# Automated UI Test Case Prioritization using TERMINATOR

**Parvez Rashid**
mrashid4@ncsu.edu
North Carolina State University

## ABSTRACT

Testing is an essential part of continuous integration process in Software Engineering. One curse of modern UI testing is to run too many test cases and not to mention in web based UI, components are dependent on each other. As a result it takes a long time to test in every integration. For example, the entire automated UI testing suite at LexisNexis takes around 30 hours (3-5 hours on the cloud) to execute, which slows down the continuous integration process. A solution to this problem is test case prioritization. But as the UI components has dependencies, it is very difficult to test every components in each integration. In modern approaches we have to have access to the code for finding the relation between the code and test cases and that is a problem as it is not always we have access to the code-base.In the era of automation, automated test case prioritization (TCP) can be an important step to solve this problem. Most of the automated UI testing is "black box" in nature, most cases the test case description and result are available. Terminator, is an novel TCP approach by Zhe Yu and el al. that dynamically re-prioritizes the test cases when new failures are detected, by applying and adapting a state of the art framework from the total recall problem. This paper is an initial implementation of Terminator and a secondary approach to identify improvement.

## CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Information systems** → *Learning to rank*.

## KEYWORDS

datasets, neural networks, decision tree, text tagging

## 1 INTRODUCTION

If test ran infinitely quickly we would run all test all the time. But it is not the reality, so we have balance between cost and value. In the test pyramid structure we run unit tests more frequently because they usually run faster, are less brittle and give more specific feedback. In particular, we frame a suite of tests that should be run as part of continuous integration[3]. Complex web applications must be validated using variety of methods. Early in the life cycle we unit and integration tests are used to validate many component of the application. but A modern web application is an amalgam of reports from many micro-services contributed by many different developers or team. For this reason unit test is not a good choice for complex web applications. We use automated UI test. Compared to unit tests, automated UI tests are more expensive to write and maintain. Also, they usually have a higher execution time. For example, the automated UI test suite at LexisNexis takes around 30 hours (3-5 hours on the cloud) to execute. The issues with automated UI test is that it is very time and resource consuming. As a result we can not do it often. Also it a long time for giving feedbacks to the developers. Developers are less agile when testing forces them to wait for feedback. As mentioned earlier modern web applications are relied one reports from different micro-services, it is difficult to make a straightforward connection between them. A simple query can be connected with results from dozens of under the hood other applications developed from different teams. This complex relationship makes it near to impossible to find a reason of any failed test cases. Furthermore much of the testing is not with detail information or we can say "black box" in nature as the source code information is not available were there is no mapping for test cases and source code. It is not possible to draw a conclusion of responsible code for test failure.
Understanding the problem associated with automated UI testing to make it faster and effective one approach is to use test case prioritization (TCP). TCP is a widely studied topic in software engineering (SE). In TCP test cases are prioritized so that they can reveal the fault earlier as running the test is expensive [17,25,34,35,39]. In traditional TCP approaches test coverage or Test Impact Analysis (TIA) is used. The key idea of TIA is to make a mapping of test cases to source codes to identify the coverage information (Figure 1 and 2). Unfortunately, such techniques are not applicable to automated UI testing since the coverage information is not available.
The Terminator paper [13] explores TCP techniques for higher failure detection rates utilizing only “black box” information; i.e. test descriptions and results from their own prior (and current) test runs (time to run the test, whether or not that test failed). The core idea of this work is that this particular type of problem belongs to a class of information retrieval problems called the total recall problem[11]. To support the idea Zhe Yu and et al. applied total recall to read

