February 14, 2019

INTELLIGENT REGRESSION TESTING FOR INTERNET OF THINGS WIRELESS DEVICE USING MIXED MACHINE LEARNING METHODS

Lele Zhang

Mingyu Xie

Chuanwei Li

Xiang Fang

INTELLIGENT REGRESSION TESTING FOR INTERNET OF THINGS WIRELESS DEVICE USING MIXED MACHINE LEARNING METHODS

AUTHORS:
Lele Zhang
Mingyu Xie
Chuanwei Li
Xiang Fang

ABSTRACT

Techniques are described herein for a mixed Machine Learning (ML) method which contains both the Ant Colony Optimization (ACO) algorithm and Bayesian method. The regression test is very common for a Quality Assurance (QA) team in its daily work, but usually has too many cases for a very little-changed patch. The techniques described herein may pick up as few desired test cases as possible but still guarantee the same testing performance as before. This may improve the efficiency of the QA team, shorten its work time, and reduce the requirement for testing devices as well.

DETAILED DESCRIPTION

Mesh networks may provide wireless mesh connectivity for Industrial Internet of Things (IIoT) customers, which is widely used with Advantaged Metering Infrastructure (AMI) and Distributed Automation (DA) devices. Currently, mesh networks may support millions of nodes to form a tree-based topology network according to Internet Engineering Task Force (IETF) Request for Comments (RFC) 6550. In order to test such a wireless Low-power and Lossy Network (LLN), the Quality Assurance (QA) team divides it into several blocks, and each block contains one or more required test cases. Example test cases are listed below:

- Personal Area Network (PAN) Join to test all steps foe a node joining a PAN based on the Media Access Control (MAC) layer.
- Frequency Hopping to test a frequency hopping schedule for nodes according to a given channel function.
- Internet Protocol version 6 (IPv6) Neighbor Discovery (ND) to test neighbor discovery behavior in IPv6.

Published by Technical Disclosure Commons, 2019

- Dynamic Host Control Protocol version 6 (DHCPv6) to test DHCPv6 behavior for nodes to check whether a node may obtain a global address.
- Traffic to test the percentage of packets lost for both unicast and broadcast on the link.
- Multicast Protocol for LLN (MPL) to test multicast behavior for nodes.
- Routing Protocol for LLN (RPL) to test routing behavior for nodes.
- Error Handling to test how a node treats an unknown error, to ensure the robustness of mesh network devices.
- Security to perform security related tests for nodes, such as Dot1x and Dot11i.

Based on the normal work process, the development team always updates an internal official software version periodically (e.g., one or two weeks), depending on the quantity of patches added since the last release. No matter how few changes the new version has, the QA team needs to run all regression test cases with the new software release. This kind of regression test may take too much time but get very poor efforts for various reasons.

One reason is that regression tests usually have quite a lot of test cases to avoid unexpected bugs imported with new patches, but most of them are not required in practice. Figure 1 below illustrates the Pareto principle, according to which only 20% of cases are necessary and 80% of efforts are redundant.

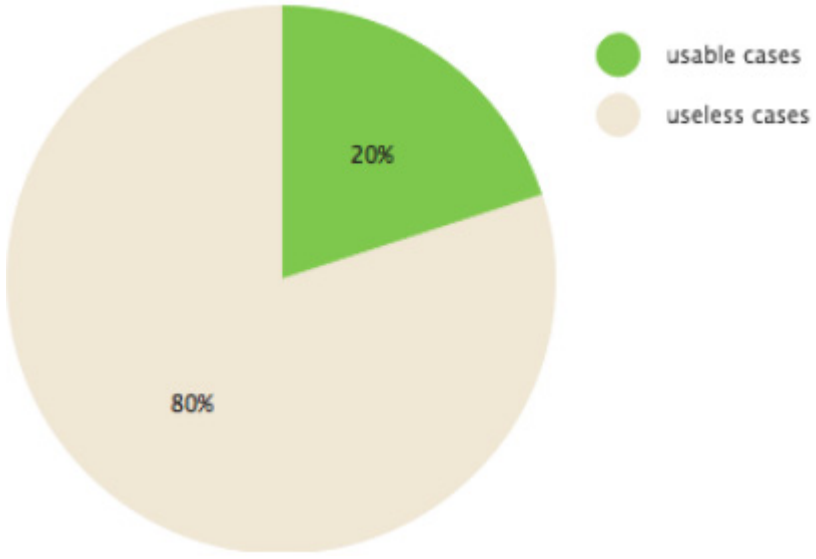

Figure 1

Another reason is that too many test cases consume limited resources, such as time, QA engineers, and test beds. There is a large penalty for undesired and useless cases. A third reason is that sometimes, QA engineers may pick up some required cases with their understanding to shorten resource consumption. However, this is not reliable because it quite relies on the experience and knowledge for QA engineers. As such, a general solution is needed without human interference.

