TÉCNICO LISBOA

Machine Learning powered Test Case Prioritization Optimization in Continuous Integration Systems

João Luís Xavier Barreira Lousada

Thesis to obtain the Master of Science Degree in

Technological Physics Engineering

Supervisor(s): Prof. Dr. Rui Dilão
Eng. Miguel Ribeiro

Examination Committee

Chairperson: Prof. Full Name
Supervisor: Prof. Full Name 1 (or 2)
Member of the Committee: Prof. Full Name 3

October 2020

Dedicated to someone special...

Acknowledgments

A few words about the university, financial support, research advisor, dissertation readers, faculty or other professors, lab mates, other friends and family...

Resumo

Inserir o resumo em Português aqui com o máximo de 250 palavras e acompanhado de 4 a 6 palavras-chave...

Palavras-chave: palavra-chave1, palavra-chave2,...

Abstract

In modern software engineering, Continuous Integration (CI) has become an indispensable step towards managing systematically the life cycles of system development. Large companies struggle with keeping the pipeline updated and operational, due to the large amount of changes and addition of features, that build on top of each other. Associated with such software changes is a strong component of Testing. By selecting the most promising test cases quicker, the round-trip time between making a change and receiving feedback is shortened, boosting productivity. To alleviate this burden, two Test Case Prioritization algorithms were developed.

Extending the research of applying Reinforcement Learning to optimize Testing strategies. After proven to be a strategy as good as traditional methods, we test its ability to adapt to new environments, by using novel data from the financial industry. Additionally, we studied the impact of experimenting a new model for memory representation: Decision Trees, without producing significant improvements relative to Artificial Neural Networks.

NNE-TCP is a Machine-Learning (ML) framework that analyses what files were modified when there was a test status transition and learns relationships between the two entities 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 smarter groupings. By applying NNE-TCP, it is shown for the first time that the modified files and tests connection is relevant and produces fruitful prioritization results.

Keywords: Continuous Integration, Regression Testing, Test Case Prioritization, Reinforcement Learning, Embeddings, Data.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, the motivation behind the work is introduced, as well as the problem and the objectives to be achieved.

1.1 Motivation

Given the complexity of modern software systems, it is increasingly crucial to maintain quality and reliability, in a time-saving and cost-effective manner, specially in large and fast-paced companies. This is why many industries adopt a *Continuous Integration* (CI) strategy, which is a popular software development technique where engineers, frequently, incorporate code changes into the mainline codebase, allowing them to easily check that their code can build successfully and pass tests across various system environments [1] . The latter task performed in the process is called *regression testing*. To ensure that the introduction of new features or the fix of known failures not only are correct, but also do not obstruct existing functionalities. Ergo, regression testing plays a fundamental role on certifying that newer versions adhere to the mainline.

Regression can have many origins, namely code does not compile, performance drops or tests fail, and, as software teams grow, due to the rising amount of changes , the probability that one creates a conflict is significant. Identifying and amending such regressions constitutes one of the most challenging, costly and time-consuming tasks in the software development life-cycle [2].

In the last decades, there has been a high demand for automated techniques that accelerate early fault detection, minimizing human intervention. Regression test prioritization lies at the core of these techniques, as one of the most thriving fields in achieving promising results, with increasing research attention (reference). Aiming to find the optimal permutation of ranked tests that match a certain criteria, i.e. the ability to detect faults a priori, the likelihood a change has of being merge and/or a limited time-frame [3].

The automation process is accomplished through developing algorithms that are, traditionally, programmed for a certain objective, by obeying a set of well-defined instructions. Many traditional algorithms may have shortcomings and not scale well with the complexity of today's systems. However, 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. In particular, there is a trend in capitalizing machine learning (ML) algorithms and data-driven approaches to solve these kind of problems [4] .

Concretely, in a supervised learning task, given a code change and a list of test cases to be prioritized, test cases can be classified as ”pass” or ”fail”, depending on some set of features and use the result to rank failing tests first. In fact, Catal [5] showed that ML approaches can improve the probability of fault detection, when compared to classic software algorithms. Powerful results have been achieved by several authors: Chen et al. [6] use semi-supervised K-Means to cluster test cases and then pick a small subset of tests from each cluster, Uber developers [7] built a speculation engine powered by Logistic regression to predict failed test cases and more recently, Palma et al. [3] combined traditional test and similarity-based quality metrics, and Lachmann et al. [8] resorted to textual information to extract features from test descriptions written in Natural Language.

1.2 Version Control Systems

In software engineering, version control systems are a mean of keeping track of incremental versions of files and documents. Allowing the user to arbitrarily explore and recall the past changes that lead to that specific version. Usually in these kind of systems, changes are uniquely identified, either by a code number or letter, an associated timestamp and the author.[1]

1.2.1 Repository & Working Copy

The central piece of these systems is the **Repository**. - which is the data structure where all the current and historical files are stored and possibly, remotely, accessible to others (*clients*).[1] When a client *writes* information, it becomes accessible to others and when one *reads*, obtains information from the repository. The major difference from an usual server is the capability of remembering every version of the files. With this formulation, a client has the possibility to request any file from any previous version of the system.

While developing, having multiple versions of a project is not very useful, given that most compilers only know how to interpret one code version with a specific file type. So the bridge linking the repository and the user is the **working copy**. - a local copy of a particular version, containing files or directories, on which the user is free to work on and later on communicate the changes to the repository.[1]

1.3 Mono-repositories vs. Multi-repositories

In order to manage and store amount of new code produced daily, there are two possible ways of organizing repositories: one single giant repository that encompasses every project or assigning one repository for each one. Google's strategy is to store billions of lines of code in one single giant repository, rather than having multiple repositories. [9] A mono-repository is defined such that it encompasses the following properties:

- **Centralization** - one code base for every project.
- **Visibility** - Code accessible and searchable by everyone.
- **Synchronization** - Changes are committed to the mainline.
- **Completeness** - Any project in the repository can only be built from dependencies that are also part of it.
- **Standardization** - developers communalize the set of tools and methodologies to interact with code.

This gives, the vast number of developers working at Google, the ability to access a centralized version of the code base - *"one single source of truth"*, foment code sharing and recycling, increase in development velocity, intertwine dependencies between projects and platforms, encourage cross-functionality in teams, broad boundaries regarding code ownership, enhance code visibility and many more. However, giant repositories may imply less autonomy and more compliance to the tools used and the dependencies between projects. Also developers claim to be overwhelmed by its size and complexity.

A multi-repository is one where the code is divided by projects. With the advantages that engineers can have the possibility of choosing, with more flexibility, the tools/methodologies with which they feel more at ease; the inter dependencies are reduced, providing more code stability, possibly accelerating development. However, the major con arises from the lack of necessity of synchronization, causing version problems and inconsistencies, for example two distinct projects that rely on the same library.

Although electing a preferred candidate is still a matter of debate, mono-repositories are linked to less flexibility and autonomy when compared to multi-repositories, but there is the gain of consistency, quality and also the culture of the company is reinforced, by unifying the tools used to link the source code to the engineer.[10]

1.4 Continuous Integration

In this section, the goal is to provide a solid comparison between the different strategies that can be adopted to make sure source-code repositories are a manageable choice.

1.4.1 CI Approaches

Naive-approach

The simplest approach is to run every test for every single commit. As teams get bigger, both the number of commits and the numbers of tests grow linearly, so this solution does not scale. Also, as [16] stated, computer resources grow quadratically with two multiplicative linear factors, making continuous integration a very expensive and not eco-friendly process in terms of resources and time.

For example, Google's source code is of the order of 2 billion lines of code and, on average, 40000 code changes are committed daily to Google's repository, adding to 15 million lines of code, affecting 250000 files every week, having the need to come up with new and innovative strategies to handle the problematic.[2]

Cumulative Approach

One possible solution to contour the large amount of time spent on testing and waiting for results is to accumulate all the commits made during working hours and test the last version, periodically, lets say during the night. This enables teams to develop at a constant speed without having the concern of breaking the master branch, but rather worrying about it in the future. In this case, the lag between making a commit and knowing if it passed or fail is very large, running the risk of stacking multiple mistakes on top of each other.

Using this approach, two possible outcomes can occur: either the build passes every test and the change is integrated, or it fails, causing a regression. Here, we encounter another crossroads, as the batch was tested as a cumulative result, we have no information whatsoever of what commit, exactly, caused the regression. This problem can be solved by applying search algorithms:

- **Binary Search** - instead of running every option, divide the list in half and test the obtained subset, if pass, the mistake is located in the other half, if fail, split the subset and test again until narrowing it down to a specific change, this approach is O(log n) complex.

After finding out which commit caused the regression, it can be fixed or rolled back to the version before the change was introduced, meaning that possibly the progress made during that working day has to be scrapped and re-done, postponing deadlines and, at a human level, lowering down motivation, with developers being ”afraid” of making changes that will propagate and will only be detected later on, so allowing a large turn-around time may results in critical hampering of productivity.

In comparison with naive-approach, cumulative approach is more scalable in the way that it avoids testing every single commit made, only trying to test the last of the day, spending more time trying to trace back the origin of the failure. So in conclusion, we forfeit absolute correctness to gain speed and pragmatism, although this solution's computer resources grow linearly with the number and time of tests.

The key is that we should be able to go beyond finding items in a list were elements are interchangeable and equally informative. We should be able to use that information to do a much more efficient search. For example, we are looking for a faulty commit in a list of 10 elements and we know with a 95% probability that the commit is one of the last 4 elements, then we should only do binary search on those 4, instead of 10, thus saving time.

Change-List Approach

In this case, changes are compiled and built in a serialized manner, each change is atomic and denoted as a Change List (CL). Clarifying, lets take a look at Google's example of workflow in Figure 1.1.

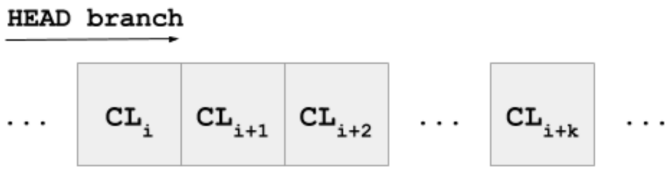


Figure 1.1: Code repository and workflow at Google. Each CL is submitted to master branch or HEAD [2]

Individually, tests validate each CL and if the outcome is pass, the change is incorporated into the master branch, resulting into a new version of the code. Otherwise, a regression is caused. For example, imagine that CL_i introduces a regression, causing failure on the consecutive system versions. Possible fixes and tips: investigate the modifications in CL_i, to obtain clues about the root cause of the regression; Debug the fail tests at CL_i, which is preferable to debug at later version CL_{i+k}, whereas after CL_i code volume might increase or mutate, misleading conclusions; revert or roll back the repository, until the point where it was "green", in this case CL_{i-1}.[2]

The novelty is introduced in test validation:

- **Presubmit tests** - Prior to CL submission, preliminary relevant small/medium tests are run, if all pass the submission proceeds, else it gets rejected. So instead of having to wait until all tests are complete, the developer has a quick estimate of the status of the project.