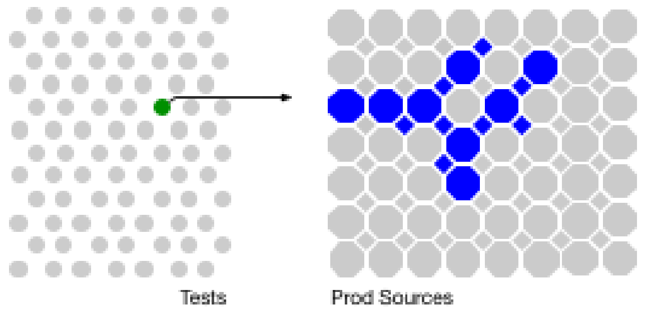


**Figure 1: Maping test case to source code**

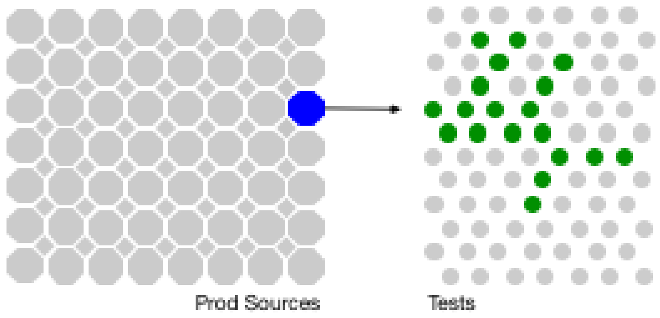


**Figure 2: Maping source code to test cases**

the description and history results of the test cases to build a classifier using support vector machine (SVM) to predict which test might fail earlier and described the problem as an active learning based [9] framework. In the continuation of their work My experiment was to find if it shows an improvement using Random Forest (RF) instead of SVM. In this work Terminator was compared with 12 state of the art TCP algorithms and 2 baseline algorithms on 54 consecutive runs of automated UI testing (data from three months of testing at LexisNexis) to answer the following two research questions:

- **RQ1: can the proposed approach TERMINATOR achieve significantly higher failure detection rates than other TCP algorithms when prioritizing automated UI tests from LexisNexis?** Results shows that TERMINATOR significantly outperformed other TCP algorithms in terms of failure detection rates. This suggests that techniques from the total recall problem can be successfully applied and adapted to address the automated UI test case prioritization problem.
- **RQ2: what is the computational overhead of TERMINATOR?** Results showes that TERMINATOR had 50% more computational overhead than the simple history-based TCP algorithms because it recursively updates the SVM model and dynamically adjusts the order of the unexecuted tests. However, TERMINATOR's overhead was still negligible compared to the runtime of the test cases (0.33% of the runtime of the test suite).

The basic idea of this project for is to understand the idea that Zhe Yu and et al. applied for test case prioritization and trying to improve the result with a different classification method.

## 2 BACKGROUND AND RELATED WORK

Terminator is a novel approach for test case prioritization for automated UI testing. This work was particularly developed to solve the testing problem of web UI of LexisNexis that provides regulatory, legal and business information and analytics to the global community[6].

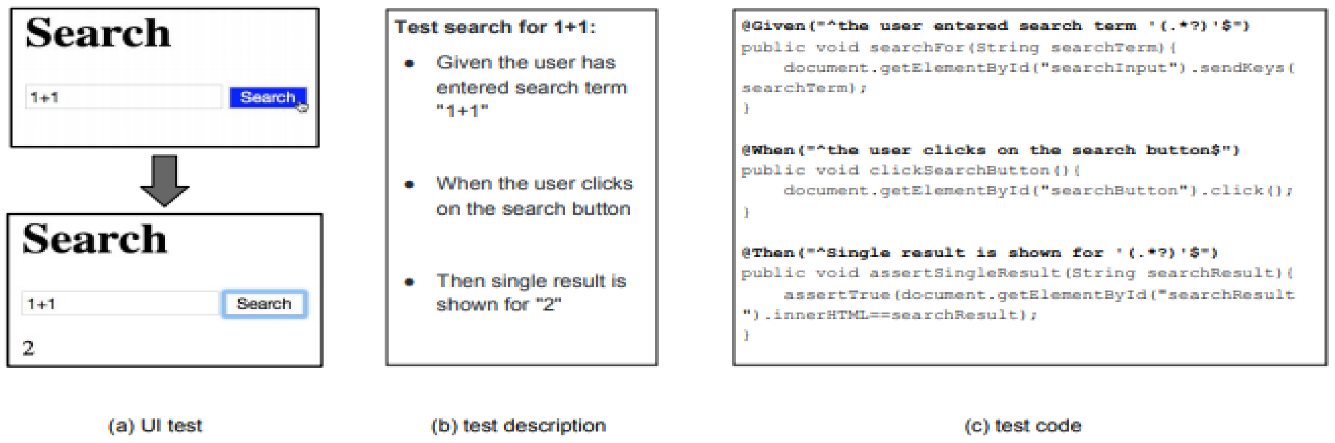
### Automated UI testing

Automated UI testing is an important part of continuous integration in SE. For each integration UI components like check box, search box or buttons get tested to see if they are working properly with recent updates or changes in the system. A sample test example is shown in the figure 3 where if a "1+1" is given as input and hit the search button, a result "2" will show up. The test code and the test cases are shown in the figure 3.