For an example, if a new patch just modifies the behavior of beacon interactions among nodes, the test cases of security and Layer 3 (L3) (e.g., MPL, RPL, DHCPv6, etc.) may be meaningless. But the regression test is fixed and executed automatically, and is required to run all test cases even though most are not very related to the new software release.

Nowadays, the development of Machine Learning (ML) technology has provided theoretical foundations for solving this problem, and some methods of ML have promising ideas for selecting the right test cases for each change. However, the techniques described herein make use of mixed ML methods to solve this problem.

In order to describe this idea clearly, some sets are defined as below:

1. The regression test plan has n test cases, and they are placed into a set named Z. The formula may be represented as Z = {TC(0), TC(1), ..., TC(n-1)}, where TC(i) represents the test case of number i.
2. A set F is defined to contain all of the code files. Furthermore, the mesh network uses a C-like language called NesC language for development, which includes many *.h and *.c files. So, if there are m files in total, the set F may be represented as F = {f(0), f(1), ..., f(m-1)}, where f(j) represents the file of number j.
3. There are k testing patches to train data. These patches include a set Y = {y(0), y(1), ..., y(k-1)}.

As described herein, parts of ACO concepts may be used for training data. As illustrated in Figure 2 below, the ACO algorithm is based on the solution for finding the shortest path by biological ants. When they are looking for food, they leave pheromones on the path. If other ants detect the pheromones, they will prefer to follow this path with a higher probability. The more ants go through this path, the more pheromones remain. This result encourages more ants to select this path again.

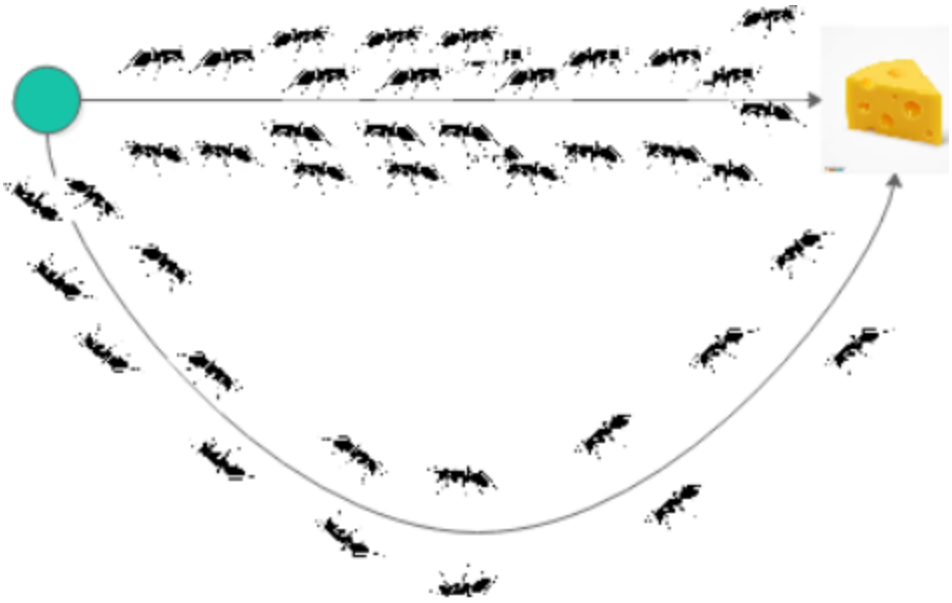

Figure 2

Based on the above information, a patch may be treated as an ant that wants to select as few cases as possible for completing its regression test. In order to leverage the ACO algorithm, some concepts from the ACO algorithm must be imported, as illustrated in Figure 3 below.