- **Postsubmit tests** - After CL submission, large/enormous tests are executed and if they indeed cause a regression, we are again in the situation where we have to find exactly what CL broke the build and, possibly, proceed in the same manner as Cumulative Approach to detect what version caused the regression. [2]

Nevertheless there are some major issues: while presubmit testing, the sole focus is to detect flaws on individual changes and not on concurrent one. *Concurrency* is the term referred to builds that run in parallel with each other, meaning that 2 developers can be committing changes of the same file, that are in conflict with each other, or with other files. Therefore, after precommit phase, there is the need to check all the affected dependencies of every commit, quickly identify were the conflict emerged and eventually roll back in the case of a red master, which can only be done in postcommit stage [7] and when a postcommit test fails: a 45 minute test, can potentially take 7.5 hours to pinpoint which CL is faulty, given a 1000 CL search window, so automation techniques, quickly putting them to work and returning to a green branch is pivotal. [2]

To better understand the concept of concurrency and the implications it brings, lets consider there are n concurrent changes in a batch, where all of them pass precommit phase: (1) committing changes 1 to $n-1$ do not lead to breakage, (2) applying the n^{th} change does not break the master, and (3) putting all the changes together breaks the mainline. Which might indicate that the n^{th} change is in conflict with the rest. Concurrency is practically unavoidable and it is more likely to happen when parts of the code that are interconnected are being altered. Frequent mainline synchronization is a possible path to diminish this effect, forcing developers to always work on the most updated version.[7]

Always Green Master Approach

After steering clear of analysing every single change with every single test, or accumulating all the changes and testing them , splitting it into smaller individualized chunks, cherry-picking an ideal and non-redundant subset of tests, prioritizing them by size making it possible to divide test phase into precommit and postcommit, and enabling a status estimate of the likelihood of regressions, [7] proposes that an always green mainline is the priority condition, so that productivity is not hampered. To guarantee an always green master, Uber designed a scalable tool called *SubmitQueue*, where every change is enqueued, tested and, only later on, integrated in the mainline branch if it is safe. The interesting aspect is which changes get tested first, while ensuring serializability. Uber resorts to a probabilistic model that can speculate on the likelihood of a given change to be integrated. - i.e. pass all build steps. Additionally, to agilize this process this service contains a *Conflict Analyzer* to trim down concurrent dependencies.

Probabilistic Speculation Based on a probabilistic model, powered by logistic regression, it is possible to have an estimate on the outcome a given change will have and with that information select the ones that are more likely to succeed, although other criteria might be chosen.

To select the more likely to succeed changes, this mechanism builds a *speculation tree*, which is a binary decision tree. Imagine a situation where two changes, C_1 and C_2, need to be integrated in the master branch M. C_1 is merged into M is denoted by $M \oplus C_1$ and the build steps for that change are denoted by B_1. B_{12} represents builds for $M \oplus C_1 \oplus C_2$. Speculating that B_1 has high probability of being integrated, it is useful to test it in parallel with B_{12}, because if the assumption is correct, C_1 is integrated and now this result can be used to determine if C_2 also passes, without testing each one individually and then together. If the speculation is not correct, C_1 is not merged and C_2 has to build separately by running B_2. In the case of three pending changes C_1,C_2 and C_3, the speculation tree is of the form of Figure 1.2.

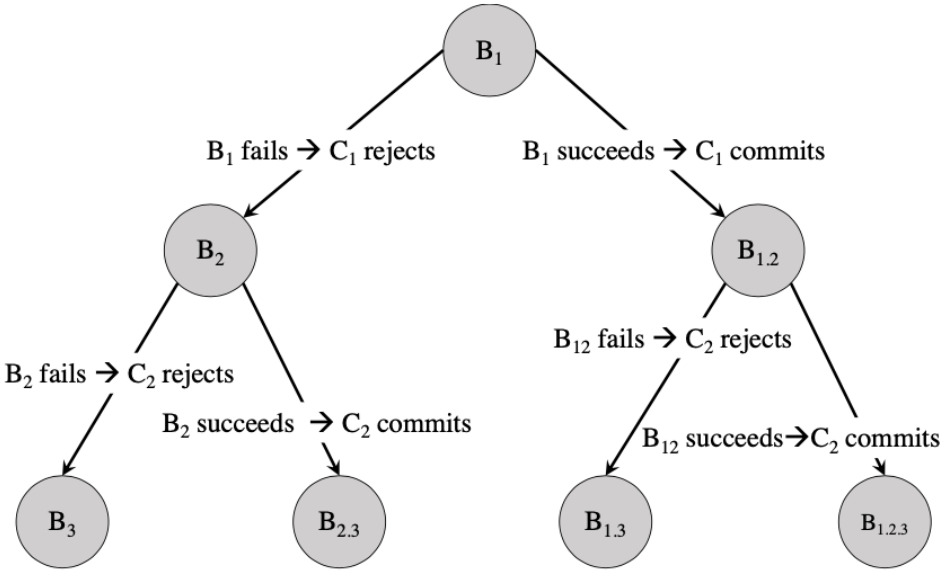


Figure 1.2: Speculation tree - builds and outcomes [7]

One possible way of handling the result from this tree would be to speculate everything, assuming that the probability of merging a change is equal to the probability of rejecting it. Having 3 pending changes, one would have to perform 7 builds (number of nodes in the tree), under this assumption. But it is unnecessary to speculate all of the builds, if B_1 is successful, B_2 now becomes irrelevant and the tree is trimmed down considerably. Only n out of $2^n - 1$ builds are needed to commit n pending changes [7].

Additionally, the assumption that every change depends on one another is made, but this is not always the case, two changes can be totally distinct. By detecting independent changes, a lot of time can be saved by building them individually and committing them in parallel. This is done by the Conflict Analyzer. This approach combining the selection of most likely to pass builds, using machine learning models, parallel execution, the identification of independent changes, trimming down speculation tree size is able to scale thousands of daily commits to a monolithic repository. Finally, a comparison of all explored approaches is shown in the Table below.

Approach	Correctness	Speed	Scales?
Naive	Very High	Very Low	No
Cumulative	Medium	Low	No
Change-List	Medium	Medium	Yes
Always Green Master	High	High	Yes

Table 1.1: Comparative analysis between strategies

1.5 Software Testing

In testing, associated with each repository, there is a battery of tests that makes sure it is safe to integrate a change in the system, but what if we are in the case of applying the same tests twice or more ? What if a test is not adequate to the type of change a developer did? What are the boundaries of test quality? If a test always passes, is it a good test ? Should one apply tests in an orderly way, prioritizing some tests by time or relevance criteria ?

Testing is a verification method to assess the quality of a given software. Although, this sentence is quite valid, it is vague, in the sense that it does not define what quality software actually is. In some contexts, quality might just refer to simply code compiling with no syntax errors or "mistakes". When tests are applied, the outcome obtained is of the form PASS/FAIL, with the purpose of verifying functionality or detecting errors, receiving quick and easy to interpret feedback. However, the connections between testing and quality are thin: testing is very much like sticking pins into a doll, to cover its whole surface a lot of tests are needed. For example, running a battery of tests (test suite) where every single one of them yields PASS. This can only mean two things: whether the code is immaculate or some scenarios were left out of the process. Usually, test suites are constructed from failed tests. When in a FAIL situation, i.e. fault detection or removal , a new test case is created, preventing this type of error to slip again in the future, incrementing the existing test suite, in a "never ending" process).

Hence it is correct to say, failed tests are a measure of non-quality, meaning there is no recipe for assuring a software is delivered without flaws. [11]

1.5.1 Regression Testing

Regression testing is performed between two different versions of software in order to provide confidence that the newly introduced features of the working copy do not conflict with the existing features. In a nutshell, whenever new features are added to an existing software system, not only the new features should be tested, but also the existing ones should be tested to ensure that their behaviours were not affected by the modifications. Usually this is done by applying test cases, to check if those features, in fact work or are still working. Therefore, this field encapsulates the task of managing a pool of tests, that are repeatedly applied across multiple platforms. [12]

The problem relies on the fact that tests do not run instantaneously and as software systems become more complex, test pools are ought to grow larger, increasing the cost of regression testing, to a point where it becomes infeasible, due to an elevated consumption of computer and time resources, so there is a high demand to search for automated heuristics that reduce this cost, improving regression testing. In this work, three solutions are proposed that can lead to possible substantial performance improvements:

- **Test Case Selection** - "do smarter" approach, selecting only relevant tests, given the type of change.
- **Test Suite Minimisation** - "do fewer" approach, removing possible redundancies.
- **Test Case Prioritisation** - also "do smarter" approach by running some tests first, increasing probability of early detection. [12]

In terms of notation, let us denote P as the current version of the program under test, P' as the next version of P. T is the test suite and individual tests are denoted by a lower case letter: t. Finally, $P(t)$ is the execution of test t in the system version P.

Test suite minimisation

Definition 1.5.1. Given a test suite T, a set of test requirements $R = r_1, ..., r_n$ that must be satisfied to yield the desired "adequate" testing, and subsets of T , $T_1, ..., T_n$, each one associated with the set of requirements, such that any test case t_j belonging to T_i can be used to achieve requirement r_i

The goal is to try to find a subset T' of T: $T' \subseteq T$, that satisfies all testing requirements in R. A requirement, r_i is attained by any test case, t_j belonging to T_i. So a possible solution might be the union of test cases t_j in T_i's that satisfy each r_i (*hitting set*). Moreover, the hitting set can be minimised, to avoid redundancies, this becoming a "minimal set cover problem" or, equivalently, "hitting set problem",

that can be represented as a *Bipartite graph*[1]. The minimal hitting-set problem is a NP-complete problem (whose solutions can be verified in polynomial time). [13]

Another aspect to consider is that test suite minimisation is not a static process, it is temporary and it has to be *modification-aware*. - given the type of modification and version of the code, the hitting set is minimised again. [12]

Test Case Selection

Definition 1.5.2. Given a program P, the version of P that suffered a modification, P', and a test suite T, find a subset of T, named T' with which to test P'.

Ideally, the choice of T', should contain all the *fault-revealing* test cases in T, which are obtained by unveiling the modifications made from P to P'. Formally:

Definition 1.5.3. Modification-revealing test case A test case t is said to be modification-revealing for P and P' if and only if the output of $P(t) \neq P'(t)$. [14]

After finding the nature of the modifications, in simple terms, one has a starting point to select the more appropriate test cases. Both test case minimisation and selection rely on choosing an appropriate subset from the test suite, however they do so, often, with different criteria: in the first case the primal intent is to apply the minimal amount of tests without compromising code coverage in a single version, eliminating permanently the unnecessary tests from the test suite, as in the second the concern is related directly to the changes made from one previous version to the current one. [12]

Test case prioritisation

Definition 1.5.4. Given a test suite 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 $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$

Where f is a real function that evaluates the obtained subset in terms of a selected criteria: code coverage, early or late fault detection, etc [12]. For example, having five test cases (A-B-C-D-E), that detect a total of 10 faults, there are $5! = 120$ possible ordering, one possible way of choosing a permutation, is to compute the metric Average Percentage of Fault Detection (APFD) [15].

Definition 1.5.5. APFD Let T be a test suite containing n test cases and F the set of m faults revealed by T. Let TF_i be the order of the first test case that reveals the i^{th} fault.[12]

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

Simply, higher values of APFD, imply high fault detection rates, by running few tests, as can be seen in the following plot:

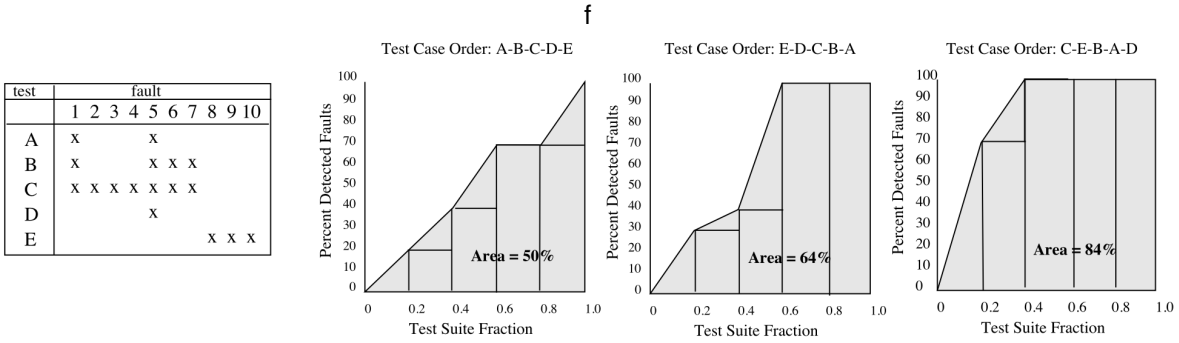


Figure 1.3: Average percentage of fault detection [15]

The area below the plotted line is interpreted as the percentage of detected faults against the number of executed test cases, maximizing APFD, demonstrating clear results in detecting early applying fewer

[1]s a graph whose vertices can be divided into two disjoint and independent sets U and V, such that every edge connects a vertex in U to one in V[13]

test cases. The order of test cases that maximizes APFD, with $APFD = 84\%$ is **C-E-B-A-D**, being preferable over other permutations.