### Test Case Prioritization

Test case prioritization is a way to schedule test cases in an order which can give a better performance for testing[1]. Better performance can be varied according to different goals. Some goals of different TCP approaches are described bellow,

- **Coverage:** Some of the TCP approach focuses on the most coverage. The term "coverage" can vary from requirement[10], statement coverage, decision coverage [14],block coverage[14], branch coverage [1] etc.
- **Fault detection rate:** Among all the approaches for test case prioritization finding maximum number of faults is the most popular one. This is because it can make the application more accurate and developers can get an feedback for fixing those faults. A metric is used to evaluate the fault coverage proposed by Rothermel et al. [1] called APFD. APFD computes the area under the curve (AUC). In APFD each test case is considered to be of the same cost and the same severity. In practice different failure can have different severity. Considering the severity Elbum et al[8] has proposed a metric with average percentage of fault detected with cost (APFDc), which takes into consideration the cost related to the resources required to execute and validate each test case and the severity of each fault. Let T be a test suite containing n test cases with costs



Test search for 1+1:

- Given the user has entered search term "1+1"
- When the user clicks on the search button
- Then single result is shown for "2"

```
@Given("^the user entered search term '(.*?)'$")
public void searchFor(String searchTerm){
    document.getElementById("searchInput").sendKeys(
searchTerm);
}

@When("^the user clicks on the search button$")
public void clickSearchButton(){
    document.getElementById("searchButton").click();
}

@Then("^Single result is shown for '(.*?)'$")
public void assertSingleResult(String searchResult){
    assertTrue(document.getElementById("searchResult
").innerHTML==searchResult);
}
```

**Figure 3: To test a page shown at the left (a), programmers write a test description (b) which is converted to test code (c).[13]**

t1,t2, . . . ,tn. Let F be a set of m faults revealed by the test suite and let f1, f2, . . . , fm be the severity of those faults. Let TFi be the first test case that reveals fault i. APFDc is calculated as follows:

$$APFDc = \frac{\sum_{i=1}^{m}(fi \times (\sum_{i=TFi}^{n} ti - 0.5t_{TFi}))}{\sum_{i=1}^{m} ti \times \sum_{i=1}^{m} fi} \qquad (1)$$

- **Failure detection rate:** : Some studies prioritize test cases to achieve higher failure detection rates. In some cases failure to fault mapping information for test cases are not available, to overcome that Liang et al. [4] apply APFDc to measure failure detection rates. Here it becomes average percentage of failures detected with the same equation as (1), except that TFi represents ith failed test case.
- **Coverage Information:** TCP approaches mostly rely on coverage information from source code [1] which requires access to the code. In our case this information is not available.
- **History information:** This approach uses history information of failure or fault to prioritize test cases for current execution.
- **Cost information:** in this kind of approach TCP algorithms take into account the cost for test case execution. It uses the execution history to estimate the cost.
- **Test description information:** In this kind of approach standard natural language processing methods are used to extract information from test description as shown in figure 2(b). In Terminator Zhe Yu and et al. used this method to gather information from the test description by LexisNexis.
- **Feedback information:** Some algorithms dynamically adjust the priority of the unexicuted test cases depending on the result of the current running test cases[12]. The priorities of unexecuted test cases are adjusted dynamically based on two heuristics: (1) if one test case fails, increase the priority of unexecuted test cases that are similar/related to the failed one; (2) if one test case passes, decrease the priority of unexecuted test cases that are similar/related to the passed one.

## 3 METHODOLOGY

The automated test case prioritization problem can be generalized as total recall problem [11]. Total recall is based on active learning. We can say that active learning is the key of the approach used in Terminator.

### Total Recall

The total recall problem in information retrieval aims to optimize the cost for achieving very high recall-as close as practicable to 100%-with a human assessor in the loop[11]. More specifically the total recall problem ca be stated as following:
Given a set of candidate examples E, in which only a small fraction $R \subset E$ are positive, each example x can be inspected to reveal its label as positive ($x \in R$) or negative ($x \notin R$) at a cost. Starting with the labeled set $L = \emptyset$, the task is to inspect and label as few examples as possible ($min|L|$) while achieving very high recall $|L \cap R|/|R|$.

### Active learning:

The key idea behind active learning is that a machine learning algorithm can train faster (i.e. using less data) if it is allowed to choose the data from which it learns[9]. Active learning provided a good result to solve total recall problem. It outperformed supervised and semi supervised learners and reached a high recall.