Parameter Name	Description
p	this element indicates the probability of being picked up for one test case, it is the same at the beginning. So, all test cases will be executed with the same probability on the first round.
p_{init}	the initial probability of being chosen for each test case, such as 70%, 80% or even 100%
$p_{threshold}$	the lowest probability of being chosen for each test case, like 25% or 20%. Because after several rounds of training, the probability of some cases will be closed to zero, so that they will never be picked up even it is required in practice. So, for overcoming this over-fitting problem, we define this parameter here.
ph	this element represents the density of pheromone, if one test case is hit (i.e., testing patch gets failure with the test case), this parameter will be added in the next round. We recommend that the scope of ph is between $0.5*p_{init}$ and $1.5*p_{init}$.
vol	volatile factor, which would reduce the probability per iteration. This parameter takes in charge of adjusting the speed of dropping probability of test cases.

Figure 3

At the beginning, each element in set Z has the same probability of being picked up by the testing patch. Therefore, a matrix Q is created for set Z accordingly.

$$Q[n] = [p_0, p_1, \cdots, p_{n-1}] \qquad (p_0 = p_1 = \cdots = p_{n-1} = p_{init})$$

Because the mesh network has m files in total, a matrix file is created for set F as shown below.

$$File[m] = [1, 1, 1, 1, 1, \ldots, 1]^T$$

Two matrices are multiplied together, and then a new matrix P is obtained.

$$P[n][m] = Q[n] * File[m] = \underbrace{\begin{bmatrix} p_{init} & \cdots & p_{init} \\ \vdots & \ddots & \vdots \\ p_{init} & \cdots & p_{init} \end{bmatrix}}_{n} \Big\} m$$

Thus, patch y(0) is tested for the first round. It contains three updated files, which are represented as a set F' = {f(3), f(5), f(8)}. These three probability tables are obtained from matrix P' with the following formula.

$$P'[n] = \frac{P[n][3]+P[n][5]+P[n][8]}{3} = [p_{init}, p_{init}, \cdots, p_{init}]$$

All the test cases may be picked up with the same probability (p_{init}). For instance, if p_{init} is 80%, each test case may be triggered with 80% probability when testing patch y(0). If y(0) is failed with TC(i), the probability may be updated by adding the ph according to the following formula. Otherwise, ph[i] is zero.

$$P''[i] = \frac{P'[i]+ph[i]-vol}{\frac{1}{n}\sum_{i=0}^{n-1}(P'[i]+ph[i]-vol)} * \frac{1}{n}\sum_{i=0}^{n-1} P'[i]$$

$$p'_i = \begin{cases} p_{threshold}, & P''[i] < p_{threshold} \\ P''[i], & 100\% > P''[i] \geq p_{threshold} \\ 100\%, & P''[i] \geq 100\% \end{cases}$$

Based on the above, a new matrix may be obtained after the first round as p'.

$$p'[n] = [p'_0, p'_1, \cdots, p'_{n-1}]$$

P[n][m] is updated with p'[n] before the second round is started.

$$P[n][m] = \begin{bmatrix} p_{init} & p_{init} & \cdots & p_{init} \\ p_{init} & p_{init} & \cdots & p_{init} \\ p_{init} & p_{init} & \cdots & p_{init} \\ p'_0 & p'_1 & \cdots & p'_{n-1} \\ p_{init} & p_{init} & \cdots & p_{init} \\ p'_0 & p'_1 & \cdots & p'_{n-1} \\ \vdots & \vdots & & \vdots \\ p_{init} & p_{init} & \cdots & p_{init} \end{bmatrix}$$

The same is performed for the remaining patches, such as y(1), y(2), ..., y(k-1), and the matrix P[n][m] is updated per round per patch. After multiple rounds of training, the machine may learn how to generate a thin but high efficient regression test list automatically.

In one example, a regression test has 50 cases, so n = 50. It is assumed that there are 100 files of source codes, and so m = 100, F = {f(0), f(1), ..., f(99)}. It is also assumed that there are two patches, so Y = {y(0), y(1)}. y(0) contains 3 files, and F'(y(0)) = {f(1), f(2), f(4)}. y(1) contains 2 files, and F'(y(1)) = {f(4), f(5)}. It is assumed that p_{init} is 80%, $p_{threshold}$ is 25%, ph = 40%, and vol = 1%.

