Test Case Prioritization Optimization with Machine Learning

João Luís Xavier Barreira Lousada
joao.b.lousada@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

December 2020

Abstract

In modern software engineering, Continuous Integration (CI) has become an indispensable step towards systematically managing the life cycles of software development. Large companies struggle with keeping the pipeline updated and operational, in useful time, due to the large amount of changes and addition of features, that build on top of each other and have several developers, working on different platforms. Associated with such software changes, there is always a strong component of Testing. As teams and projects grow, exhaustive testing quickly becomes inhibitive, becoming adamant to select the most promising test cases earlier, without compromising software quality. After proving to be a strategy as good as traditional prioritization methods in three different datasets, we test its ability to adapt to new environments, by testing it on novel data extracted from a financial institution, yielding a Normalized percentage of Fault Detection (NAPFD) of over 0.6 for the Network Approximator and Test Case Failure Reward. Additionally, we studied the impact of experimenting a new model for memory representation: Decision Tree Approximator, without producing significant improvements relative to Artificial Neural Networks. Neural Network Embeeding for Test Case Prioritization (NNE-TCP) is a novel Machine-Learning (ML) framework that analyses which files were modified when there was a test status transition and learns relationships between these files and tests by mapping them into multidimensional vectors and grouping them by similarity. When new changes are made, tests that are more likely to be linked to the files modified are prioritized. Furthermore, NNE-TCP enables entity visualization in low-dimensional space, allowing for other manners of grouping files and tests. By applying NNE-TCP, we show for the first time that the connection between modified files and tests is relevant and competitive relative to comparison methods. This research is carried out for Instituto Superior Técnico in collaboration with BNP Paribas. I have benefited from a fellowship of the BNP Paribas, in the framework of the IST Technology Transfer Office.

Keywords: Continuous Integration, Regression Testing, Test Case Prioritization, Machine Learning

1. Introduction

Given the complexity of modern software systems, it is increasingly crucial to maintain quality and reliability, in a time-saving and cost-effective manner, especially in large and fast-paced companies. This is why many industries adopt a *Continuous Integration* (CI) strategy, a popular software development technique in which engineers frequently merge their latest code changes, through a *commit*, into the mainline codebase, allowing them to easily and cost-effectively check that their code can successfully pass tests across various system environments [9].

1.1. Regression Testing

One of the tools used to manage software change is called regression testing. It is critical to ensure that the introduction of new features, or the fixing of known issues, is not only correct, but also does not obstruct existing functionalities. *Regressions* occur when a software bug causes an existing feature to stop functioning as expected after a given change and can have many origins (e.g. code not compiling, performance dropping, etc.), and, as more changes occur, the probability that one of them introduces a fault increases [10]. On the other hand, *progressions* occur when the software bug that was at the origin of a regression is fixed, restoring the feature proper functioning.

As software development teams grow, identifying and fixing regressions quickly becomes one of the most challenging, costly and time-consuming tasks in the software development life-cycle, rapidly inhibiting its adoption. Such teams often resort to modern large-scale test infrastructures, like coregrids or online servers [14]. Consequently, in the

last decades, there has been intensive research into solutions that optimize Regression Testing, accelerating fault detection rates either by alleviating the amount of computer resources needed or by reducing feedback time, i.e. the time delay between a software change and the information regarding whether it impacts the system's stability [1, 2, 6].

One of the most prominent techniques for Regression Testing optimization is Test Case Prioritization (TCP), that aims to find the optimal permutation of test cases that match a certain target, i.e. the ability to reveal failures as soon as possible, which is useful when time budget or computer resources are limited [10]. Another common technique is Test Case Selection (TCS), where only a relevant subset of all the tests is chosen.

The key aspect of this study is how to know *a priori*, which test cases to prioritize i.e. without running them. One possibility is to have a professional test engineer cherry-pick the most promising test cases. Unfortunately manual test case executing is time-consuming, counter-productive and error-prone [4], and is not scalable. Therefore, there has been a high demand for techniques that can automatically pick test cases, minimizing human intervention [14].

1.2. Machine Learning (ML)

In recent years, the field of Artificial Intelligence as been expanding at an astounding pace, fueled by the growth of computer power and the amount of available data. Some problems can not be solved by traditional algorithms, due to limited or incomplete information. In our case, we do not know which tests are more relevant to apply first. However this information can be extracted from historical data and it can be learned to enable future predictions, by estimating the probability of being relevant, e.g. probability of regression, progression or or both, a transition. Hence, with the rise of data availability, there has been a growing interest in solutions that involve learning from data. Particularly, in this thesis two ML framework were developed: one adapted from the literature, based on Reinforcement Learning and one from scratch, based on Neural Network Embeddings.

2. Background

In software engineering, version control systems are a means of keeping track of incremental versions of files and documents, allowing the user to arbitrarily explore and recall the past commits that lead to that specific version[9]. Testing is a verification method used to assess the quality of a given software version. The building block of software testing is the test case, which specifies on which conditions the System Under Test (SUT) must be executed in order to detect a fault, i.e. for a given input, what are the expected outputs [4].