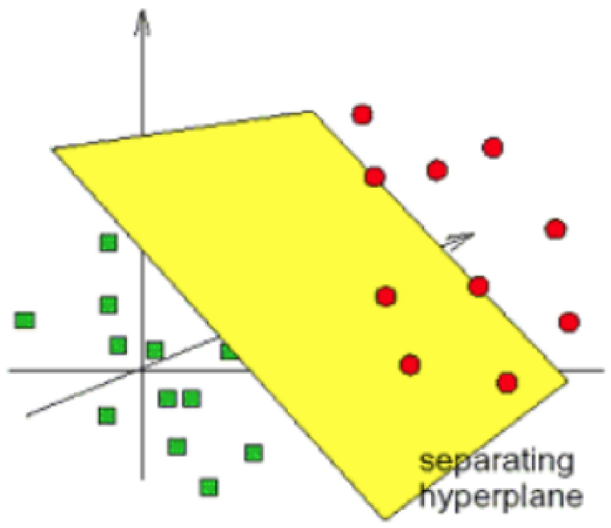


**Figure 4: Separating failing (red circles) from passing (green squares) test cases.**

In figure 4 a decision plane is shown to explain active learning. If we want to find more fail cases and we have access to the figure 3 model. We can way is to execute the test cases that fall into red circle region as far as from the green ones. This is called certainty sampling. Another way is to execute test cases that are on the boundary. This is called uncertainty sampling.

### TERMINATOR

The automated UI test case prioritization problem can be generalized to the total recall problem as follows:

- E: the test suite to be prioritized.
- R: the set of test cases that will fail if executed.
- L: the set of test cases already executed in the current run.
- LR = $L \cap R$: the set of failed test cases in the current run.

## 4 EXPERIMENT

An empirical study was presented by the Terminator paper to motivate the comparison and answer RQ1 and RQ2.

### Dataset

Test case prioritization using Terminator is an research work funded and inspired by LexisNexis. The dataset was provided by LexisNexis for 54 consecutive runs of 2661 automated UI test cases from September 27th to November 15th 2018. The collected data contains the test description written using Gherkin syntax (as shown in Figure 2 (b)), test duration, outcome (passed/failed), and error message associated with each test case. Some of the test results contains "time out" and are not considered failures.

**Algorithm 1:** Pseudo Code for TERMINATOR

```
Input: E, the test suite
R, test cases that will fail
$N_1$, batch size
$N_2$, threshold of query strategy
Result: L, list of executed test cases,
$L_R$ list of failed test cases
$L \leftarrow \emptyset$;
$L_R \leftarrow \emptyset$;
while $|L| < |E|$ do
    if $|L_R| >= 1$ then
        $L_{pre} \leftarrow Presume(L, E \backslash L)$;
        $CL \leftarrow Train(L_{Pre})$;
        $X \leftarrow Query(CL, E \backslash L, L_R)$
    else
        $X \leftarrow Random(E \backslash L)$;;
    end
    foreach $x \in X$ do
        $L_R, L \leftarrow Execute(x, R, L_R, L)$
    end
end
return $L, L_R$
Function $Presume\ (L, E \backslash L)$
    return $L \cup Random(E \backslash L, |L|)$;
Function $Train(L_{Pre})$
    $CL \leftarrow SVM(L_{Pre}, Kernel = linear)$;
    if $L_R >= N_2$ then
        $L_I \leftarrow L_{Pre} \backslash L_R$
        $CL \leftarrow SVM(L_R \cup tmp, Kernel = linear)$;
    else

    end
    return $CL$
```

| | | Test Session | | | |
|---|---|---|---|---|---|
| Test | Description | 1 | 2 | 3 | 4 |
| t1 | Test Check Box in page A | P | F | S | F |
| t2 | Test Radio Button in page A | F | F | P | F |
| t3 | Test Check Box in page B | P | P | F | P |
| t4 | Test Radio Button in page B | F | P | F | F |

**Figure 5: A simple example of automated UI test case prioritization dataset, where P, F, and S indicate passed, failed, and skipped testing results. For each test session, the runtime is t1<t2<t3<t4.**

### Comparison

The collected dataset is used to simulate the performance of different TCP algorithms. simulate Run 6 to Run 54 by using different prioritization approaches and compare their performance . When prioritizing for Run n, the execution results from Run 1 to Run n-1 are available as information for the prioritization algorithm. Figure 5 shows a small example of the dataset with 4 test cases and 4 consecutive runs. To prioritize for test cases in Run 4, all testing results from Run 1 to Run 3 and the description of each test case can be utilized. In Terminator paper three variants of active learning approach is compared with 12 existing blackbox TCP technique and 2 baseline. A short description is given bellow:

*Group A: Baselines.* Group A shows the baseline for comparison.

- **A1** executes testcases in random order. A1 is the lower limit for a algorithm to compare with.
- **A2** is the upperlimit for an algorithm to compare with. A2 uses the prior knowledge to run test cases in order.