By detecting faults early, the risk of developing over faulty code is reduced, causing minimal breakage in productivity avoiding, for example, possible delayed feature releases.[7]

1.6 Machine Learning

Machine Learning algorithms are at the forefront in achieving effective automatic regression testing. By building predictive models, it is possible to accurately, and most importantly, quickly, identify if a newly introduced change is defective or not. This helps developers to check and amend faults, when the details still remain fresh on their minds and it allows to prioritize and organize changes by the risk of being defective, this way testing the ones that are more likely to fail first, diminishing the lag-time between committing and feedback.

In recent years, several researchers studied the usefulness of applying fault-prediction techniques , powered by machine learning algorithms. More concretely, resorting to an emergent area which is Deep Learning. Kamei et. al propose an approach called *Deeper*, that is divided into two phases: the feature extraction phase and the machine learning phase. The first consist on identifying a combination of relevant features from the initial dataset, by using Deep Belief Networks. The latter is encapsulated on building a classifier/predictive model based on those same features.

In this work, the author aims to replicate and explore the techniques developed by ... and ..., extending the domain of validity and achieving higher levels of accuracy, by experimenting other classifiers and feature extraction techniques.

In the next sections, a theoretical background will be elaborated on the several techniques used throughout the thesis, to provide a solid base for moving forward.

1.7 State of the Art

1.7.1 Baseline Approaches

1.7.2 Code-Coverage Approaches

1.7.3 History-based Approaches

1.7.4 Modification-based Approaches

1.8 Test Case Prioritization in Industrial Environments

What is happening in the current CI system ? numbers of commits per day, number of resources used, amount of data generated, time it takes on average to integrate changes and current ”naive approach”

How is the company expected to grow in the near future ? as teams and projects grow time and resources grow quadratically.

Consequences of increasing turn around time: context-loss, productivity hampering, delays. CI aims to spend less time integrating different developers’ code changes.

trade off speed vs. correctness

1.9 Repository

For transparency and replication purposes, all the files developed and used throughout this thesis are stored and available to access at `https://github.com/johnnylousas/Master-Thesis`. The online repository contains all open-source datasets, Python files implemented to apply machine learning and collect results, along with pertinent documentation.

1.10 Objectives & Thesis Outline

Th goal of this thesis is to use Machine Learning pipelines to forecast test case failure likelihood. Datasets were algorithms are trained: dummy data (controlled environment) , real-data of BNP (real-world case), OpenSource Data (other datasets available to compare results). Then for a given revision, choose which chain of test cases minimize pass/fail uncertainty, reducing fault detection time. Finally, provide a live-estimate of the status of a Project in real time. (farfetched)

The objectives delineated for this work are:

- Detect common usage patterns, in a controlled environment, by generating synthetic data.
  - Learn heuristics to automate fault detection process
- Optimize regression testing systems using real world data.
  - Analyse how different system configurations affect Continuous Integration
  - Reduce fault detection time
  - Provide a Live-Estimate of the Status of a Project
  - Given a commit, choose which chain of tests minimize pass/fail uncertainty

The thesis is organized as follows:
Chapter 2 Background
Chapter 3 Paper 1
Chapter 4 Paper 2
Chapter 5 Conclusions

Chapter 2

History-based Reinforcement Learning

This section provides the necessary formalism for the Test Case Prioritization Optimization problem .

2.1 Background

Reinforcement Learning (RL) is an adaptive method where an agent learns how to interact with an environment, that responds with reward signals that correspond to the feedback of taking a certain action. These back-and-forth interactions take place continuously: the agent receives some representation of the environment's state and selects actions either from a learned policy or by random exploration. Consequently, the environment responds to them and presents new situations to the agent, finding itself in a new state. The main goal is to maximize the cumulative sum of rewards, rather than just immediate ones [29]. More formally, considering a set of discrete time steps, $t = 0, 1, 2...$, the representation of the environment's state is defined as $S_t \in S$, where S is the set of 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 in 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.

2.1.1 Formalism

2.1.2 Notation

The test pool is defined as T and it is composed by the set of test cases $\{t_1, t_2, ..., t_N\}$. For each commit C_i, a test suite T_i can be selected and ordered to be executed, such that $T_i \subset T$. Usually, subsets of T are scheduled when there is a time or resource restriction, otherwise the ordered sequence of all test cases $T_i^{all} = T$ is applied. Note that T does not have an ordering, while both T_i and T_i^{all} are meant to be ordered sequences. Therefore, the definition of a ranking function that acts over all test cases should be defined: $rank : T_i \rightarrow N$, where $rank(t)$ is the index of test case t in T_i. Each test case t contains information about the duration $t.duration_i$, which is known before execution, and the test status $test.status_i$, only known after execution and it is equal to 1 if the test has passed, or 0 if it has failed. In T_i, the subset of all failed test cases is denoted $T_i^{fail} = \{t \in T_i \; s.t. \; t.status_i = 0\}$.

Note that in real world scenarios, there are situations where the cause of failure of a test case is due to exogenous reasons such as stochasticity or hardware problems, so tests can have multiple status at C_i or in other words, have *flakyness*. For simplicity, if a test is executed more than once in C_i, the last execution status is kept. Also it is worth distinguishing between failed test cases and faults: a failure of a test case can be originated due to one or several faults in the system under test, and oppositely a single fault can cause several test cases to fail. In this study, because there is no information available about the actual faults, solely the status of each test case.

Lastly, to evaluate the performance on estimating the status of a test case in a commit, we define $p_i(t)$. The overall performance P_i of a test suite T_i can be formulated with any cumulative function over $p_i(t_1), ..., p_i(t_N)$. Concretely, it can be the average of each individual prediction: $\frac{1}{|T_i|} \sum_{t \in T_i} p(t)$

2.1.3 Problem Formulation

The end-goal of Test Case Prioritization is to re-arrange test cases according to a given criteria. Formally speaking, given a test suite T_i, the set containing all possible permutations of T_i, PT, and a function from PT to real numbers $f : PT \rightarrow \mathbb{R}$, TCP aimds to find a subset T' such that $(\forall T_i \in PT : P_i(T_i') \geq P_i(T_i)$. Where f is a real function that evaluates the obtained subset in terms of a selected criteria: code coverage, early or late fault detection, etc [12]. For example, having five test cases (A-B-C-D-E), that detect a total of 10 faults, there are $5! = 120$ possible permutations.

This formulation does not comprehend the constraint of having a time limit for test execution nor takes into account the history of test executions. To incorporate this information, we need to solve the *Adaptive Test Case Selection Problem* (ATCS). Considering a sequence $T_1, ..., T_{i-1}$ of previously executed test suites, then the *ACTS's* goal is to choose a test suite T_i, in order to maximize $P_i(T_i)$ and given a time constraint where $\sum_{t \in T_i} t.duration \leq M$, where M is the time allocated to run T_i.

2.1.4 Machine Learning

The path to automatically solving a problem entails producing sets of instructions, that turn an input into a desired output, namely, algorithms. Nevertheless, not all problems can be solved by traditional algorithms, due to limited or incomplete information. In our case, we don't know which tests are more likely to uncover failures. It could be the case, that there is not even a single failure. Also a previously failing test case is not an indicator that it will fail again, in the exact same context. Hence, with the rise of data availability, there has been a growing interest to investigate solutions that involve learning from data [4].

Benjamin Busjaeger et al. [24] proposed an optimization mechanism for ranking test cases by their likelihood of revealing failures, in an industrial environment. The ranking model is fed with four features: code coverage, textual similarity, failure history and test age. These features were the used to train a Support Vector Machine Model, resulting in a score function that sorts test cases. More recently, Palma et al. [3] similarly trained a Logistic Regression model, based on textual test information, for example, the number of methods called or how different two test cases are from each other. Liang et al. [25] proved that commit-level prioritization, instead of test-level would enhance fault detection rates, on fast-paced software development environments.

Another way of achieving effective prioritizating is using Semi-Supervised Learning approaches, like Clustering algorithms. Shin Yoo et al. [26] and Chen et al. [6] endorse coverage-based techniques, claiming that making fast pair-wise comparisons between test cases and grouping them in clusters allows for humans to pick, more straight-forwardly, relevant non-redundant test cases. The assumption being that test cases belonging to the same cluster will have similar behaviour.

Recently, Spieker et al. [27] were the first to implement a Reinforcement Learning approach to TCP, introducing RETECS. The method prioritizes test cases based on their historical execution results and duration. RETECS has the perk of being an adaptive method in a dynamic environment without compromising speed and efficiency. Applied to three different industrial datasets, according to Spieker et al. [27], RETECS challenges other existing methods and has caught research attention from other researchers, namely Wu et al. [28]. This work is an extension of the research conducted by the authors described above.

2.2 RETECS

This section describes the algorithm used to solve the ATCS problem, based on the approach developed by Spieker et al. [27] called *Reinforced Test Case Selection* (RETECS).

2.2.1 Reinforcement Learning for Test Case Prioritization

In the context of TCP, RETECS prioritizes each test case individually and afterwards a test schedule is created, executed and, finally, evaluated. Each state represents a single test case $t_i \in T_i$ - containing it's duration, the time it was last executed and the previous execution results - and the set of possible actions is the set of all possible prioritizations a given test case can have in a commit. 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: positive rewards reinforce, whereas negative rewards discourage the current behaviour. The agent-environment interface is depicted in Figure 2.1.

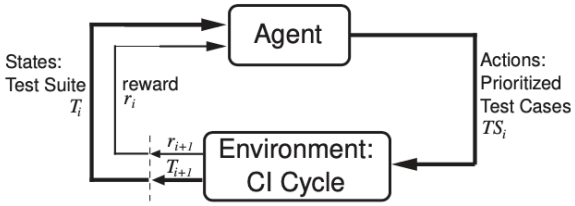


Figure 2.1: Reinforcement Learning applied to TCP cycle of Agent-Environment interactions (adapted from [29])

The RETECS framework has the following characteristics:

1. **Model-Free** - has no initial concept of the environment's dynamics and how it's action will affect it.
2. **Online** - learns *on-the-fly*, constantly during its runtime. It is particularly relevant in environments where test failure indicators change over time, so it is adamant to update the agent's strategy.

2.2.2 Reward functions

In a RL problem, the formalization of goals passes by collecting numerical rewards that measure the performance of the agent at a given task. Hence, properly defining a reward function that reflects these goals will steer the agent strategy towards optimality. Within TCP, the goal is to prioritize test cases that will detect faults as early as possible, to minimize the feedback-loop. So we define 3 reward functions: Failure Count, Test-Case Failure and Time-Ranked.

Failure-Count Reward

$$reward_i^{fail}(t) = |T_i^{fail}| \quad (\forall_t \in T_i) \tag{2.1}$$

The first reward awards all test cases with the number of failed test cases that were executed, therefore the RL agent is directly encouraged to maximize the number of failed test cases. This simple approach considers the test schedule as a whole, by only looking at the number of failures. However by not taking into account which specific test-case was responsible for detecting them, the risk of favouring undesired behavior, such as always including passing test-cases amongst the ones that are failing, or assigning low priorities to test cases that would fail if executed.

Test-Case Failure Reward

$$reward_i^{tcfail}(t) = \begin{cases} 1 - t.status_i, & \text{if } t \in T_i \\ 0, & \text{otherwise} \end{cases} \tag{2.2}$$

The second reward function bridges the gaps identified with Failure Count reward in terms of refinement, by rewarding each test case individually, based on it's status obtained after execution. This approach agrees with the strategy of executing failing test cases and therefore, this action is reinforced, prioritizing them higher. On the other hand, if a passing test case is scheduled, the reward value is equal to zero, not reinforcing nor discouraging its inclusion on the test schedule. Although test-case failure reward is specific to each test-case, it does not explicitly take into account the order in which they are applied.