When test cases are applied, the outcome obtained is in the form of PASS/FAIL, with the purpose of verifying functionality or detecting errors. However, testing is very much like sticking pins into a doll - to cover its whole surface a lot of pins are needed, and the larger the doll, the more pins we require. Likewise, the larger and more complex the SUT, the greater the variety of test cases required. Therefore, to ensure that the health of the SUT is maintained throughout time, exhaustive testing is required to cover all possible scenarios [10].

Inevitably, this task becomes impractical or even unfeasible due to the increasing complexity of the SUT, so testers have to find scalable approaches to counteract exhaustive testing, usually resorting to three techniques: Test Case Minimization, Test Case Selection and Test Case Prioritization (TCP), the latter being the target of this work.

2.1. Test Case Prioritization

As mentioned before, TCP rearranges test cases according to a given criteria, such as the probability of revealing faults.

Definition 2.1. ***TCP*** *Given the set of test cases, T, the set containing the permutations of T, PT, and a function from PT to real numbers $f : PT \to \mathbb{R}$, find a subset T' such that*

$$[f(T') \geq f(T'')], \quad \forall T'' \in PT. \tag{1}$$

In TCP, the function f should be some relevant criteria such as code coverage, early fault detection, fewer resource demand, etc. [10].

One possible way of evaluating the optimal permutation is to compute the Average Percentage of Fault Detection (APFD) metric [8].

Definition 2.2. ***APFD*** *Let T be the set of tests containing n test cases and F the set of m faults revealed by T. Let TF_i be the position, in a given permutation, of the first test case that reveals the i^{th} fault [10]. Thus, APFD is defined as*

$$\text{APFD} = 1 - \frac{TF_1 + \cdots + TF_n}{nm} + \frac{1}{2n}. \tag{2}$$

Simply put, higher values of APFD, imply higher fault detection rates, i.e. when APFD has value 1, all the failing tests are applied before all the ones that are passing, whilst when it has value 0, all the failing tests are applied at the end of the permutation. Furthermore, the APFD metric can be extended to the case where only a subset of T_i is executed, called Normalized Average Percentage of Fault Detections (NAPFD) [11].

In the literature, APFD is the most common metric for measuring TCP performance across different methods [8]. Nonetheless, ee define, for the first time, a metric for Test Case Prioritization based on test case transitions, the Average Percentage of Transition Detection (APTD).

Definition 2.3. ***APTD*** *Let T be the set of tests containing n test cases and τ the set of m transitions revealed by T. Let $T\tau_i$ be the order of the first test case that reveals the i^{th} transition.*

$$APTD = 1 - \frac{T\tau_1 + \cdots + T\tau_n}{nm} + \frac{1}{2n}. \quad (3)$$

Therefore, similar to the APFD metric, if the APTD is 1, all test cases that will suffer transitions are applied first, and if near 0, all relevant test cases will be the last to be executed. By being able to create schedules that have a high APTD, we shorten the the time needed to both detect newly introduced regressions and find possible progressions.

2.2. Reinforcement Learning

Reinforcement Learning (RL) handles problems that involve learning which course of action to take, given a set of possible states and possible actions. Each action taken produces a given reward. In RL, the goal of an *agent*, i.e. the decision maker, is to interact with the environment and select the actions that maximize the cumulative sum of the reward signal. The agent's ability to design an optimal strategy is strongly dependent on three factors: the way the reward function is defined, which features are fed into the model and its ability to generalize instead of memorize.

When dealing with a large dataset, wherein state space representation complexity grows, it is not feasible to represent the state space in a tabular manner, i.e. store the state space discretely in a table. Hence, to reduce the memory needed to represent the state space, we use Approximators - a viable alternative of memory representation. These can be ML algorithms (e.g. Artificial Neural Networks (ANNs), Decision Trees (DTs), Nearest Neighbours) [12].

In the context of TCP, we want the agent to learn how to rank test cases, such that the ones that are more prone to reveal faults have a higher priority than the ones that are not. First, each test case is prioritized individually, so that a test schedule is created, executed and, finally, evaluated. Traditionally, the only information provided to the algorithm is historical results. The main contributors to this field were Spieker *et al.* [11], that implemented *Reinforced Test Case Selection* (RETECS) - the framework adapted in this thesis - and, later, Wu *et al.* [13] extended their work. In both publications, the method preferably prioritizes test cases which have been failing recently.

2.3. Neural Network Embeddings

The goal of embeddings is to map high-dimensional categorical variables into a low-dimensional learned representation that places similar entities closer together in the embedding space. This can be achieved by training a neural network [5].

One-Hot Encoding, the process of mapping discrete variables to a vector of 0's and 1's, is commonly used to transform categorical variables, i.e. variables whose value represents a category (e.g. the variable *color* can take the value *red*, *blue*, *purple*, etc.), into inputs that ML models can understand. One-Hot encoding is a simple embedding where each category is mapped to a different vector (e.g. *red*, *blue*, *purple* can correspond, respectively, to $[1, 0, 0]$,$[0, 1, 0]$ and $[0, 0, 1]$).