*Group B: History based algorithm.* Group B algorithms use metrics extracted from the execution history to order the test cases in each run.

- **B1:**Applies higher priority to the most recent failed test cases and reorder testcases for the current run.
- **B2:**considers the number of time a test case failed previously. The more it failed gets higher priority in he current run.
- **B3:** uses an exponential decay metrics[5]. For calculating the metrics a value 0 is assigned to a variable if the test case was skipped or passed in previous run and 1 is assigned if the test case was failed in the previous run.
- **B4:** ROCKET metrics [2] is used for this approach where, test case get higher value if it failed in previous run and lower value if it passed. Higher priority is assigned to higher B4 metrics.

*Group C:.* Group c considers cost (usually runtime) of running for test cases.

*Group D:.* uses test case description from previous runs. If two test cases have similar descriptions they have high chance to have same outcome.

- **D1:** uses simple history from previous runs. Assign 0 to the passed cases and 1 to the failed cases.
- **D2:** it uses the entire run history rather than just the previous one.
- **D3:** Uses a weighted average to all previous runs placing importance on recent runs.

*Group E:.* Algorithms from this group uses feedback from previous runs and finds a coo-relation between test cases to make a dynamic ordering for current run..

- **E1:** Looks for coo-relation between test cases with similar failing history
- **E2:** analyzes the execution history of the test suite and built a correlation matrix of test cases. Two test cases are correlated when their results are changed to the opposite status by one commit in two consecutive test sessions (flipped together). The correlation matrix is composed of values reflecting the accumulated number of flipped results
- **E3:**By mining fail and pass rules over 90% confidence with the Repeated Incremental Pruning to Produce Error Reduction (RIPPER) algorithm, and dynamically prioritizes the unexecuted test cases. If a test case fails, its corresponding fail rule test cases will be executed next; if a test case passes, its corresponding pass rule test cases will be pushed back to the end of the execution queue. The initial order of test cases is calculated by their failure rate (B2) and the number of rules associated.

*Group F:.* Group F uses Terminator approach and its three variation depending on what kind of data it uses. Terminator starts with a confident sampling and as soon as it has a single positive data point it uses it to active learning and gathers more data point for uncertain sampling.

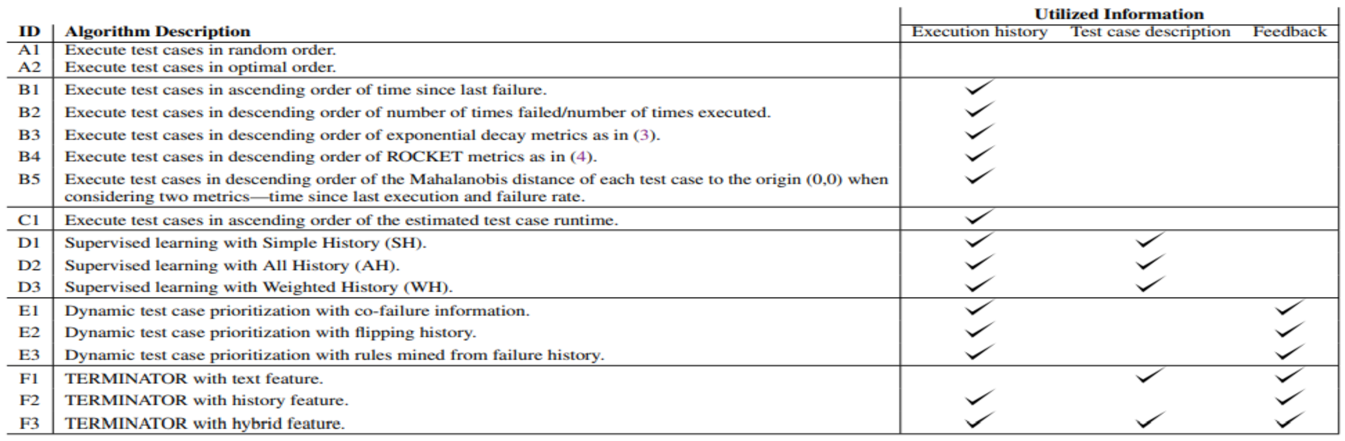
## 5 COMPARING FACTORS

For measuring performance two performance metrics were used.

- **APFDc:** average percentage of failure detected with cost gives the area under curve (AUC). We calculate it with equation 1 described earlier.
- **Overhead:** Overhead is calculated using computational time of the algorithm / total runtime of all test cases. With APFDc we only have the metrics of failure test cases but we also need to consider the running time.