Time-Ranked Reward

$$reward_i^{time}(t) = |T_i^{fail}| - t.status_i \times \sum_{\substack{t_k \in T_i^{fail} \wedge \\ rank(t) < rank(t_k)}} 1 \tag{2.3}$$

The last reward function deals exactly with including the order of test cases and not only considers it's status, but also it's rank on the schedule. An ideal test schedule ranks every failing test case at the beginning and the passing ones thereafter. This reward explicitly penalizes passing test cases by the number of failing ones ranked after it. While for failing test cases, time-ranked reward reduces to the failure count reward.

2.2.3 Action Selection

As mentioned earlier, actions can be chosen by relying on a learned policy or by random exploration. The policy is a function that maps states, i.e. test cases, to actions, i.e. a prioritization. For a state $s \in S$ and an action $a \in A(s)$, the policy π yields the probability $\pi(a|s)$ of taking action a when in state s. Therefore, the policy function π is arbitrarily close to the optimal policy. In the beginning, high exploration is encouraged to explore the unknown environment and adjust by *trial-and-error*, whereas in a later phase, as learned policies become more reliable, the exploration rate is reduced. However, it is not annulled. The exploration rate can be tuned depending on how dynamic the environment is. This type of algorithm is denoted as $\epsilon - greedy$. The degree of exploration is governed by the parameter ϵ and actions are picked from the learned policy with $(1 - \epsilon)$ probability.

2.2.4 Memory Representation - Value Functions

Commonly in RL algorithms it is useful to know how good it is for an agent to perform a given action in a given state, by estimating what will be the rewards to receive in the future [29]. These estimates are calculated by defining a *value-function* $v_\pi(s)$ with respect to the learned policy π and can be learned from experience. For example, if an agent follows a policy π and for each state encountered an average of the obtained rewards is mantained, after a while, the average will converge to the state's actual value $v_\pi(s)$ as the number of encounters reaches infinity. However, with an increasing number of states it may not be practical to keep record of each state individually, instead $v_\pi(s)$ should be a parameterized function, whose parameters should be adjusted with experience in order to match observations. These type of parameterized functions are called *approximators* and are used to store the returns of observed states and generalize to unseen ones. Generalization reduces the amount of memory needed to store and time needed to update information about every single state, even though these values correspond to estimations rather than observed values.

The topic of function approximation is an instance of *supervised learning*, because it gathers examples and attempts to optimize parameters to construct an approximation of the entire function. [29]. A valid example for such function is the Artificial Neural Network (ANN), where the parameters to be adjusted are the network's weigths. The downside of ANN's is that a more complex configuration is implied to achieve highest performance.

Alternatively, Figure 2.2 shows how a Decision Tree can be used to map an input vector, i.e. a state, to one of the leaf nodes that points to a specific region in the state space. Then the RL agent is able to learn the values of taking each path/actions and where they will lead. [30]

2.3 Experimental Setup

The next section presents the application and evaluation of RETECS, describing, first, the setup procedures as well as a description of the datasets used (section 2.3.1. Then an overview of possible evaluation metrics to assess the framework's performance is provided 2.3.2. To maximize these metrics, fine-tuning is used to find the best combination of parameters and it is shown in section 2.3.3. Finally, in section 2.3.4, results obtained are presented and discussed, according to the research question formulated below. Additionally, threads and future work are discussed to close the evaluation process.

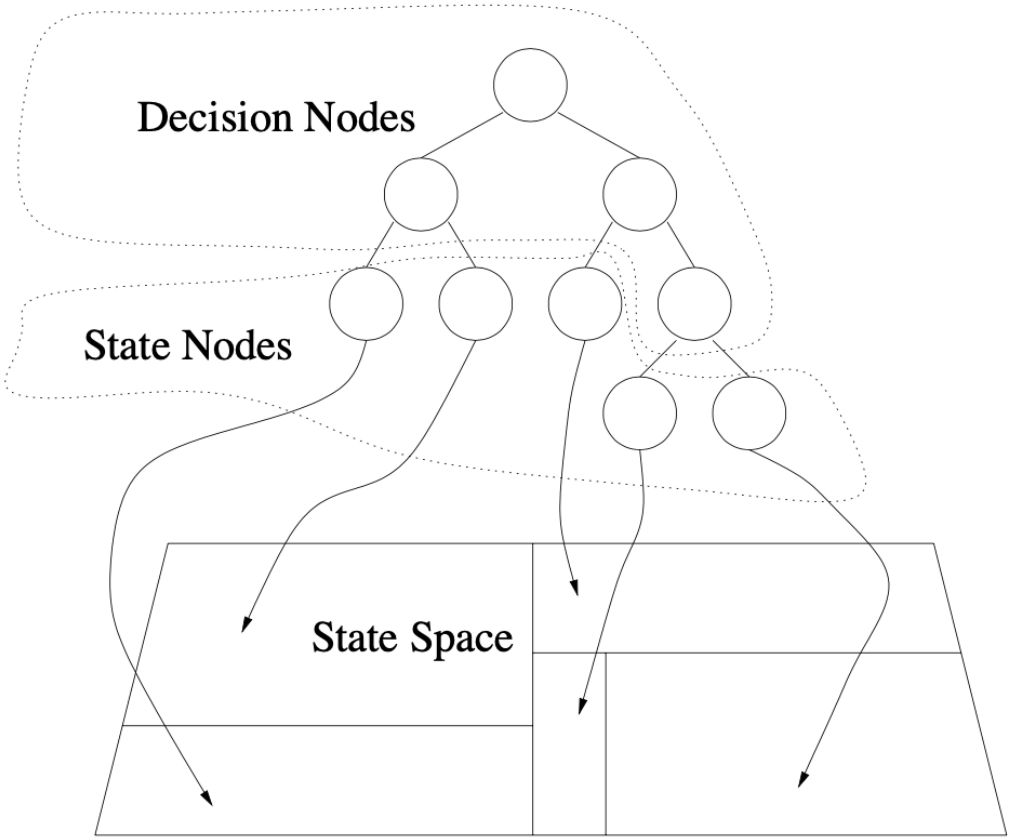


Figure 2.2: Represent state space regions with Decision Trees (Adapted from [30])

RQ1: How will RETECS behave in the presence of a novel dataset with different characteristics ? With which Machine Learning Algorithm works best as an approximator function ? We compare the RETECS performance against two different models: Artificial Neural Networks and Decision Trees.

RQ2: How is RETECS model performance compared to traditional prioritization techniques, in the new context?

2.3.1 Data Description

Data used in this work corresponds to industrial real-world environments, from ABB Robotics Norway [1], *Paint Control* and *IOF/ROL* that test complex industrial robots and *BNP* data, corresponding to the financial sector. Each dataset contains historical information of test results, around 300 commits, and have different characteristics. Table 2.1 summarizes main statistics about the datasets.

Data Set	Test Cases	Commits	Test Executions	Failed
IOF/ROL	2,086	320	30,319	28.43 %
Paint Control	114	312	25,594	19.36 %
BNP	1,379	303	417,837	63.87 %

Table 2.1: Dataset Statistics

The datasets are alike in number of commits, but it is clear that the testing strategy is constrastive. The IOF/ROL dataset contains the least amount of test executions facing the number of test cases it has on the system, meaning that the strategy is much more focused on test case selection. As for the

[1] Website: http://new.abb.com/products/robotics

BNP dataset, the number of test executions is equal to the number of test cases times the number of commits. So there is no test case selection and every test is applied on each commit and that is why the rate of failed test is higher relative to the other two datasets.

2.3.2 Evaluation Metric

The common metric to evaluate the framework's performance is the NAPFD (Normalized Average Percentage of Fault Detection), as defined by Spieker et al. [27] and it represents the most recurrent metric to assess the effectiveness of test-case prioritization techniques. Usually the metric appears in the unnormalized form (APFD) where there is no test case selection. In this case, it includes the ratio between found and possible failures within the test suite.

$$NAPFD(T_i) = p - \frac{\sum_{t \in T_i^{fail}} rank(t)}{|T_i^{fail}| \times |T_i|} + \frac{p}{2 \times |T_i|}$$
$$\text{with } p = \frac{|T_i^{fail}|}{|T_i^{totalfail}|}$$

When $p = 1$, all possible faults will be detected and $NAPFD$ reduces to its original formulation, APFD. The higher it's value, the higher the quality of the test schedule. If equal to 1, all relevant test are applied in order, in the beginning, and if it equal to 0, every relevant test is applied at the end of the schedule.

2.3.3 Fine-Tuning

Parameter tuning allows to find the best combination of parameters that maximize the performance of the RL agent, while providing necessary flexibility to adapt to different environments. For the IOF/ROL and PaintControl datasets the same configuration as Spieker et al. [27] was used to replicate the same results, for the network agent.

ANN Tune: For the BNP dataset the architecture of the hidden layer for the Network Agent was studied, by calculating the NAPFD with different configurations, like depicted below.

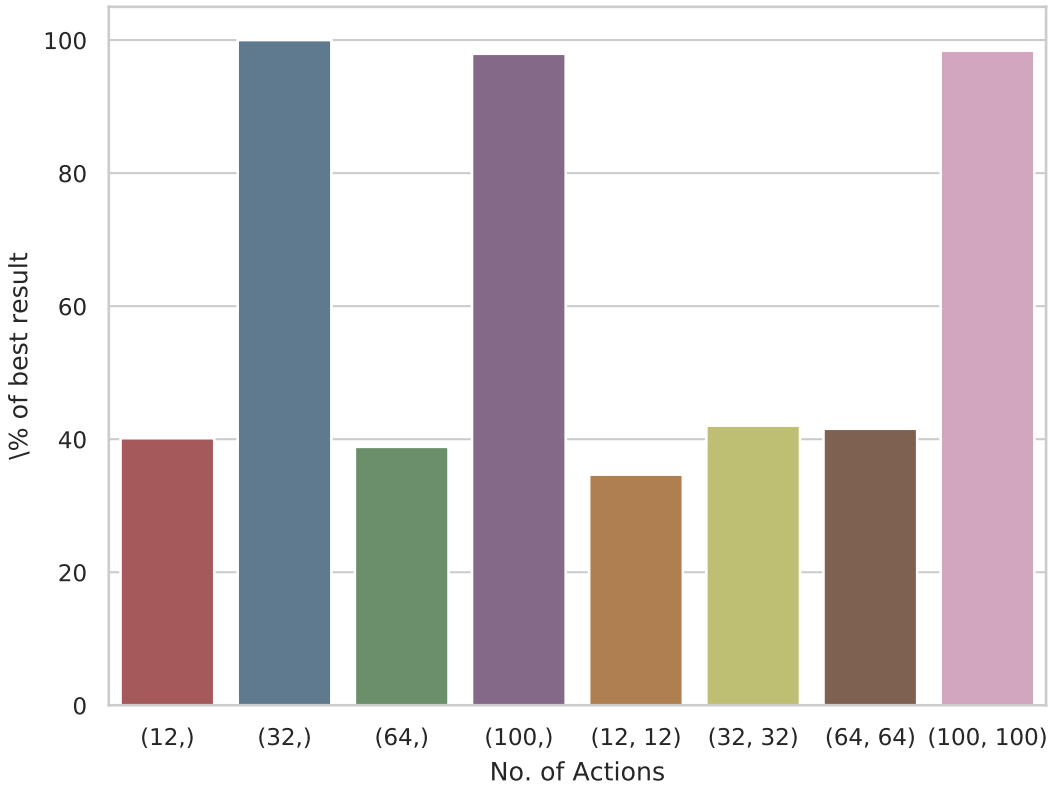


Figure 2.3: ANN Approximator - Hidden Layer architecture

The default value used for the other datasets is 12 nodes with one hidden layer. In Figure 2.3, the configurations that maximizes the metric are: 1 layer with 32 nodes; 1 layer with 100 nodes and 2 layer with 100 nodes each. Since the performances are similar, the simplest architecture is chosen: 1 layer with 32 nodes.

Decision Tree Tune: The parameters to be tuned in Decision Trees are: (1) *criterion*, (2) *maximum depth* and (3) the *minimum samples*. (1) measures the quality of a split, where the options are *gini* for the Gini impurity or *entropy* for the information gain; (2) is the distance between the root node and the leaf node, if depth is infinite the nodes are expanded until all are pure, and (3) is the number needed to split a node in the tree. The variation of these parameters was studied by running a grid search and evaluating the performance for the BNP dataset.