This technique has two severe limitations: first, dealing with high-cardinality categories, (e.g. trying to map each possible color with this method would be unfeasible), and secondly, mappings are "blind", since vectors representing similar categories are not grouped by similarity (e.g. in this representation, the category *purple* is no closer to *blue* than it is to *yellow*).

Thus, to drastically reduce the dimensionality of the input space and also have a more meaningful representation of categories, we could introduce *embeddings*, lower dimensional vectors that represent categories by mapping similar categories to similar vectors. For example, we could map the variable *color* to a lower dimensional space, *red* $= [1, 0]$, *blue* $= [0, 1]$ and *purple* $= [1, 1]$, by taking advantage of the fact that *purple* is a combination of *red* and *blue*.

In our case, we take each file and each test and represent them as n-dimensional vectors, with the goal of representing similar files and similar tests as similar vectors. The key aspect of embeddings is that these n-dimensional vectors are trainable , which means that each vector component can be adjusted in order to push vectors representing of related objects together. As a result, after training, the supervised learning task will be able to predict whether two categories are similar.

3. Study I: RETECS

3.1. Implementation

RL is an adaptive method where an agent learns how to interact with an environment that responds with reward signals, which correspond to the feedback of taking a certain action. For example, when driving a car, the environment is the real-world, the state can be defined as the position, speed of the car and neighbouring cars. Possible actions are

turning the wheel and accelerating or braking. The reward function can be calculated by how much time it took to reach the destination, while respecting traffic rules.

These back-and-forth interactions take place recurrently: the agent receives some representation of the environment's state and selects actions either from a learned policy - mapping from perceived states of the environment to actions to be taken when in those states - or by random exploration - randomly choosing an action when in a given state to account for environment changes. Consequently, the environment responds to these actions and presents new situations to the agent, which finds itself in a new state after exercising its action. The main goal of RL is to maximize the cumulative sum of rewards, rather than just consider immediate rewards [12]. Following the example above, a possible action to take is to increase the travel speed of the car. If there are no accidents until the final destination, then the reward receive will be higher.

More formally, considering a set of discrete time steps, $t = 0, 1, 2, ...$, the representation of the environment's state at time t is defined as $S_t \in S$, where S is the set of all possible states. In S_t, the agent has the option to select an action $A_t \in A(S_t)$, where $A(S_t)$ corresponds to the set of actions accessible from state S_t. By applying A_t to state S_t, the agent finds itself, one time step later, in a new state S_{t+1} and a reward R_{t+1} is collected as feedback from action A_t.

In the context of TCP, RETECS prioritizes each test case individually and, with these prioritizations, a test schedule is then created, executed and finally evaluated. Each state represents a single test case $t_i \in T_i$, and it contains information on the test's duration, when it was last executed and the previous execution results. The set of possible actions corresponds to the set of all possible prioritizations a given test case can have in a commit, which is translated into an integer, e.g. if test A has prioritization equal to one, then it will be the first to be executed. After all test cases are prioritized and submitted for execution, the respective rewards are attributed based on the test case status. From this reward value, the agent can adapt its strategy for future situations: an action yielding positive rewards is reinforced, whereas negative rewards discourage the current behaviour.

The RETECS framework has the following characteristics:

1. **Model-Free** - has no initial concept of the environment's dynamics or how its actions will affect it.

2. **Online** - learns *on-the-fly*, adapting to a dynamic environment. It is particularly relevant in environments where test failure indicators change over time, so it is adamant to update the agent's strategy.

3.2. Results

In our experiments, we trained two RL agents. The first resorts to an Network representation of states, while the second uses a DT. On both cases, the reward function varies between failure count, test-case failure and time ranked. For each test agent, test-cases are scheduled in descending order of priority and until the time limit is reached, if there is one. It is worth noticing that the *IOF/ROL* and *Paint Control* datasets were already analysed by Spieker *et al.* [11]. In this work, we replicate their result to ensure that no errors were introduced by us, while adapting their framework. Therefore we will only describe the novelties.

Fig. 1 shows a comparison of the prioritization performance between the Network Agent and the DT Agent, with regards to different reward functions (rows), applied to three different datasets (columns). The commit identifier is represented in the x-axis and for each one there is a corresponding NAPFD value, ranging from 0 to 1. (represented as a line plot in red and blue for the Network and DT agents, respectively). The straight lines show the overall trends of each configuration, which is obtained by fitting a linear function - full line for Network and dot-dashed line for DT Approximator. It is clear to see that Test Case Failure reward function produces the best results, in terms of maximizing the slope of the NAPFD trend. When combined with the Network Approximator, this approach proves to be the best configuration overall, for the three datasets, where we see a more significant growth in the trend line, indicating that the algorithm is learning from the data.

The supremacy of the Network Approximator remains valid for the reward function that produces the best results. Yet, in some cases, the DT Approximator was able to surpass its performance by a small amount. If, for example, the *Finance* dataset had more records, it is possible that DT would follow the growing trend and surpass Network by a significant amount. Therefore, the collection of more data is crucial to correctly evaluate the DT Approximator's performance. Choosing the best configuration, test case failure reward and the Network Approximator, when RETECS is applied in an environment completely different from Robotics and with different characteristics, it was able to adapt and learn how to effectively prioritize test cases. This shows that the RETECS domain of validity expands to distinct CI environments, which is particularly useful for companies that increasingly rely on the health of these systems.