## 6 RESULTS

To test the significance of any improvements Scott-Knott analysis is applied to cluster and rank the performance metrics of each algorithm from run 6 to 54. It clusters algorithm that has a little or no difference and ranks each cluster with median performances[7]. Figure 6 shows the results of the 17 algorithms discussed earlier. Using Scott-Knott they are clustered according to their medians and iqrs. Ranks indicates that two algorithm performed significantly similar or different. For APFDc the IQR value the higher the better and for overhead the less the better. We look for the RQ1 and RQ2 answers with the results that we got from the experiment:



| ID | Algorithm Description | Utilized Information | | |
|---|---|---|---|---|
| | | Execution history | Test case description | Feedback |
| A1 | Execute test cases in random order. | | | |
| A2 | Execute test cases in optimal order. | | | |
| B1 | Execute test cases in ascending order of time since last failure. | ✓ | | |
| B2 | Execute test cases in descending order of number of times failed/number of times executed. | ✓ | | |
| B3 | Execute test cases in descending order of exponential decay metrics as in (3). | ✓ | | |
| B4 | Execute test cases in descending order of ROCKET metrics as in (4). | ✓ | | |
| B5 | Execute test cases in descending order of the Mahalanobis distance of each test case to the origin (0,0) when considering two metrics—time since last execution and failure rate. | ✓ | | |
| C1 | Execute test cases in ascending order of the estimated test case runtime. | ✓ | | |
| D1 | Supervised learning with Simple History (SH). | ✓ | ✓ | |
| D2 | Supervised learning with All History (AH). | ✓ | ✓ | |
| D3 | Supervised learning with Weighted History (WH). | ✓ | ✓ | |
| E1 | Dynamic test case prioritization with co-failure information. | ✓ | | ✓ |
| E2 | Dynamic test case prioritization with flipping history. | ✓ | | ✓ |
| E3 | Dynamic test case prioritization with rules mined from failure history. | ✓ | | ✓ |
| F1 | TERMINATOR with text feature. | | ✓ | ✓ |
| F2 | TERMINATOR with history feature. | ✓ | | ✓ |
| F3 | TERMINATOR with hybrid feature. | ✓ | ✓ | ✓ |

**Figure 6: Test Case Prioritization Algorithms and Information They Utilize[13]**

First we analyze the result we got from APFDc from figure 6

- A2 is the base line which is the upper limit. For A2 we have to have prior knowledge and it is not used in practice.
- F3, is TERMINATOR with feature using both history features and text features performed the best in terms of APFDc. Its failure detection rates are significantly higher than the nearest rank 3.
- In rank 3 history based algorithms (B2-B5, F2) are clustered and in second position.
- Rank 2 and rank1 contains the rest of the algorithms some of the are complex but in terms of performance they are not doing good.

F3 has a 9% higher APFDc than than rank 3 algorithms like B4. It answers RQ1.
**RQ1:** can TERMINATOR achieve significantly higher failure detection rates than other TCP algorithms when prioritizing automated UI tests from LexisNexis?
As seen in the Rank column of Figure 6, F3 (TERMINATOR with the hybrid feature) performed significantly better than other TCP methods. Further, TERMINATOR performed within 73/95 = 75% of the optimal (A2) in median

