is known as the reinforcement signal.

Machine learning enables analysis of massive quantities of data. While it generally delivers faster, more accurate results to identify profitable opportunities or dangerous risks, it may also require additional time and resources to train it properly.

In our use case, by using the previously obtained test cases datasets, we trained various machine learning classifiers in two environments.

First, to replicate the results obtained in [3], we used Weka [5], a java framework written in Java containing a variety of machine learning algorithms for data mining tasks. Weka contains tools for data preparation, classifications, regression, clustering, association rules mining and visualization and can be used both as a Java API and in a GUI format. The classifiers we run and evaluated in Weka are:

- Random Forest.
- Decision Tree.
- Naïve Bayes,
- K-Nearest Neighbors.

Once these first results were extracted, we decided to compare them with the ones obtained by using one of the most popular machine learning libraries in python, scikit learn. Also, one additional classifier, Support Vector Machines, was used in this phase.

3.3 Genetic algorithms and feature selection

A genetic algorithm is a search heuristic inspired by Charles Darwin's theory of natural evolution. From an algorithmic perspective, this evolution generally starts from a population of randomly generated individuals, and is an iterative process, with the population in each iteration called a generation. In each generation, the fitness of every individual in the population is evaluated; this fitness is generally the value of the objective function in the optimization problem being solved. The more fit individuals are stochastically selected from the current population, and each individual's genome is modified through a process of crossover and possibly mutation, to form a new generation of individuals- The new generation of

candidate solutions is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when a condition or a set of conditions has been met; usually this includes terminating a pre-set number of iterations or exceeding a threshold of consecutive iterations in which the optimum value does not improve.

The population size depends on the nature of the problem, but typically contains several hundreds or thousands of possible solutions. Often, the initial population is generated randomly, allowing the entire range of possible solutions (the search space). During each successive generation, a portion of the existing population is selected to breed a new generation. Individual solutions are selected through a fitness-based process, where fitter solutions (as measured by the fitness function) are typically more likely to be selected. But why isn't the fittest individual automatically selected for reproduction? Given the nature of optimization problems, we can see the entire space of possible solutions as a function; in most circumstances this function is non-monotonic, therefore it will contain many oscillations. Now, when calculating our fitness score, we are "moving" onto the curve trying to find the optimal value (aka the global maximum point on the function). While searching for this value may get "stuck" in a local maximum point and the future generations of the genetic algorithm will not be able to make any further progress.

In machine learning, feature selection is the process of reducing the number of input variables when developing a predictive model. It is desirable to reduce the number of input variables to both reduce the computational cost of modelling and, in some cases, to improve the performance of the model. Traditional approaches are statistical-based feature selection methods that involve evaluating the relationship between each input variable and the target variable using statistics and selecting those input variables that have the strongest relationship with it. Mathematically, input selection for a classification model can be formulated as a combinatorial optimization problem. The objective function is the predictive model's generalization performance, represented by some parameter on the selection instances of a dataset. An exhaustive selection of features would evaluate all 2^N different combinations, where N is the number of features. This process is obviously quite expensive from a computational standpoint and if the number of features is significant, it quickly becomes impractical. Therefore, we need intelligent methods that allow the selection of features in practical scenarios.

For our Java implementation, we decided to employ genetic algorithms to select the best features on which to fit the machine learning classifiers discussed earlier; our objective function is therefore the maximization of some parameter of the classifier, such as its accuracy. First, our dataset is composed of a total of 1067 features, excluding the target attribute for flakiness; this means that our population is made of individuals whose trait is a binary string of 1068 characters. The correspondence between this binary string and our feature list is as follows: if the n^{th} character of the string is set to 1 then the n^{th} feature will be considered among the ones fed as input to the classifier in the fitness calculation process. On the other hand, if the n^{th} character is set to 0, then we will not consider that feature later. Initially, individuals have been initialized randomly, and different percentages for the starting number of 1s and 0s in each string has been tried; later we will be presenting the results that emerged from each choice.

The selection of the individuals has been performed as a fitness proportionate selection, also known as roulette wheel selection; with this technique, the fitness level calculated for each individual is used to associate a probability of selection to each individual chromosome. While candidate solutions with a higher fitness score will be less likely to be eliminated, there is still a chance they may not be selected since their probability of selection is less than 1. If f_i is the fitness of individual i in the population, its probability of being selected is:

$$p_i = \frac{f_i}{\Sigma_{j=1}^{N} f_j}$$

Given the simple structure of our binary individuals, crossover has been implemented by selecting a cutting point for the two parent individuals and recombining their features in the offspring individuals. This approach is called single point crossover and, while the selection of the cutting point is randomized, often ensures a fair redistribution of the parents in the two offspring chromosomes. The image below highlights a simple crossover example using this technique.

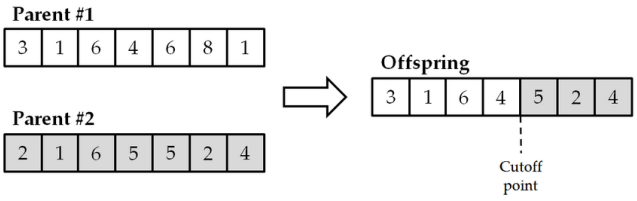

Figure1. Example of single point crossover

Finally, new individuals may be subject to mutation; in our case, if an individual is selected to be mutated according to a pre-set mutation probability, a position in its string is randomly selected and the corresponding bit is flipped.

4 PRELIMINARY RESULTS AND FINDINGS

4.1 Machine Learning

After the mentioned pre-processing steps on the dataset, we were successfully able to replicate the results obtained by [3]

in Weka. Furthermore, to assess these results and see how they held up compared to the output of the same classifiers implemented from different libraries, we decided to perform the same classification in scikit-learn. The implementation of this second approach was also overall easier to develop (except for Weka's GUI of course) compared to Weka's API.