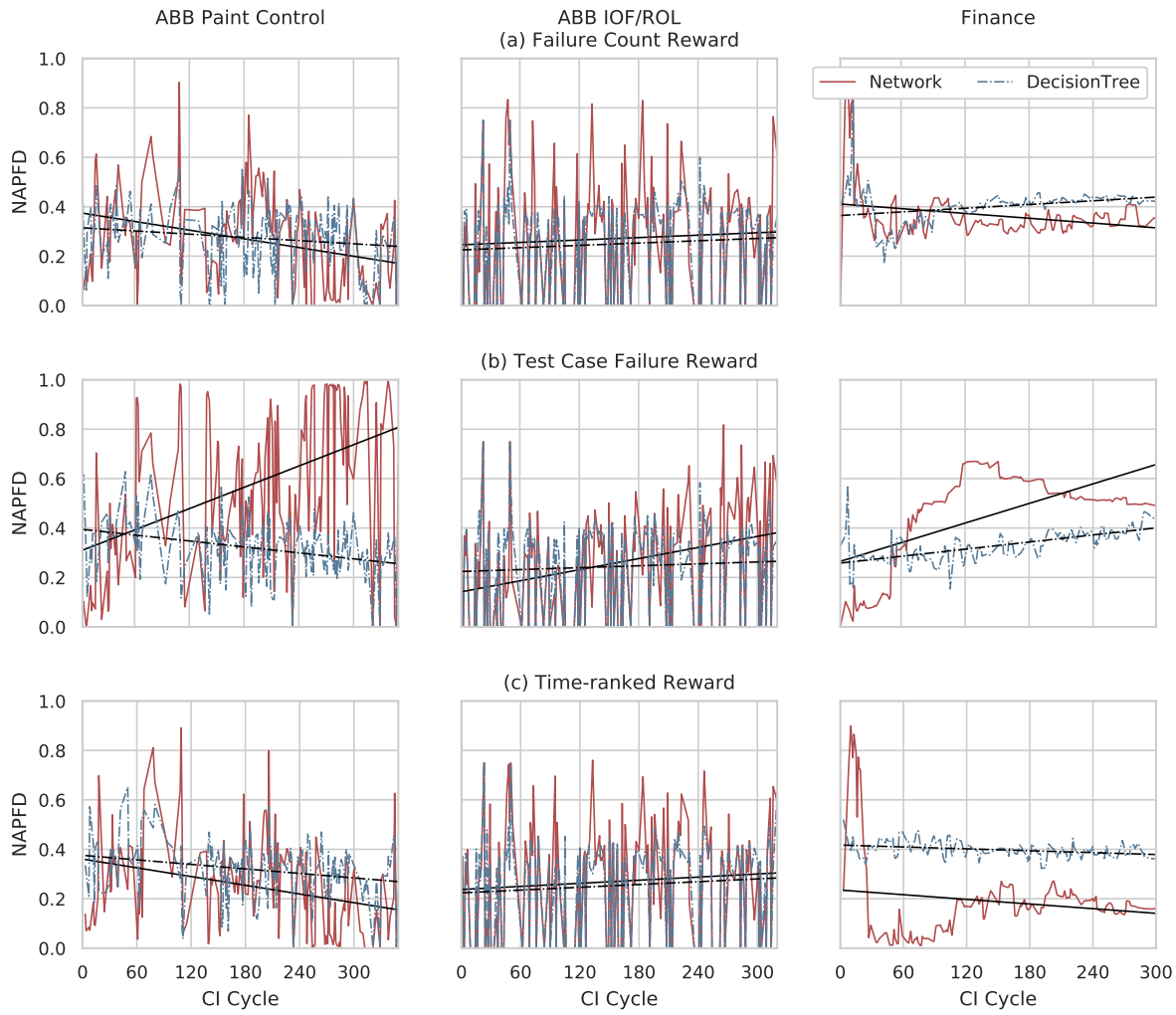

Figure 1: NAPFD Comparison with different Reward Functions and memory representations: best combination obtained for Test Case Failure reward and Network Approximator (straight lines indicate trend)

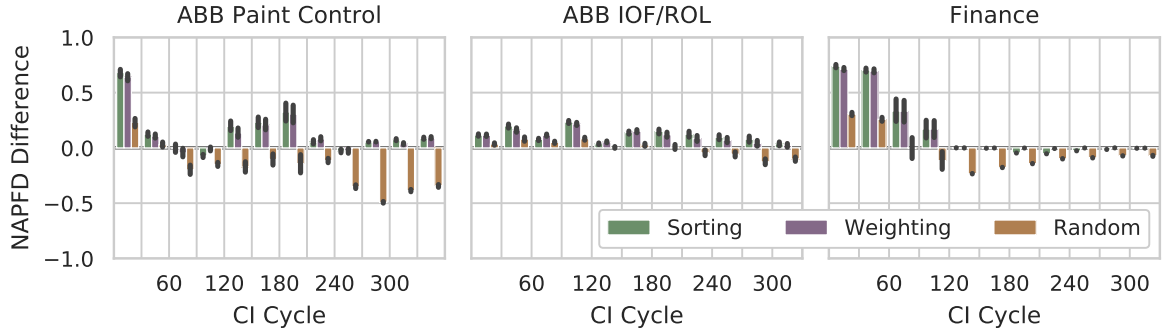

Figure 2: NAPFD difference in comparison to traditional methods: Random, Sorting and Weighting. Positive differences indicate better performance from traditional methods and negative differences show better performance for RETECS

Traditional Methods Three methods were included as a means of comparison: *Random*, *Sorting* and *Weighting*.