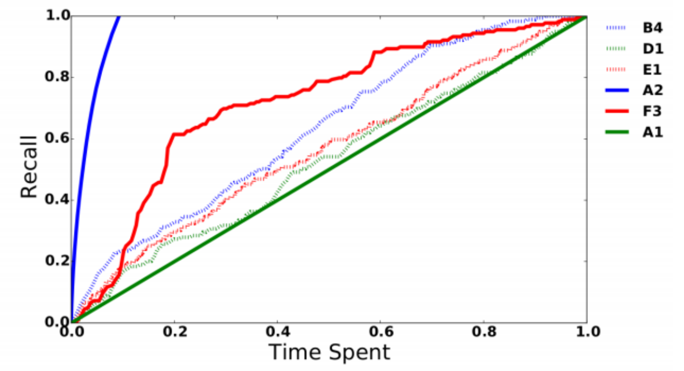
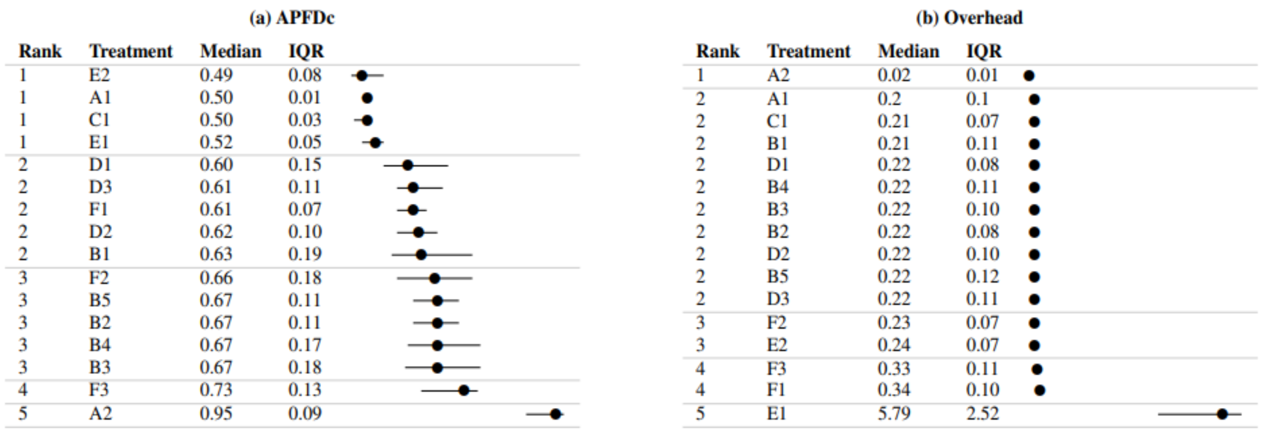


**Figure 7: : Failure detection curve of an example run at 02:02:34, October 24th, 2018. Here recall is the number of failures detected divided by the total number of failures in the test suite. An algorithm is considered better than another if it achieves higher recall with less time spent.**

To answer RQ2 the overhead results from figure 6 can be used.

- in terms of overhead we see that TERMINATOR (F3) has not done a good job. It has a higher runtime than simple history based ones like Group B. But compare to the cost it potentially save we can consider the result in favor of TERMINATOR.
- Group E and F dynamically adjust test cases for each run which increases the running time for them.

Overhead test answers the RQ2 for TERMINATOR:
RQ2: what is the computational overhead of TERMINATOR?
The computational overhead of TERMINATOR is 0.33% of the total runtime of the test suite, which is negligible compared to the cost it can potentially save by achieving higher failure detection rates.



(a) APFDc

| Rank | Treatment | Median | IQR |
|---|---|---|---|
| 1 | E2 | 0.49 | 0.08 |
| 1 | A1 | 0.50 | 0.01 |
| 1 | C1 | 0.50 | 0.03 |
| 1 | E1 | 0.52 | 0.05 |
| 2 | D1 | 0.60 | 0.15 |
| 2 | D3 | 0.61 | 0.11 |
| 2 | F1 | 0.61 | 0.07 |
| 2 | D2 | 0.62 | 0.10 |
| 2 | B1 | 0.63 | 0.19 |
| 3 | F2 | 0.66 | 0.18 |
| 3 | B5 | 0.67 | 0.11 |
| 3 | B2 | 0.67 | 0.11 |
| 3 | B4 | 0.67 | 0.17 |
| 3 | B3 | 0.67 | 0.18 |
| 4 | F3 | 0.73 | 0.13 |
| 5 | A2 | 0.95 | 0.09 |

(b) Overhead

| Rank | Treatment | Median | IQR |
|---|---|---|---|
| 1 | A2 | 0.02 | 0.01 |
| 2 | A1 | 0.2 | 0.1 |
| 2 | C1 | 0.21 | 0.07 |
| 2 | B1 | 0.21 | 0.11 |
| 2 | D1 | 0.22 | 0.08 |
| 2 | B4 | 0.22 | 0.11 |
| 2 | B3 | 0.22 | 0.10 |
| 2 | B2 | 0.22 | 0.08 |
| 2 | D2 | 0.22 | 0.10 |
| 2 | B5 | 0.22 | 0.12 |
| 2 | D3 | 0.22 | 0.11 |
| 3 | F2 | 0.23 | 0.07 |
| 3 | E2 | 0.24 | 0.07 |
| 4 | F3 | 0.33 | 0.11 |
| 4 | F1 | 0.34 | 0.10 |
| 5 | E1 | 5.79 | 2.52 |

**Figure 8: Here overhead is presented as percentages. Medians and IQRs show the 50th and (75-25)th percentile results for 49 simulations runs. Results are divided into "ranks" (shown in the left most columns). Results have the same rank if our statistical tests showed no significant different between them. Note that the result of E3 is not shown because it took too much time to mine the association rules from 2661 test cases. E3 ran for 30 hours and was still not finished. As a result, E3 has an overhead of more than 100% and is considered not useful no matter how high an APFDc it can achieve[13]**

In TERMINATOR, SVM is used to classify the test cases in fail or pass test cases. In this project to identify if it can improve the result if we change the classifier with Rendom Forest classifier. The result shows that it gives pretty similar result and does not show any improvement(Fig 9).