In the first round, all test cases have the same probability (80%) to be picked up by y(0). It is assumed that y(0) has failed with cases 1, 2, and 3. The probability matrix is updated based on the above result, so the P'[1] = P'[2] = P'[4] = [p', p'', p'', p', ..., p']. p' = 77.6%, and p'' = 116.9%. Because p'' is larger than 100%, p'' = 100%.

$$p' = \frac{0.8-0.01}{\frac{1}{50}(50*(0.8-0.01)+0.4*3)} * \frac{1}{50} * (50 * 0.8) = 77.6\%$$

$$p'' = \frac{0.8+0.4-0.01}{\frac{1}{50}(50*(0.8-0.01)+0.4*3)} * \frac{1}{50} * (50 * 0.8) = 116.9\%$$

Finally, P[n][m] is updated.

$$P[50][100] = \begin{bmatrix} 0.8\ 0.8\ 0.8\ 0.8\ \dots\ 0.8 \\ 0.776\ 1\ 1\ 1\ \dots\ 0.776 \\ 0.776\ 1\ 1\ 1\ \dots\ 0.776 \\ 0.776\ 1\ 1\ 1\ \dots\ 0.776 \\ 0.8\ 0.8\ 0.8\ 0.8\ \dots\ 0.8 \\ 0.8\ 0.8\ 0.8\ 0.8\ \dots\ 0.8 \\ \vdots \\ 0.8\ 0.8\ 0.8\ 0.8\ \dots\ 0.8 \end{bmatrix}$$

In the second round, y(1) picks up test cases according to P[n][m] from the first round, so the new P'[n] is detailed as follows. It is assumed that that y(1) has failed with case 2 and 3.

$$P'[n] = [0.788, 0.9, 0.9, 0.9, \dots, 0.788]$$

The probability matrix is updated based on above result, so P'[4] = P'[5] = [p', p'', p''', p''', p', ..., p']. p' = 77.1%, p'' = 89.1%, p''' = 128.8%. Because p''' is larger than 100%, p''' = 100%.

$$p' = \frac{0.788-0.01}{\frac{1}{50}(47*(0.788-0.01)+3*(0.9-0.01)+0.4*2)} * \frac{1}{50} * (47 * 0.788 + 3 * 0.9) = 77.1\%$$

$$p'' = \frac{0.9-0.01}{\frac{1}{50}(47*(0.788-0.01)+3*(0.9-0.01)+0.4*2)} * \frac{1}{50} * (47 * 0.788 + 3 * 0.9) = 89.1\%$$

$$p''' = \frac{0.9+0.4-0.01}{\frac{1}{50}(47*(0.788-0.01)+3*(0.9-0.01)+0.4*2)} * \frac{1}{50} * (47 * 0.788 + 3 * 0.9) = 128.8\%$$

Finally P[n][m] is updated.

$$P[50][100] = \begin{bmatrix} 0.8\ 0.8\ 0.8\ 0.8\ \dots\ 0.8 \\ 0.776\ 1\ 1\ 1\ \dots\ 0.776 \\ 0.771\ 0.891\ 1\ 1\ \dots\ 0.776 \\ 0.771\ 0.891\ 1\ 1\ \dots\ 0.776 \\ 0.8\ 0.8\ 0.8\ 0.8\ \dots\ 0.8 \\ 0.8\ 0.8\ 0.8\ 0.8\ \dots\ 0.8 \\ \vdots \\ 0.8\ 0.8\ 0.8\ 0.8\ \dots\ 0.8 \end{bmatrix}$$

The more rounds this algorithm runs, the more useless cases are dropped, as shown in Figure 4 below. The parameter "vol" may adjust the speed of convergence.