- **Random** assigns random prioritizations to each test case, and this serves as a baseline method. The other two methods are deterministic.
- **Sorting** method sorts each test case according to its most recent status, i.e. if a test case failed recently it has higher priority.
- **Weighting** method is a naive version of RETECS without adaptation, because it considers the same information - duration, last run and execution history - but uses a weighted sum with equal weights.

The results are depicted in Fig. 2 as the difference between the NAPFD for the traditional methods and RETECS over several CI Cycles. Each bar comprises 30 commits. For positive differences, the traditional methods have better performance, and on the contrary negative differences show the opposite.

For the *Finance* data, there is clearly a learning pattern with an adaptation phase of around 90 commits which the RETECS method requires to have a similar performance to the traditional methods and a significant improvement with respect to Random. Then for the following commits, Random method progressively catches up with other methods, which can be a sign of a mutating environment, i.e. test cases at commit 300 are not failing for the same reasons that they were in commit 90. Overall, the algorithm achieves promising results, when applied to this novel dataset.

It is evident that RETECS can not perform significantly better than traditional methods. RETECS starts without any representation of the environment and it is not specifically programmed to pursue any given strategy. Yet it is possible to make prioritizations as good as traditional methods commonly used in the industry and by increasing the number of records available on each dataset, adding more features and conducting a more refined parameter tuning analysis, there is strong evidence that there can be a performance boost.

4. Study II: NNE-TCP

4.1. Implementation

Our NNE-TCP approach is sustained by a predictive model that tries to learn whether a modified file and a test case are linked or not. After training, the model can be used to make new predictions on unseen data and create test schedules that prioritize test cases more likely to be related with files modified in a given commit.

The steps taken to develop the framework were:

1. Load and Clean Dataset.
2. Create Training Set.
3. Build Neural Network Embedding Model.
4. Train Neural Network Model.
5. Evaluate Model's Performance.
6. Visualize Embeddings using dimensionality reduction techniques.

1) The dataset should contain records of the files modified in every commit, as well as test cases that suffered transitions. Then to obtain a cleaner dataset, to obtain better results, it is useful to eliminate redundant or outdated files and tests as well as removing files and tests that have not been modified or transitioned recently. Three important parameters were defined to clean data:

- **Date Threshold**: timestamp from which we consider file/test modifications/executions - if a file/test has not been modified/executed for n months, it is considered deprecated and is removed.
- **Individual Threshold**: the individual frequency of each element - files and tests that appear fewer than n times are removed, because they are likely to be irrelevant.
- **Pairs Threshold**: frequency of file/test pairs - pairs that occur fewer than n times are likely to have happened by chance, so they are removed.

To remove the noise from the data we need to remove files, tests and pairs that rarely are part of a commit in the dataset. A file/test is said to occur in a commit, if it was modified/executed. We want the average number of occurrences per file/test to be larger, leading to a higher density of relevant files and tests, in order to obtain a higher quality dataset.

2) The supervised learning task we are trying to solve can be stated as: given a file and a test case, predict whether the pair is linked, i.e. predict if the modification of a given file could impact the outcome of a given test case's execution, based on the commit history. Subsequently, the training set will be composed of pairs of the form: $(file, test, label)$. The label will indicate the ground-truth of whether the pair is positve or negative - is or is not present in the dataset.

To create the training set, we need to iterate through each commit and store all pair-wise combinations of files and test cases. In any of the commits, if there is a test case that suffered a transition

where a given file was modified, then that file and test case constitute a positive pair.

Then, because the dataset only contains positive examples of linked pairs, in order to create a more balanced dataset we need to generate negative examples, i.e. file-test pairs that are not linked. A negative example constitutes test case that is not linked to a modified file.

As a result of having to create balanced examples for every commit and specially when dealing with large datasets, it could become unpractical, in terms of memory and processing power, to generate and store the entire training set at once. Consequently, to alleviate this issue, the *Keras* class *Data Generator* offers an alternative to the traditional method, by generating data *on-the-fly* - one batch (i.e. subset of the entire dataset) of data at a time [3].

3) Having created the training set, the next step is to build the learning model's architecture. The inputs of the neural network are each (file, test) pairs, that will be mapped to a n-dimensional embedding vector. Afterwards, the dot product of both vectors is computed, to determine the similarity between them. Subsequently, the output will be a prediction of whether or not - positive values vs. negative values of the dot product - there is a link.