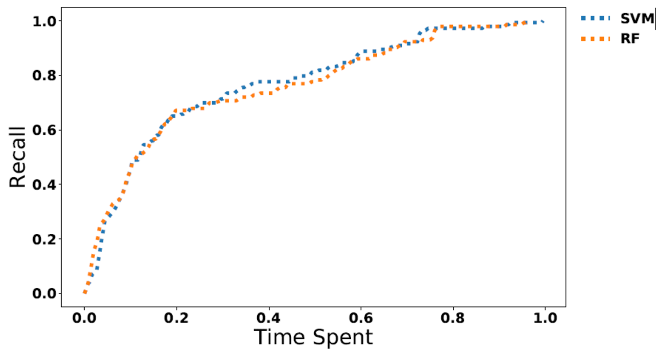


**Figure 9: Comparison of result achieved by SVM and Random Forest(RF)**

## 7 CONCLUSION AND FUTURE WORK

Faster feedback is essential for continuous integration process to make the software development life cycle shorter and keep the developers agile. Automated UI testing is an important key to faster testing. The idea of this project was to understand the TERMINATOR approach and applying a different classification algorithm to study the changes. From the experiments results we can conclude that TERMINATOR solves the automated test case prioritization problem by generalizing the problem to total recall and using active learning. Though with the results from the experiment asserts that TERMINATOR outperformed the current other approaches, still we have room to improve. In future we can try with different other classifiers and Hyper-parameter tuning to improve the current result, applying TERMINATOR to different dataset.

## ACKNOWLEDGMENTS

To Zhe Yu, for the help and suggestions

## REFERENCES

[1] 2001. Prioritizing test cases for regression testing. (2001).

[2] Arnaud Gotlieb Dusica Marijan and Sagar Sen. [n.d.]. *Test case prioritization for continuous regression testing: An industrial case study.* In Software Maintenance (ICSM), 2013 29th IEEE International Conference on. IEEE.

[3] Paul Hammant. [n.d.]. The Rise of Test Impact Analysis. ([n. d.]). https://martinfowler.com/articles/rise-test-impact-analysis.html

[4] Sebastian Elbaum Jingjing Liang and Gregg Rothermel. [n.d.]. *Redefining Prioritization: Continuous Prioritization for Continuous Integration.* n. In Proceedings of the 40th International Conference on Software Engineering (ICSE '18).

[5] Jung-Min Kim and Adam Porter. 2002. *A history-based test prioritization technique for regression testing in resource constrained environment.* In Proceedings of the 24th international conference on software engineering.

[6] LEGAL. 2019. (2019). https://doi.org/10.1145/1219092.1219093

[7] ANDREW JHON Scott and M Knott. 1974. A cluster analysis method for grouping means in the analysis of variance.

[8] Alexey Malishevsky Sebastian Elbaum and Gregg Rothermel. 2001. *Incorporating varying test costs and fault severities into test case prioritization.* 329–338.

[9] Burr Settles. 2012. . Synthesis Lectures on Artificial Intelligence and Machine Learning. *Commun. ACM* 6, 1 (2012), 31−−114.

[10] Kuan-Li Peng Yen-Ching Hsu and Chin-Yu Huang. 2014. *A study of applying severity-weighted greedy algorithm to software test case prioritization during testing.* IEEE International Conference.

[11] Zhe Yu and Tim Menzies. [n.d.]. Total Recall, Language Processing, and Software Engineering. *In Proceedings of Workshop on NLP for Software Engineering (NL4SE@ ESEC/FSE 2018)* ([n. d.]).

[12] Emad Shihab Yuecai Zhu and Peter C Rigby. [n.d.]. Test Re-prioritization in Continuous Testing Environments. 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME).

[13] Tim Menzies Gregg Rothermel Kyle Patrick Snehit Cherian Zhe Yu, Fahmid Fahid. 2019. TERMINATOR: Better Automated UI Test Case Prioritization. *In Proceedings of the 27th ACM Joint European Software Engineering* (2019).

[14] Mark Harman Zheng Li and Robert M Hierons. 2007. *Search algorithms for regression test case prioritization.* Number 33. IEEE Transactions on software engineering.