The following table summarizes the overall measurements obtained:

Table 2: Machine learning classification results

Classifier	Acc	Pre	Rec	F_1	AUC	MCC
Weka						
Random Forest	0.95	0.91	0.99	0.95	0.99	0.90
Decision Tree	0.90	0.93	0.93	0.95	0.98	0.90
Naïve Bayes	0.83	0.79	0.90	0.84	0.92	0.68
KNN	0.92	0.88	0.96	0.92	0.92	0.85
Scikit-learn						
Random Forest	0.95	0.95	0.95	0.95	0.95	0.91
Decision Tree	0.92	0.93	0.91	0.92	0.92	0.84
Naïve Bayes	0.77	0.93	0.58	0.71	0.77	0.58
KNN	0.83	0.74	0.99	0.85	0.83	0.70
SVM	0.94	0.91	0.98	0.94	0.94	0.88

We can immediately notice how similar the two tree algorithms (Random Forest and Decision Tree) are between the two techniques. The next classifier shows us mixed results: Weka's version of Naïve Bayes has considerable better accuracy, recall (and F_1-score, consequently), AUC and MCC, while for some reason the implementation in scikit-learn has produced a higher precision. The KNN algorithm also performs quite better in the Weka implementation, with an improvement of around 10% in most measurements. Finally, the implementation of Support Vector Machine form scikit-learn can be considered on par with the higher end of the considered classifiers, however its execution time is quite a bit slower compared to them.

4.2 Genetic Algorithm

During the evaluation process of the developed genetic algorithm, different configurations of the parameters have been tried to determine the best results. These parameters include:

- Number of generations: the maximum number of iterations after which the algorithm must stop.
- Maximum number of generations with no improvement: the number of consecutive iterations with no improvement of the optimum after which the algorithm must stop.
- Number of individuals making up each generation.
- Probability of crossover.
- Probability of mutation.
- Number of initial features considered (1s in the individual's string).
- Classifier for the fitness score: the machine learning model used for the evaluation of each feature list.

The main downside of this algorithm is its execution speed when the classifier performs the evaluation: choosing a slower machine learning model can severely impact on the time it takes to evaluate a full generation. For this reason, we mainly focus on applying this technique on the Naïve Bayes classifier, which represents a good compromise between execution time and prediction accuracy. The next table highlights the results obtained after running the algorithms for 100 generations, with 100 individuals, a maximum number of generations without improvements set to 20, a crossover probability of 0.5 and a mutation probability of 0.3.

Table 3: Results of the application of the genetic algorithms with the Naïve Bayes classifier

%1s	Stop gen.	Acc	Pre	Rec	F_1	AUC	MCC
40	21	0,862	0.82	0.92	0.87	0.92	0.73
50	**32**	**0,875**	**0.84**	**0.92**	**0.88**	**0.92**	**0.75**
60	37	0,869	085	0.89	0.87	0.93	0.74
70	36	0.855	0.81	0.92	0.86	0.92	0.71
80	50	0.862	0.82	0.91	0.86	0.92	0.73

Perhaps surprisingly, initializing the individuals with only 50% of the total features produced the best results, while also resulting in a quicker fitness score calculation.

When comparing the information highlighted in table 2 and 3 we notice a nice improvement over the performance of the

Naïve Bayes classifier, especially considering the relatively low number of individuals and generations used. The results may differ when employing a more complex classifier such as Random Forest, for which the evaluation process takes significantly longer, and for which we did not bother to run the algorithm extensively.

5 IMPLICATIONS OF THE RESULTS

From a research standpoint, the various solutions to detect and treat test flakiness can still be explored; in particular, the performance of a neural network can be compared to that of the traditional classifiers we focused on. Besides this, the application of genetic algorithms for feature selection can easily be extended to other classifiers and problems since it is very adaptable by nature.

6 CONCLUSIONS

Flaky test cases can seriously impact on the development process of a software system, especially with today's heavier focus on practices such as Continuous Integration and Delivery. Factors such as misuse of concurrency, test order dependency violations or resource leakage are the main factors that cause test flakiness. The results obtained with the employed predictors in our study tell us that machine learning can be an effective method for dealing with test flakiness prediction, provided we are equipped with a proper test suite to mine. Furthermore, what we can take from the obtained results with our genetic algorithm is that using this approach for feature selection can bring a noticeable improvement on the performances of a classifier, even when traditional statistics-based approaches, such as information gain, are employed. Ultimately, considering this approach in a wider scope, that transcends from the detection of flaky tests, and valuing whether it is worth it or not to apply genetic algorithms in addition to traditional machine learning techniques, we can say that it vastly depends on both the problem's performance requirements and the time constraints at disposal to the developers. If gaining an edge is the absolute priority when developing a predictor, then using genetic algorithms may serve the purpose.

One potential downside to this whole process can be expressed as the following question: what would change by considering test cases written in another programming language? Obviously, depending on the language, things such as semantics, conventions, keywords and size of test cases might change, and as a result, new tokens associated with flakiness may arise.

7 REFERENCES

[1] Spotify article on flaky tests: https://engineering.atspotify.com/2019/11/18/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/

[2] "DeFlaker: Automatically Detecting Flaky Test Cases", Jonathan Bell et al., 2018.

[3] "What is the Vocabulary of Flaky Test?", Gustavo Pinto et al., IEEE/ACM 17th International Conference on Mining Software Repositories (MSR), 2020.

[4] DeFlaker benchmark: http://www.deflaker.org/icsecomp/

[5] Weka: https://www.cs.waikato.ac.nz/ml/weka/