The *Keras* Deep Learning model is depicted below in Fig. 3 and is composed of the following layers:

- **Input**: 2 neurons. One for each file and test case in a pair.
- **Embedding**: map each file and test case to a n-dimensional vector.
- **Dot**: calculate the dot product between the two vectors, merging the embedding layers.
- **Reshape**: reshape the dot product into a single number.
- [Optional] **Dense**: generate output for classification with sigmoid activation function.

The *Dense* layer is optional, because the supervised learning task can be classification or regression.

4) Having built the model's architecture, the next step is to train it with examples produced by the Data Generator, for a certain number of epochs. At this stage, the weights are updated such that the respective loss function is minimized and the accuracy when predicting whether the pair is positive or negative is maximized. If the algorithm converges, the model is ready to make predictions on unseen data and produce meaningful representations of file and test case embeddings.

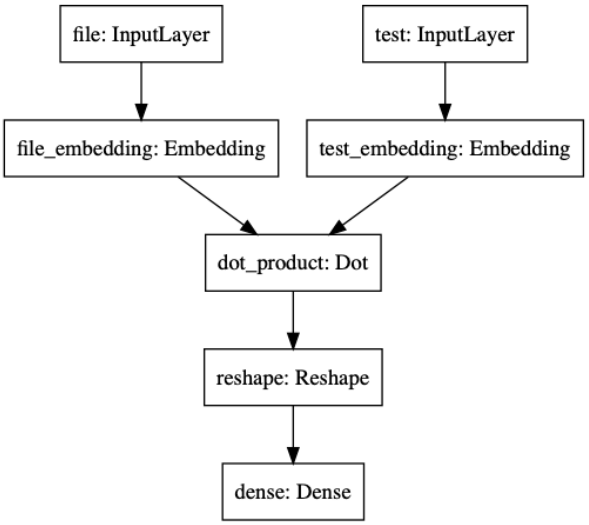

Figure 3: Neural Network Embedding Model Architecture

5) After training the model, we are able to make predictions on new, unseen commits. We can evaluate the true accuracy of our model using the test set. To evaluate the model's performance, we measure its APTD.

6) Lastly, a useful application of training embeddings is the possibility of representing the embedding vectors in a reduced dimensional space, providing a helpful intuition about entity representation. Since embeddings are represented in an n-dimensional manifold, one has to resort to manifold reduction techniques to represent elements in 2D or 3D, in order for the manifold to be understandable by humans.

4.2. Results

After framing the problem and having the data cleaned, we can move on to training the model. This process consists of learning the embedding representation, by updating the neural network's weights, through backpropagation. Additionally, we have some hyper-parameters that influence the model's performance, and these need to be fine-tuned to reach the best results. Some examples of hyper parameters are the embedding size, batch size and negative-ratio. Fine-tuning analysis allows us to find the best combination of parameters that maximize our metrics, by experimenting different values. To determine them, a grid search was conducted, covering different combinations between possible values for each parameter and the best combination was used to produce the histogram depicted in Fig. 4

We can conclude that the model managed to produce a desirable result, with a fairly high APTD. It can also be seen that highest bar in the histogram corresponds to an APTD near 1, which may indicate that, for some commits, the algorithm was

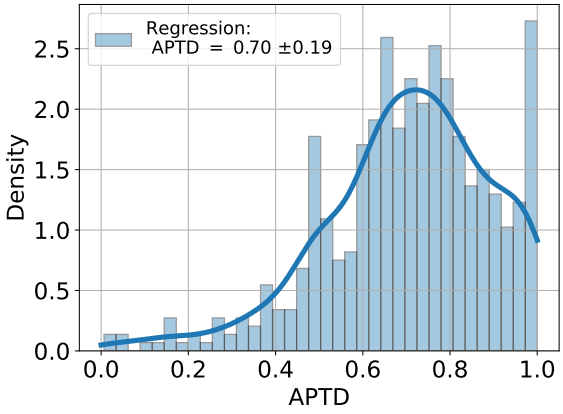

Figure 4: APTD histogram and density distribution functior (Blue line) for optimal parameter combination

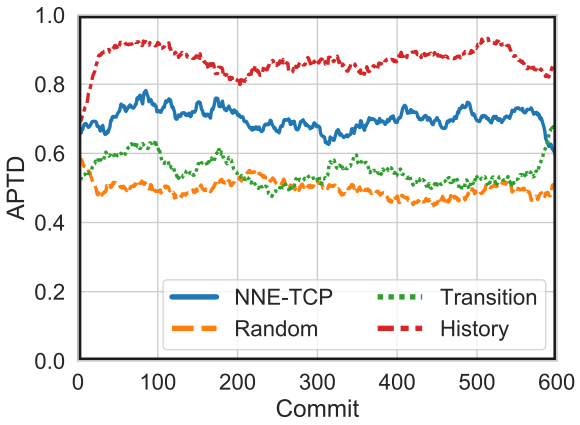

Figure 5: APTD Trend of NNE-TCP and traditional methods for the test set. Each line is obtained by calculating the rolling average over a 50 commit window (for a total of 600 commits).

able to correctly predict which tests would suffer transitions and prioritized them first.

Traditional Methods Having calibrated and trained the NNE-TCP model to produce the best APTD value, we are now able to study how this new approach performs when compared to the three traditional methods, described below. The results of this comparison are depicted in Fig. 5.