From Figure 2.4, it is clear that there is no significant difference in performance by varying the criterion. Also there is no clear correlation between better performance and the number of minimum samples, which can be due to the reduced number of combinations and range of values. As for the depth, there is a slight increase in performance as the maximum depth increases. Still the best combination of parameter being: gini criterion, a maximum depth of 20 and the minimum number of samples to split is 3. The configuration is maintained throughout the following experiments.

History-Length determines how long the execution history should be. A short history-length may not suffice to empower the agent to make meaningful future predictions, although the most recent results are most likely the more relevant. A larger history-length may encapsulate more failures and provide more fruitful information. However, having a larger history increases the state space and therefore the complexity of the problem, taking longer to converge into an optimal strategy. Moreover, the oldest history record has the same weight as the most recent. Hence, there is no guarantee that a longer execution history will lead to a performance boost. Figure 2.5 studies how different history length values affect the RL agent, with BNP data. It is noticeable that there is no direct relationship between the two

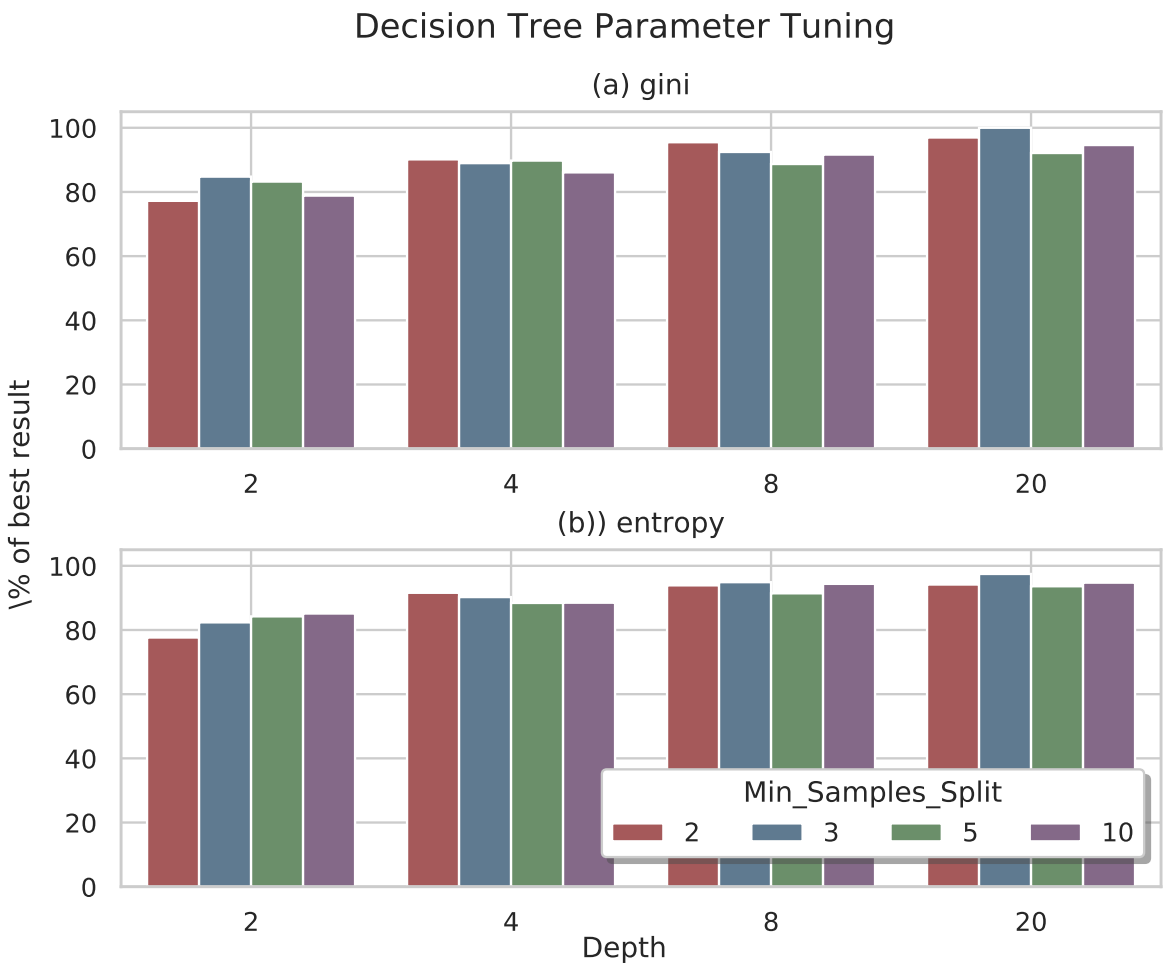


Figure 2.4: DT Approximator - Parameter Tuning

quantities depicted in Fig 2.5. The best result obtained is a history-length of 25 executions, which is disparate from the optimal history lenght obtained by Spieker et al. [27] for the other two datasets, which as 4. This reinforces the fact that BNP data has dissimilar characteristics.

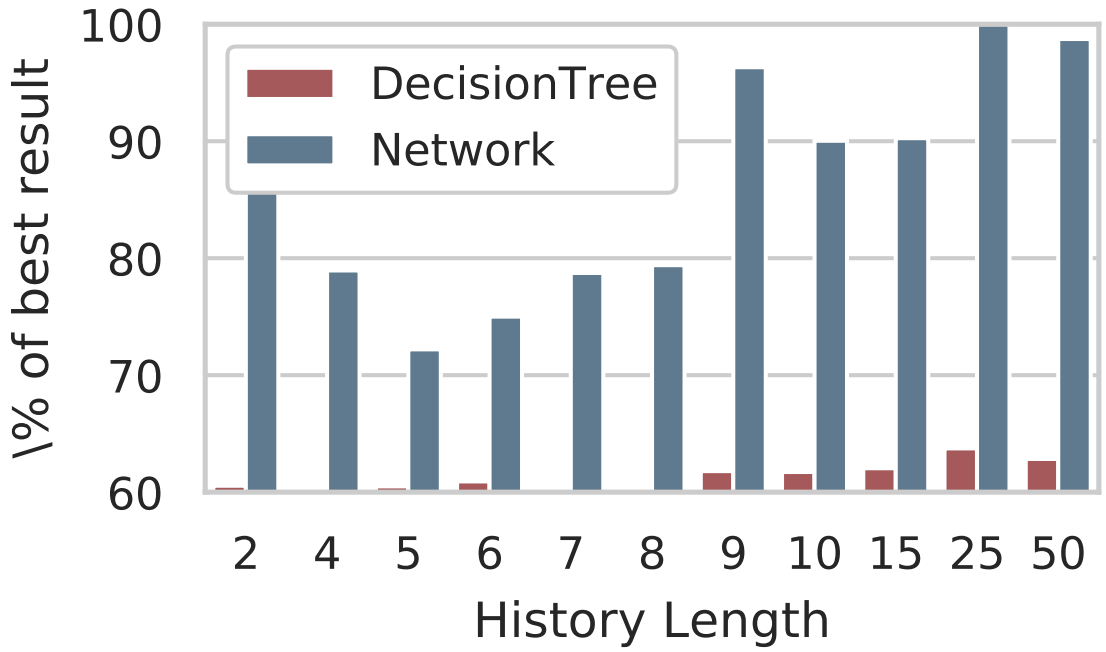


Figure 2.5: History Length

2.3.4 Results

The experiments are calculated for two RL agents. The first resorts to an ANN representation of states, while the second uses a Decision Tree. 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. Traditional prioritization methods were included as a mean of comparison: *Random, Sorting and Weighting*.

The first consists on assigning random prioritization to each test case, to form the baseline method. The other two methods are deterministic. The Sorting method sorts each test case according to their most recent status, i.e. if a test case failed recently it has high priority. The third method is a naive version of RETECS without adaptation, because it considers the same information - duration, last run and execution history - but as a weigthed sum with equal weights.

Due to the fact that RETECS has online learning properties, the evaluation metric NAPFD is measured on each commit and due to the exploratory nature of the algorithm, randomness is taken into account by iterating through the experiments 30 times. If not stated otherwise, reported results show the mean value of all iterations.

For reproducibility and feedback purposes, the upgraded version of RETECS is implemented here [2], in Python using scikit-learns toolbox for ANN's and Decision Trees.

2.3.5 RQ1

Figure 2.6 shows a comparison of the prioritization performance between the ANN Agent and the Decision Tree Agent, with regards to different reward functions (rows), applied to three different datasets

[2]https://github.com/jlousada315/RETECS

(columns). On the x-axis, the commit identifier is represented and for each one there is a correspondent NAPFD value, ranging from 0 to 1. (represented as a line plot in red and blue). The straight lines show the overall trend of each configuration, which is obtained by fitting a linear function - full line for Network and dashed line for Decision Tree Approximator.

It is noticeable that both the approximator used for memory representation and the choice of the reward function have deep impact on the agent's ability to learn better prioritization strategies. Generally, both approximators go hand in hand and present similar trends, i.e., for a given dataset and reward function, both decrease or increase in the same amount (more pronounced when using Failure Count and Time-Ranked rewards).

However, the behaviour observed above no longer holds when looking at the Test Case Failure reward function. It is evidently, the function that produces the best results, in terms of maximizing the slope of the NAPFD trend. When combined with the Network Approximator, this approach reveals the best configuration overall, for the three datasets. Implying that attributing specific feedback to all test-cases individually enables the agent to learn how to effectively prioritize them and adapt to heterogeneous environments.

Another aspect reinforcing the notion of heterogeneity is the differences in the fluctuations of each dataset. For the first two datasets, we see evidently fluctuations in the results. Spieker et al. [27] correlates them with the presence of noise in the original dataset, that may have occurred for numerous reasons and are hard to predict, such as test that were added manually and produced cascade down failures. Notwithstanding, this behaviour is not observed for the BNP data, which suggests a much more stable system with less fluctuations, so there is a stronger indicator that, with the right set of features, a crystal-clear relation between test case and the likehood they have of failing can be learned.

In conclusion, the supremacy of the Network Approximator remains valid for the reward function that produces the best results. Yet, in some cases, the Decision Tree Approximator was able to surpass it's performance, by a small amount. 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 more and more rely on the health and proper functioning of these systems.

2.3.6 RQ2

The focus of RQ1 was to discover what combinations of components would maximize performance: Test case Failure Reward and the Artificial Neural Network Approximator. Now, with RQ2, the aim is to compare this approach to traditional test case prioritization methods: *Random*, *Sorting* and *Weighting*. The results are depicted in Figure 2.7 as the difference between the NAPFD for the comparison methods and RETECS. Each bar comprises 30 commits. For positive differences, the comparison methods have better performance, on the contrary negative differences show the opposite. Due to the exploratory character of the algorithm, it is expected that at the beginning, the comparison method will make more meaningful prioritizations and this trend is verified in all datasets, although more evidently in Paint Control and BNP.

For Paint Control there are 2 adaptation phases: there is a steep convergence in the early commits, only needing around 60 commits to perform as good or better than the comparison methods. Then for the next 90 commits, RETECS performance was progressively worse indicating lack of adaptation and then for the remaining commits, the performances of Sorting and Weighting both match RETECS's and are better than Random.

For IOF/ROL, it is evident that the results were inferior to Paint Control, having small increments on performance as was expected from analysing Figure 2.6. Also the performance of the comparison methods shows the disability to prioritize failing test cases prematurely.