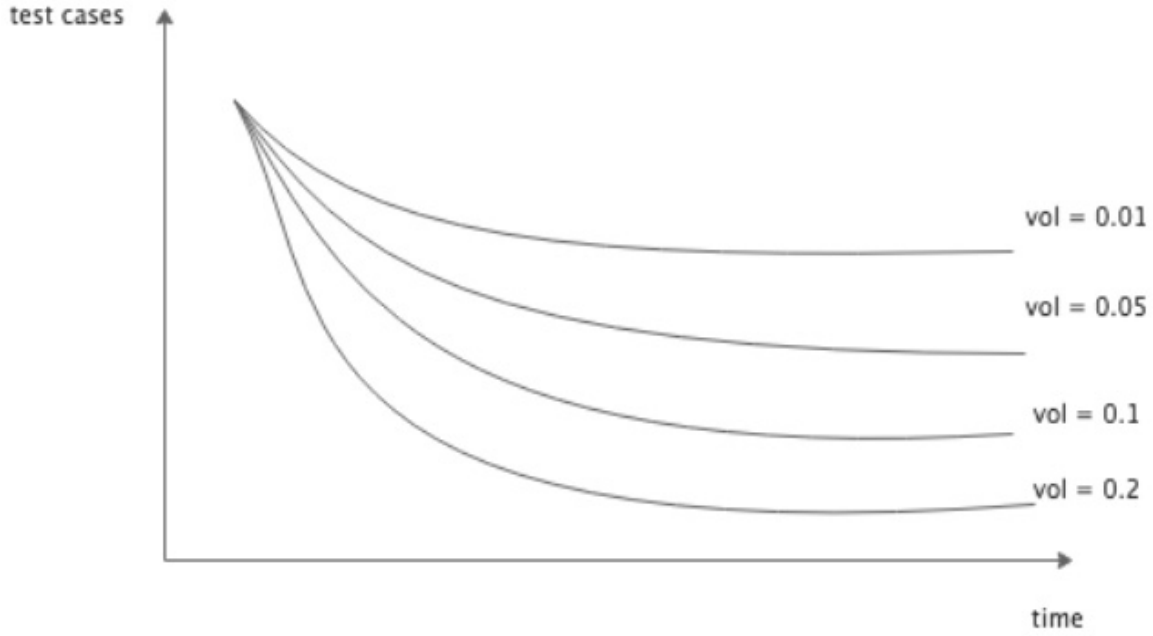

Figure 4

Bayesian methods may also be used for complementing required test cases which are missed while using the improved ACO algorithm. Sometimes, the result of the required test cases generated by the improved ACO algorithm are not used enough. In one example, one patch has updated some codes about beacon behavior, and the improved ACO

algorithm gives the result is just to test PAN Join cases. But actually, some cases of Frequency Hopping are required as well. Although the improved ACO algorithm provides the parameter "pthreshold" to guarantee randomness of selection, some key test cases will still be missed with high probability. So, in order to solve this problem, a Bayesian method may be used.

Bayesian discloses the potential relationship between two events (e.g., event B will occur with high probability if event A occurs), even if they do not seem to be connected. It is very useful for this solution. If the improved ACO algorithm just picks up test case group A, but group B is required as well in practice, the Bayesian method may know that and recommend the system to add group B into the regression test list.

Matrix P[n][m] may be used as input for the Bayesian method. Matrix P[n][m] indicates the probability of being picked up between test case and code file, but does not disclose the relationship among files because the probability of each file is totally independent. The Bayesian method may be used to find out the relationship between a pair of files, and then adjust the probability value of matrix P[n][m].

Figure 5 below illustrates an example process flow.

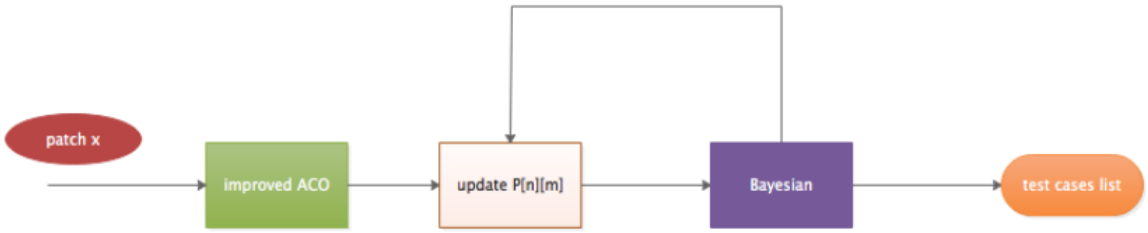

Figure 5

In summary, techniques are described herein for a mixed ML method which contains both the ACO algorithm and Bayesian method. The regression test is very common for a QA team in its daily work, but usually has too many cases for a very little-changed patch. The techniques described herein may pick up as few desired test cases as possible but still guarantee the same testing performance as before. This may improve the efficiency of the QA team, shorten its work time, and reduce the requirement for testing devices as well.