- **Random** - each test is assigned a random prioritization. This method will serve as a baseline for comparison.
- **Transition** - a fixed prioritization of the test cases is used across every commit. Tests are ranked by their rate of transition, determined by the number of times they have suffered transitions in the past.
- **History** - tests that suffered transitions more recently are assigned a higher rank. This approach is based on two assumptions: regressions are likely to be fixed quickly, i.e. a test that just started failing should transition through a progression soon, and stable test cases, i.e. tests that rarely have any transitions are less relevant. Furthermore, while a project is in state of active development, it is likely that only the same subset of test cases will be involved in transitions, which will be prioritized by *History*. However, this approach does not take into account the fact that a test case that has been failing for some time is more likely to be fixed than a test that had just started failing.

Out of the four methods presented, *History* is the one that shows higher APTD trends, followed by our approach NNE-TCP and, further down, *Transition* and *Random*, respectively. Table 1 shows the average value for the APTD and the root-mean-squared error (RMSE) - the average of the quadratic difference between two curves - between NNE-TCP and the other traditional methods.

Method	Mean APTD	RMSE
NNE-TCP	0.70	0
Random	0.50	0.21
Transition	0.59	0.16
History	0.87	0.17

Table 1: Performance comparison between TCP methods. The RMSE value is calculated in relation to NNE-TCP

As expected for a baseline method, *Random* (yellow line) has an approximately constant APTD trend of around 0.5, meaning that, on average, relevant test cases are not ranked in the beginning nor the end of the test schedule, but are rather uniformly distributed.

Looking at the *Transition* method (green line) it is possible to see a slight improvement relative to *Random*. Although this method only considers the frequency of past transitions for each test case it is able to assign less relevance to stable tests, i.e. test that almost do not cause transitions, have low priority and are therefore ranked at the bottom of the schedule.

The *History* method (red line) was able to achieve the highest trend of the four methods, with the assumption that tests that suffered transitions recently, are more likely to transition again. In short, the longer a test remains stable, the less relevant it becomes and, consequently the lower priority it has. On the other hand, when a regression is detected, it is expected that it will be fixed soon, since the bug-source has clearly been identified, causing a progression. For this reason, a test that transitioned recently through a regression is more likely to transition again through a progression. All these factors help explain the success of the *History* method.

It is worth noticing that NNE-TCP is a novel approach that is evaluated by the equally novel APTD metric and, to the best of our knowledge, there are no other ML frameworks whose performance can be compared to NNE-TCP. Thus, it is of the utmost importance that we validate the model against other traditional methods, that do not learn from experience.

Entity Representation One of the advantages of using embeddings is that, after solving a Supervised Learning problem, the results can be represented in low dimensional space with UMAP [7]. It aims to find a low dimensional projection of the embeddings, while maintaining the same high-dimensional topological structure. This allows us to project multi-dimensional entities into two dimensions, granting us the possibility to observe some clusters of files and test cases with the same behaviour.

Fig. 6 shows the embedding representation for files and test cases and by labelling them according to the corresponding folder where they are stored in the system. With these results, we can see that in the test case projection, some dots overlap. This is a clear indicator of test case redundancy, i.e. test cases that suffer transitions concurrently, meaning they must have a very high degree of similarity. This information can then be used by a test engineer to inspect these particular test cases and, if applicable, clean the redundant ones. Also, because we can not observe any single color clusters in the projections, it is possible to conclude from the plot that there is no correlation between the folder where files/tests are stored and whether they cause/suffer transitions together. It should be noted that there were cases where files and tests' folder names were corrupted or unavailable.

5. Threats to Validity and Future Work

The first threat, associated with both ML frameworks, is the quality of the dataset. It is relatively small, facing the number of test cases and modified files it encompasses. Adittionally, there is strong evidence for the presence of noise, therefore collecting more data is a crucial step for Machine Learning models to learn better and more complex relations between inputs and outputs.

Regarding Study I, it is not possible yet to compare the APTD metric, defined by us, measured by NNE-TCP with other ML algorithms defined in the literature. Moreover, NNE-TCP was only applied to one dataset from a specific industrial environment. Hence, it is crucial to both measure the APTD with other frameworks but also to evaluate the performance of NNE-TCP in other environments.

Regarding Study II, RETECS' major limitation arises form only considering three features to describe a test case - its duration, timestamp of last execution and failure history - that can and should be extended to have a more complete picture of what a possible failing test case looks like. Also Decision Trees were used for the first time, in this framework, and there are more ML algorithms that can serve the same purpose, e.g. Nearest Neighbours.

In both studies, due to limited time and computer power, parameter tuning analysis was limited, but it can be further refined by exploring more combinations of parameters and the respective impact on evaluation metrics.

Finally, we propose that the two algorithms are "merged", combining test history features with file linkage. This way, by prioritizing not only tests that have a strong failing history, but also only the relevant ones given a commit, we expect to maximize performance.