For BNP data, there is clearly a learning pattern with an adaptation phase of around 90 commits needed to have similar performances to the comparison methods and a significant improvement with re-

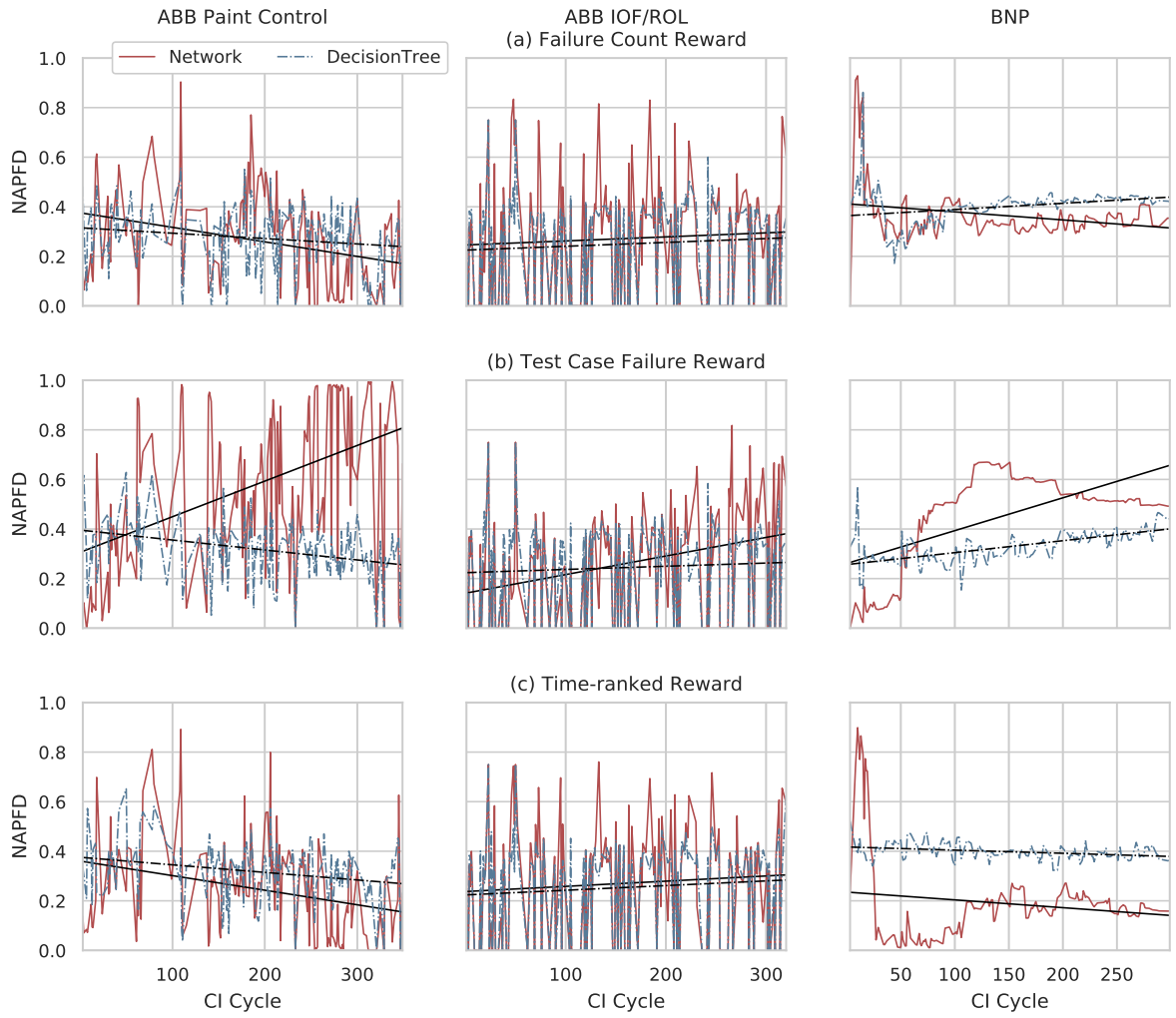


Figure 2.6: NAPFD Comparison with different Reward Functions and memory representations: best combination obtained for Test Case Failure reward and Network Approximator (straight lines indicate trend)

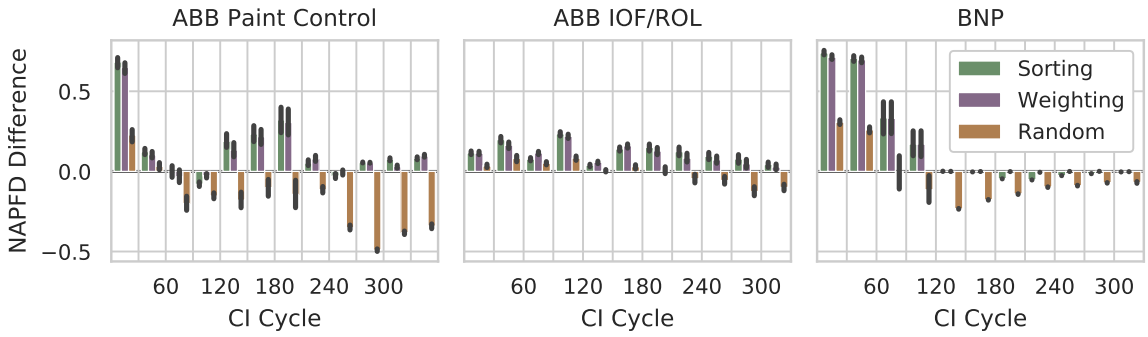


Figure 2.7: NAPFD difference in comparison to traditional methods

spect 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 as they were in commit 90. Overall, the algorithm achieves promising results, when applied to this novel dataset.

In conclusion, it is evident that RETECS can not perform significantly better than comparison method. 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.

2.3.7 Threats to Validity

Internal. RETECS is not a deterministic algorithm and because of it's exploratory nature, randomness influences the outcomes. In order to mitigate the effect of random decisions, experiments are run for 30 iterations and the presented results correspond to an average. The second thread is related to parameter selection, that due to limited computer power, should have been more extensive and therefore the chosen parameters, most likely are not optimal for each scenario. Ideally, for each specific environment parameters should be thoroughly adjusted. Finally, due to the fact that this version of RETECS is an extension of the work developed by Spieker et al. [27], there's a thread related to implementation issues. Machine learning algorithms were implemented with the *scikit-learn* library and the framework is available online for inspection and reproducibility.

External. The main gap related with external threads, pointed out by Spieker et al. [27], was the fact that the inclusion of only three datasets was not representative of the wide diversity of CI environments. Although this study bridges this gap by including a novel dataset, increasing data availability and providing more validity to this framework, four datasets still fall short of a representative number of examples.

Construct. In this study, the concepts of failed test cases and fault are indistinguishable, yet this is not always true. For example, two test cases can fail due to the same fault, and vice-versa, one test-case can reveal two or more faults. Nonetheless, because information about total faults is not easily accessible, the assumption that each test case indicates a different fault in the system is formulated. Yet by finding all failed test cases, indirectly all faults are detected too. Regarding function approximators, there are more machine learning algorithms that should be experimented and finely tuned to have a more accurate state space representation, steering the agent with more precision. Regarding the information the agent uses in the decision process, i.e. duration, last execution and execution history, has proven to fail to surpass significantly the performance of simpler traditional methods, like *Sorting*. To bridge this gap, more features should be added to enrich the information the agent has on each state, for example by using code-coverage, so that only test cases that will affect the files modified in a certain commit are considered. Finally, RETECS was only compared to three baseline approaches, although there are more in the literature that should be included, including other machine learning methods.

Chapter 3

Identifying File-Test Links with Neural Network Embeddings

3.1 Background

3.1.1 Embeddings

The idea 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.

More commonly, *One-Hot Encoding* - the process of mapping discrete variables to a vector of 0's and 1's - is used to transform categorical variables into inputs that ML models can understand . In fact, one-hot encoding is a simple embedding where each category is mapped to a different vector. However, this technique has severe limitations: (1) Dealing with high-cardinality categories originates ungovernable input and output spaces, and (2) Mappings are "blind", as vectors representing similar entities are not grouped by similarity. Therefore to reduce drastically the dimensionality of the input space and also have a more meaningful representation of categories, a supervised learning task can be designed to learn embeddings.

3.1.2 Neural Network Embeddings

Embeddings are trainable n-dimensional vectors of fixed size that represent a category and by default are initialized with random values. In order to push vectors representing similar objects together, the supervised learning task will predict whether, in this case, a modified file and a test case are linked. By doing so, during training, the embedding representation will gradually improve with gradient descent, minimizing the loss of the predictions and making more meaningful entity representations as entities that have similar behaviour will be pointing in the same direction [31]. In the case of a neural network embedding, it's parameters - the weights - are the embedding components, that are adjusted by back-propagation [1].

3.2 NNE-TCP Approach

In this section, our approach is presented with detail, explaining how both test-case and file embeddings can be learned from historical data, based on the assumption: Files that cause similar tests to transition, are similar to each other. First, a brief description of what embeddings are is provided, and then an in-depth walkthrough of the implementation of NNE-TCP.

3.2.1 Implementation

The 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. The learner is used to make new predictions on unseen data and create

[1] Algorithm for Supervised Learning of Artificial Neural Networks with Gradient Descent

test schedules that prioritize test-cases more likely to be related with files modified in a given revision. The implementation was done with the Python Deep Learning Library - Keras [32]. The steps taken to develop the framework are summarized below:

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.

The dataset should contain records of the files modified in every revision, as well as test cases that suffered transitions. Data cleaning is increasingly important as development environments become more dynamic, fast-paced and complex. Dealing with modified file and test-case records can have noise associated. Since these elements are simply files, they can become outdated, deprecated, duplicated, renamed or unused. Therefore, eliminating redundant as well as files/tests that have not been modified/applied recently, constitutes a solid data cleaning strategy.

After loading and cleaning the dataset, in order for the model to learn the supervised learning task, it needs a training set. It will be composed by pairs of the form: $(file, test, label)$. The label will indicate the ground-truth of whether the pair is or is not present in the data.

To create the training set, we need to iterate through each revision and store all pair-wise combinations of files and test-cases in a list. Nonetheless, because the dataset only contains positive examples of linked pairs, there is the need to generate negative examples, i.e. file-test pairs that are not linked, in order to create a more balanced dataset. For the positive samples - pairs that appear in the data - the label is set to 1, whereas for negative examples - pairs that are not in the data - the label is set to 0, for classification or -1 for regression. We define that a negative example occurs when a certain file is modified, but a test-case status remained unchanged, i.e. did not transitioned. Moreover, to generate a balanced dataset and there are, for example, 10 positive pairs in a revision, other 10 unique pairs of modified files and test-cases not involved in transitions are randomly chosen to form the negative examples.

As a result of having to create balanced examples for every revision and specially when dealing with large datasets, it could become unpractical, in terms of memory and processing, to store and generate the whole training set at once. Consequently, to alleviate the issue of having to store large amounts of data and overcome limited computer resources, the Keras Class *Data Generator* offers an alternative to generate data *on-the-fly* - one batch [2] of data at a time. Once the weights are updated for every batch, an epoch [3] has passed.

The Data Generator class has parameters to be adjusted: the batch-size, comprising the number of steps to be taken in an epoch; the shuffle flag, that if true mixes the order of the batches, once a new epoch begins; the negative-ratio, that represents the dataset class balance ratio. If it is equal to 1, then there are as many positive examples as there are negative, if it is equal to 2, then there are twice as many negative examples, as there are positives and so on.

Now that a training set is created, the next step is to build the learning model's architecture. The inputs of the neural network are (file, test) pairs, converted to integers and the output will be a prediction of whether or not there is a link. The model is composed by the following layers and depicted below in 3.1:

- Input: 2 neurons for each file and test-case in a pair
- Embedding: map each file and test-case to a n-dimensional vector.

[2] subset of the entire dataset
[3] number of passes through the entire dataset

- Dot: calculates the dot product between the two vectors, merging the embedding layers.
- Reshape: Layer to reshape the dot product into a single number.
- $[Optional]$ Dense: generates output for classification with sigmoid activation function [4].

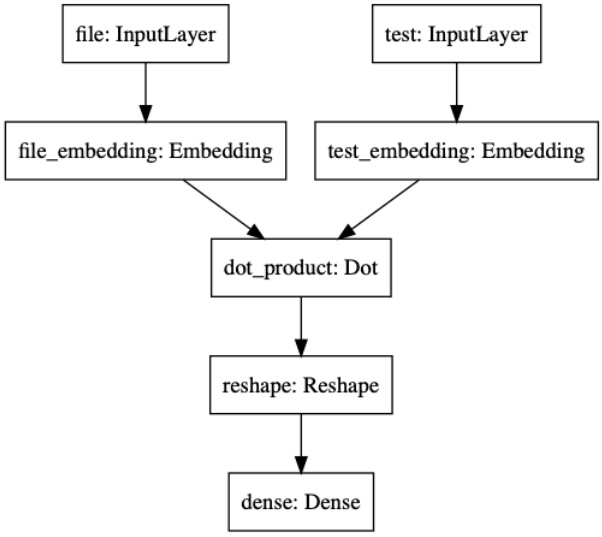


Figure 3.1: Neural Network Embedding Model Architecture

After converting each file and test-case to a unique integer, they are fed as inputs to the neural network. Afterwards, the embedding layer deals with representing each one of them as a vector in n-dimensional space. Then, the two representations need to be combined into a single number, by calculating the dot product - pair-wise multiplication and sum over all elements. This number will be the output, for a regression task, which will then be compared with the true label (-1 or 1). Then, by calculating the loss function - the mean squared error - the distance between the true and predicted labels will be minimized, by readjusting the model's weights.
For classification, because the label is either 0 or 1, a Dense layer with a sigmoid activation function needs to be added to the model to squeeze the output between 0 and 1. The chosen loss function was the binary cross-entropy, that measures the similarity between binary classes [32].

Once the model is built, the next step is to train it with examples produced by the Data Generator, for a certain number of epochs. At this stage, weights are updated such that the according loss function (depending on choosing classification or regression) is minimized and the accuracy of predicting if 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 files and test-cases embeddings.

The metrics used to evaluate the model's performance, are the accuracy and the APFD. Focusing on the latter, the ultimate goal is to maximize APFD in order to apply relevant tests as soon as possible. Since there is no information *apriori* of which test-cases will pass or fail in a given revision, the testing set will only be composed by combinations the files that were modified and all test-cases. Then the algorithm will rank test-cases by likelihood of being linked to each modified file, resulting in a matrix of scores $n \times m$ where n corresponds to the number of files that were modified in the current revision and m the total number of test-cases. Subsequently, to create a test ordering the maximum value is chosen for each column, resulting in a single vector of size m and test-cases are ranked by descending order, from higher score to lower. Finally, the APFD metric is calculated by applying test-cases according with the obtained test schedule and evaluating when faults were discovered.

Lastly, another useful application of training embeddings is the possibility of plotting them in a reduced dimensional space for intuition about entity representation.

[4] Commonly known as logistic function $S(x) = \frac{1}{e^{-x}+1}$

Since embeddings are represented in a n-dimensional manifold, one has to resort to manifold reduction techniques to represent elements in 2D or 3D. This can be done by using T-SNE: t-Stochastic Distributed Neighbors Embedding [33] or UMAP: Uniform Manifold Approximation and Projection [34], which are both utilized to map vectors to lower dimensional spaces, while preserving the structure of the manifold and the relative distances between elements.

3.3 Experimental Setup

An experimental evaluation of our approach to the algorithm is presented, with a brief description of the dataset used, the corresponding cleaning steps applied and the results obtained: both prioritizations and embedding representation in 2D space.

3.3.1 Data Description

The dataset used to trained NNE-TCP consists on historical data collected from the SVN log [5] for a team of around 90 developers, belonging to a large company of the financial sector. The data was collected over a period of four years and contains information relative to the commits, tests and modified files. In short, the dataset contain over 4000 commits, 10800 files and 4000 test-cases. Each dataset row has 3 columns: the first with the commit identifier, which is a number, then a list of the files modified in that commit and lastly, a list of the tests that transitioned. It is worth noting that for privacy purposes the data was anonymized and therefore the dataset does not contain any information that corresponds to reality.

3.3.2 Data Cleaning

Data Cleaning has 3 steps: the date from which we consider relevant files, the individual frequency of each element - files that appear few times are likely to be irrelevant or noisy - and, finally, the frequency of the pair itself - rare pairs may have occurred by chance. To remove the noise from the data we want the average number of occurrences per file to be larger - we want more density of relevant files and tests. The density varies according to the surface plot depicted in 3.3:

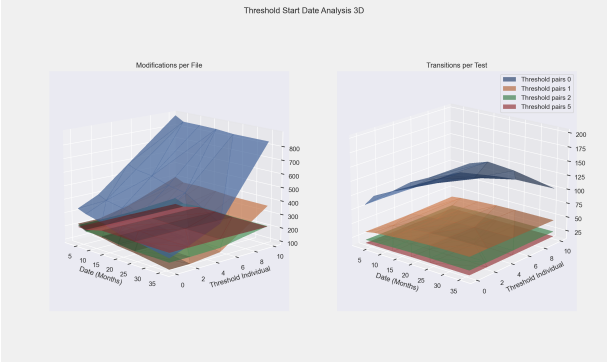


Figure 3.2: Average number of occurrences per files/tests

In the first plot, the parameter that influences the most the average modifications per file is the individual threshold, that when is high, only the most often files prevail. However, from the initial 10800 files, only 300 remain. So there is a compromise between having an expressive dataset with only very frequent pairs and having a broader scope of valid commits. In both plots, for a pair threshold of 1, it is possible to see increases in the density, whilst making sure relevant files and tests are not being dropped.

3.3.3 Evaluation Metric

the metric is the APTD

[5] Document that displays commit log messages.

3.4 Results

3.4.1 Training

After framing the problem and having the data ready and clean, a baseline was established for training NNE-TCP for the first time. Then, the algorithm is evaluated by predicting test orderings for 100 unseen commits

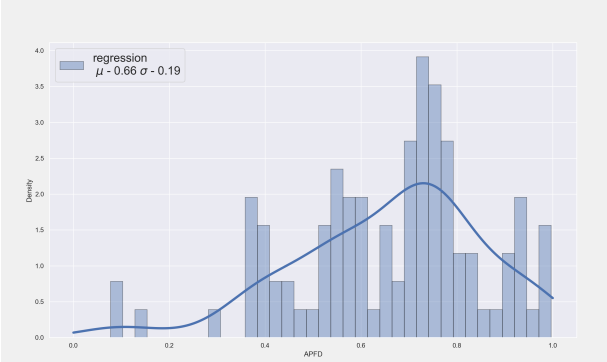


Figure 3.3: Baseline Approach: Embedding size - 100, Negative ratio - 2, batch size - 5 commits, Task - Regression and epochs - 10

By taking the mean of the distribution the result is $APFD = 0.66 \pm 0.19$, and for a first approach it represents a positive result, since ordering test-cases randomly, on average would yield an $APFD$ 0.5.

3.4.2 Cross-Validation

Now that the model is trained, the next step is to account for overfitting or underfitting, or in other words, making sure the algorithm can generalize the results and not simply perform well on the training data or the model is not complex enough to adjust to the relations present in the data. By using Scikit Learn's KFold cross validation feature, the training set can be divided into 10 separated subets called *folds*, then the model is trained 10 times, picking a different fold for evaluation every time, while the other 9 are used for training. The one fold is called the *validation-set* and, if the model is not underfitting or overfitting the data, the loss and the metric on the training set should be always close to the validation set. If the curve of the validation set is above the training set's, then there is underfitting, otherwise there is overfitting. Figure 3.4 shows the evolution of the loss function and the metric (in this case, the Mean Absolute Error) for 10 epochs, for the baseline approach trained above.

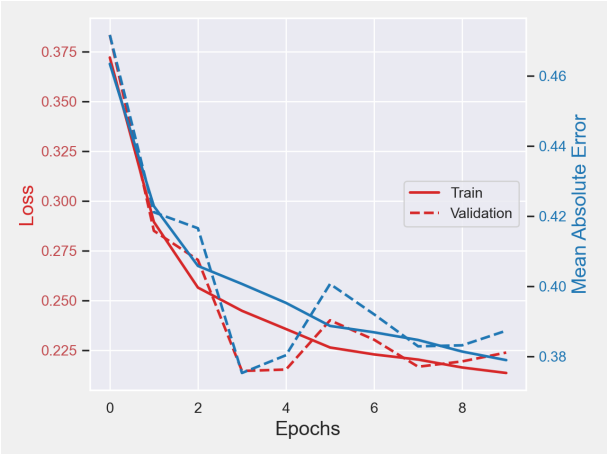


Figure 3.4: Cross Validation

Although there are discrepancies between the full line (training set) and the dashed line (validation set), it is possible to validate this model and say that there is slight overfitting.

3.4.3 Fine-Tuning

After training our model and accounting for overfitting, it is time to find the best combination of parameters that maximize our metrics. To determine them, a search in a grid was conducted, by making combinations between the values for each parameter. These parameters represent a gross approximation and likely do not correspond to their optimal value. Hereafter, if not stated otherwise, the same set of parameters are used throughout the paper. Table 3.1 shows an overview of the chosen parameters and figure 3.5 depicts the evaluation result.

Parameter	Embedding Size	Negative Ratio	Batch Size	Nb_Epochs	Task	Optimizer	Start Date
Value	200	1	1	10	Regression	SGD	1 year ago

Table 3.1: Optimal found Parameter Values

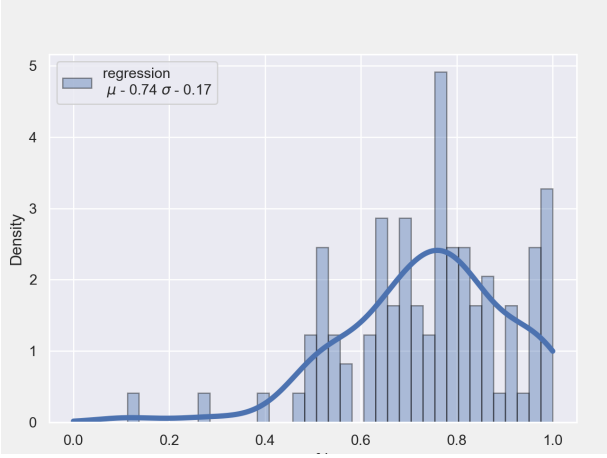


Figure 3.5: Best APFD distribution - Maximum mean value

Configuring the model with the parameters presented in Table 3.1, the best result achieved was $APFD = 0.74 \pm 0.17$, which corresponds to an increase of 48% when compared to a random approach.

3.4.4 Entity Representation

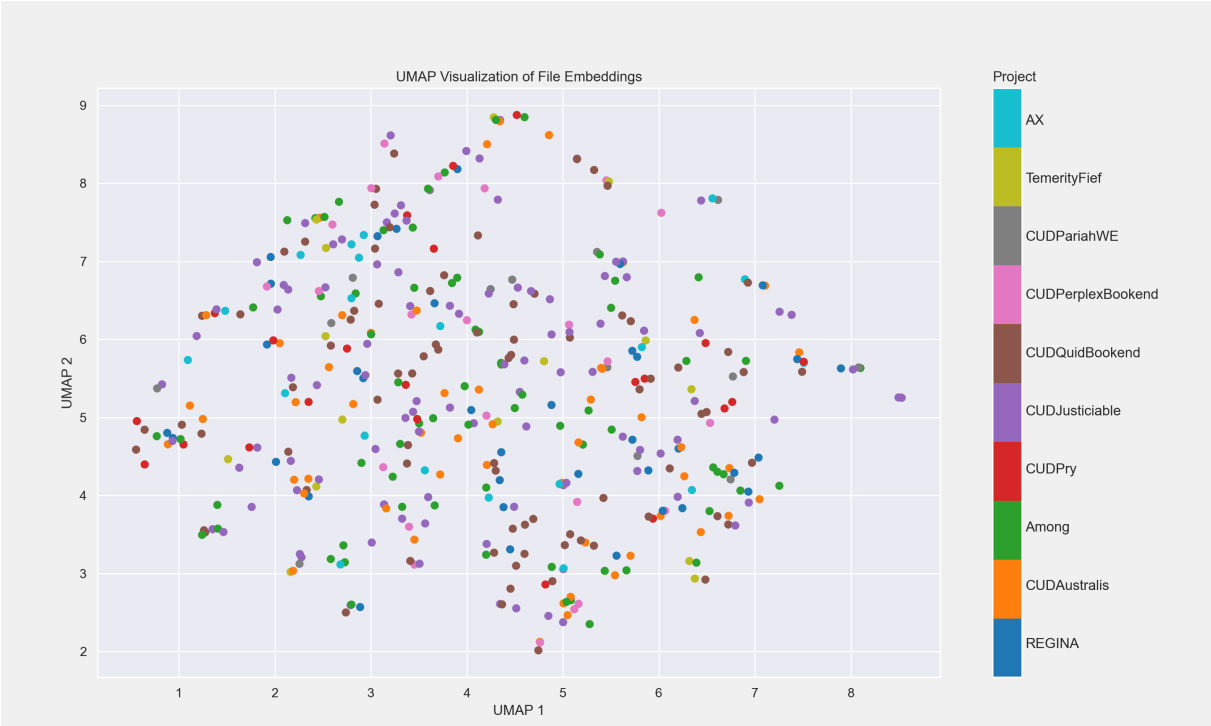
The advantage of using Embedding Models is that beyond solving a Supervised Learning Problem, the results can be visualized in 2D space. Figure 3.6 shows the embedding representation for files and test-cases, by using the manifold reducing technique UMAP and by labelling files and tests appropriately. Due to limitations in the data, the available labels that can be extracted are the directory where each file/test is stored and only for tests an identifier written in the name of the tests pointing to functionality [6]. Figure 3.6 shows labelled embeddings representation, for the ten most frequent labels. Also, labels were made by humans, hence some of them are not written in non-uniform format and are incorrect or uninformative.