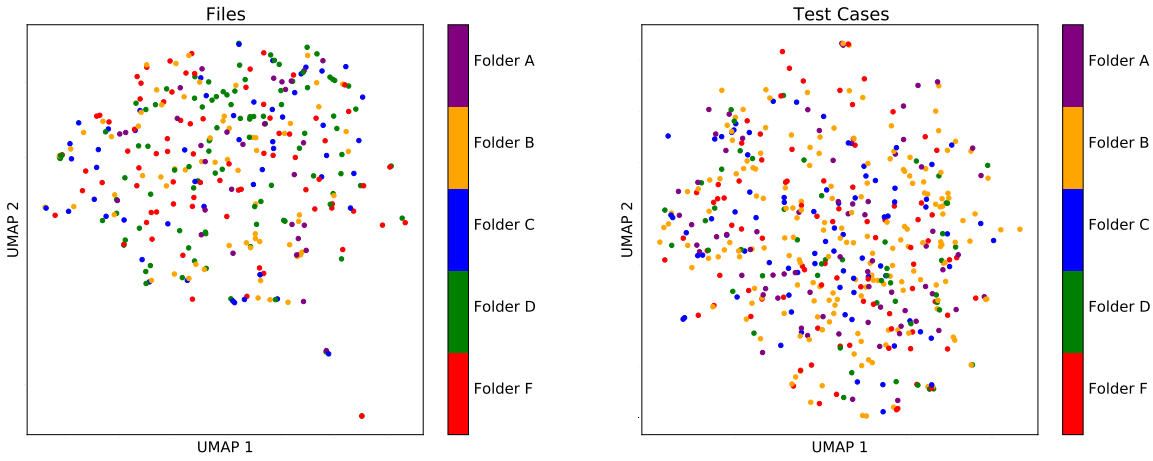

Figure 6: Labelled Embeddings corresponding to the five most populated folders where files/test cases are stored in the system

6. Conclusions

´ In this study, an extension of the RETECS framework was developed, as well an original approach to Test Case Prioritization using Machine Learning, called NNE-TCP.

Results indicate that RETECS can effectively create meaningful test schedules in different contexts. In the new *Finance* dataset, with a combination of the Test Case Failure reward with the Artificial Neural Network Approximator, around 90 commits suffice to reach the performance of deterministic methods and surpass random prioritization of test-cases. Initially, the evaluation metric NAPFD starts at a value of only 0.2 but, as the algorithm progresses, the trend shows values over 0.6. The inclusion of DT's in the framework failed to produce better results relative to the Network, in the best possible case. However in some cases, with other reward functions, performance is comparable and might not be discarded right away, as it can be useful to apply, in future research, to other CI environments with distinct characteristics.

Our results point to a major improvement over some traditional TCP methods, which we called *Random* and *Transition*. However, NNE-TCP was unable to match the performance of a third, more complex traditional method, called *History*. Nevertheless, we strongly believe that NNE-TCP has enough potential to reach higher performance levels. If a mapping between files and tests can be effectively learned by a data-driven approach, then only relevant tests will be executed, reducing feedback time. To validate this hypothesis, further experiments must be conducted on richer and cleaner datasets. Finally, the ability to visualise embeddings in 2D space represents a valuable improvement over other commonly used methods, providing insights on the structure of the data. We showed that there does not seem to be exist any correlation between the folders where files/tests are stored and the similarity between the files/tests themselves. Notwithstanding, it is already possible to detect possibly redundant tests and discover dependencies between files.

References

[1] S. Ananthanarayanan, M. S. Ardekani, D. Haenikel, B. Varadarajan, S. Soriano, D. Patel, and A.-R. Adl-Tabatabai. Keeping master green at scale. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 2019.

[2] B. Busjaeger and T. Xie. Learning for test prioritization: An industrial case study. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 975–980, New York, NY, USA, 2016. Association for Computing Machinery.

[3] F. Chollet et al. Keras, 2015.

[4] V. H. S. Durelli, R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. C. Dias, and M. P. Guimarães. Machine learning applied to software testing: A systematic mapping study. *IEEE Transactions on Reliability*, 68(3):1189–1212, 2019.

[5] A. Geron. *Hands-on machine learning with Scikit-Learn and TensorFlow*. O'Reilly, 2017.

[6] J. Liang, S. Elbaum, and G. Rothermel. Redefining prioritization: Continuous prioritization for continuous integration. 2018.

[7] L. McInnes, J. Healy, and J. Melville. Umap: Uniform manifold approximation and projection for dimension reduction, 2018.

[8] G. Rothermel, R. J. Untch, and C. Chu. Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.*, 2001.

[9] M. Santolucito, J. Zhang, E. Zhai, and R. Piskac. Statically verifying continuous integration configurations, 2018.

[10] Y. Shin. *Extending the Boundaries in Regression Testing: Complexity, Latency, and Expertise*. PhD thesis, King's College London, 2009.

[11] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige. Reinforcement learning for automatic test case prioritization and selection in continuous integration. *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2017*, 2017.

[12] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[13] Z. Wu, Y. Yang, Z. Li, and R. Zhao. A time window based reinforcement learning reward for test case prioritization in continuous integration. 2019.

[14] C. Ziftci and J. Reardon. Who broke the build? automatically identifying changes that induce test failures in continuous integration at google scale. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP '17. IEEE Press, 2017.