One general observation is that embedding representation and the current labelling show no correlation. Entities are grouped by likelihood of being involved in transitions together and in the case of modified files, that does not occur for the ones that are kept in the same directories.

3.4.5 Threats to Validity

Internal. The first threat to validity is associated with randomness when training the model, which was mitigated by using cross-validation, that performs training multiple times. Another threat can derive from errors associated with our code implementation. Both Scikit-learn and Keras are well established frameworks used for Machine Learning and furthermore, the code is open-source and available online [7] for

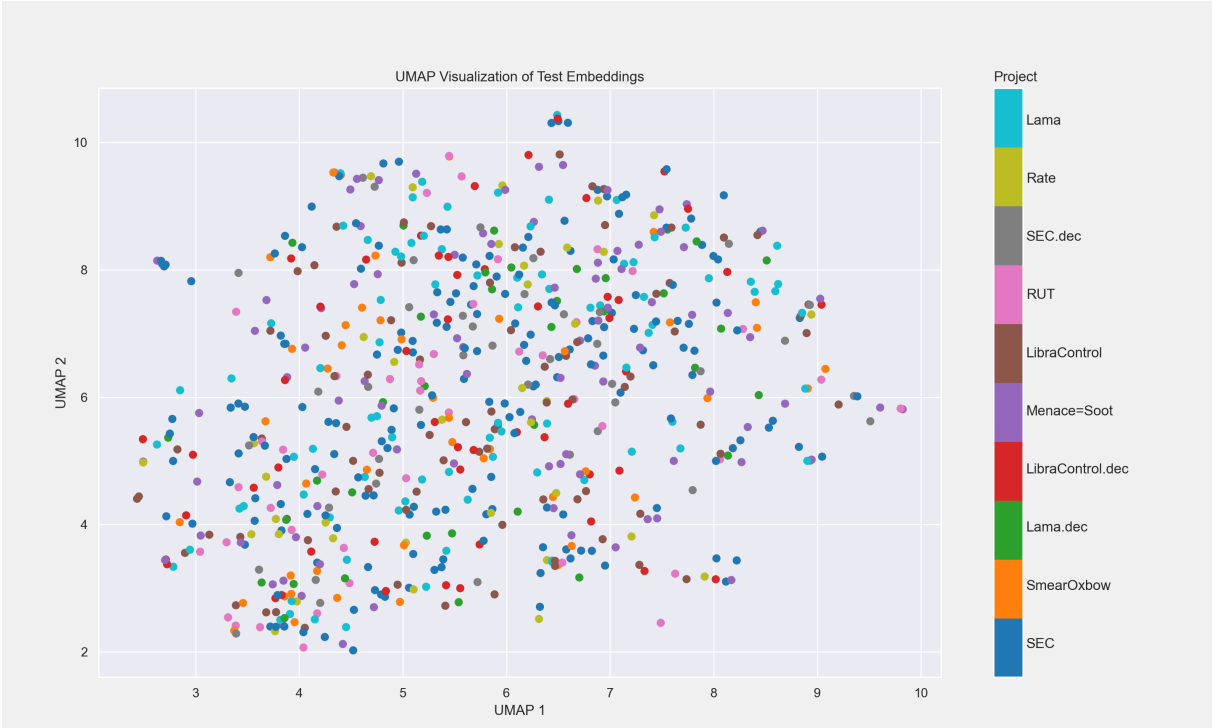
[6] information provided by domain expert
[7] https://github.com/jlousada315/NNE-TCP



reproduction and inspection purposes.

External. The evaluation made is based on a single development environment, which is a major limitation considering the amount of real-world scenarios where CI is applied. This threat has to be addressed by running more experiments on data from several industries, with different development paces, team size, resources, etc. To contribute to more data availability, publication of the dataset used in this paper is being authorized.

Construct. In real-world complex CI systems, sometimes test-cases change status due to extraneous reasons: system dependencies of several platforms can affet the outcome of a test , the infrastructure where tests are run can suffer critical failures or malfunctions, some tests can fail to produce the same result each time they are run (flaky tests). Therefore it is not certain that a test case changed status, because a certain file was modified.
Regarding the features used for training, our model only looks at the link between modified files and tests, whereas in real life additional features may have a positive impact on the capability to make better predictions, e.g. commit author, execution history of each test - test that are not applied in a long time have more chance of changing status and tests that have failed recently are more likely to fail again -, test age, etc. To better validate the results achieved by NNE-TCP it should be compared with other Machine Learning based Test Case Prioritization techniques described in the literature.


UMAP Visualization of Test Embeddings
UMAP 1
UMAP 2
Project
Lama
Rate
SEC.dec
RUT
LibraControl
Menace=Soot
LibraControl.dec
Lama.dec
SmearOxbow
SEC

Chapter 4

Conclusions

In this study, an extension of the RETECS framework was developed, in order to determine it's ability to prioritize and select test-cases, when presented with a novel dataset, extracted from a different CI environment, validating its generalization. Additionally, Decision Trees were applied for the first time in this context as a model for state space representation.

Result indicate that RETECS can effectively, create meaningful test schedules in different contexts. In the BNP case, with a combination of the Test Case Failure reward with the Aritifical 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 0.2 and has the algorithm progresses, the trend shows values over 0.6.

Including Decision Trees in the study did not produce enhancements relative to the ANN, 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 suited to apply, in future reseach, to other CI environments with specific characteristics.

A novel approach to Test Case Prioritization using Machine Learning was presented, NNE-TCP, by combining Neural Network Embeddings with file-test links obtained from historical data. NNE-TCP is a lightweight and modular framework that not only predicts meaningful prioritizations, but also makes useful entity representations in the embedding space, grouping together similar elements.

Evaluation results point to a major improvement, when compared to the currently used random ordering approach, thus discovering an effective prioritization technique that only requires information about which files in the system, when modified, cause test-cases to transition status. However, it is expected that incorporating additional features would enhance even further the results.

Finally, the ability to visualise embeddings in 2D space represents an added value, providing insights on the structure of the data. It showed that there is no correlation between the current labels and how these elements are grouped by similarity, which can be due to labelling mistakes or a that there is a new, more elegant, way to organize files and tests by similarity.

4.1 Achievements

4.2 Future Work

The results obtained strongly indicate that RETECS can match performance with traditional prioritization methods and is flexible enough to adapt to different contexts. However due to its complexity in relation to traditional methods, to be worth applying, it's performance has to surpass these methods. To do so, it needs more information to formulate better reasonings of expected failures, e.g. links between test cases and modified files.

Regarding Machine Learning model as function approximators, Decision Trees showed worse performance when compared to ANN's, however more parameter tuning should be conducted to try to find optimal values for all parameters and more models should be evaluated, e.g. Nearest Neighbors.

Furthermore, in real world environments, test-cases are usually run on a grid that allows for paralellization. In our framework, test cases are applied sequentially, one by one, based on their rank. If two test cases are very similar, most likely they will appear together in a test-schedule and detect exactly

the same fault. With paralellization, it would be more fruitful to create groups of non-redundant tests to maximize the ”surface” covered by each group of test-cases on each run.

The results of this work were the first step towards applying Embeddings in the context of TCP, by using file-test links as features, estabilishing a solid baseline for further studies. Due to limited time and computer power, parameter tuning analysis can be refined further, by exploring more combinations of parameters and measuring the impact on the APFD.

Furthermore, the approach should take into consideration additional features to allow better understanding of expected transitions, e.g. status history of test cases, shortening even more the feedback-loop. In terms of entity representation, with a better extraction of labels, embedding groupings can give valuable insight about the architecture of the system, giving the chance to optimize it and reorganize it in the most convenient way.

Finally, NNE-TCP should be validated and tested against other types of data, so that we know how to make it more flexible and adapt it to different contexts.

Bibliography

[1] M. Santolucito, J. Zhang, E. Zhai, and R. Piskac. Statically verifying continuous integration configurations, 2018. Paper on technical issues regarding Continuous Integration systems.

[2] 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, page 113–122. IEEE Press, 2017. ISBN 9781538627174. doi: 10.1109/ICSE-SEIP.2017.13. URL `https://doi.org/10.1109/ICSE-SEIP.2017.13`.

[3] F. Palma, T. Abdou, A. Bener, J. Maidens, and S. Liu. An improvement to test case failure prediction in the context of test case prioritization. pages 80–89, 10 2018. doi: 10.1145/3273934.3273944.

[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] C. Catal. Software fault prediction: A literature review and current trends. *Expert Syst. Appl.*, 38: 4626–4636, 04 2011. doi: 10.1016/j.eswa.2010.10.024.

[6] S. Chen, Z. Chen, Z. Zhao, B. Xu, and Y. Feng. Using semi-supervised clustering to improve regression test selection techniques. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 1–10, 2011.

[7] 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*, pages 29:1–29:15. ACM, 2019.

[8] R. Lachmann, S. Schulze, M. Nieke, C. Seidl, and I. Schaefer. System-level test case prioritization using machine learning. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 361–368, Los Alamitos, CA, USA, dec 2016. IEEE Computer Society. doi: 10.1109/ICMLA.2016.0065. URL `https://doi.ieeecomputersociety.org/10.1109/ICMLA.2016.0065`.

[9] R. Potvin and J. Levenberg. Why google stores billions of lines of code in a single repository. *Commun. ACM*, pages 78–87, 2016. ISSN 0001-0782.

[10] C. Jaspan, M. Jorde, A. Knight, C. Sadowski, E. K. Smith, C. Winter, and E. Murphy-Hill. Advantages and disadvantages of a monolithic repository: A case study at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, pages 225–234. ACM, 2018. This source compares and draws conclusion on whether it is more advantageous to prefer a monolithic repository over a multi-repository, by conducting a survey to Google's engineers and drawing conclusion based on the collected data, reaching to a conclusion that there are pros and cons depending on the context of the company.

[11] B. Meyer. Seven principles of software testing. *Computer*, pages 99–101, 2008. Introduction on the fundamental pilars of software testing in a general way, useful in defining concepts.

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

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[14] G. Rothermel and M. J. Harrold. A framework for evaluating regression test selection techniques. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 201–210. IEEE Computer Society Press, 1994. This source explores in detail the topic of test selection.

[15] G. Rothermel, R. J. Untch, and C. Chu. Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.*, 2001. This source explores in detail the topic of test prioritisation.

[16] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming google-scale continuous testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP '17, pages 233–242. IEEE Press, 2017. This paper served as a guideline to get to know how much volume of code changes a company the scale of Google has to handle. Also this source provides different complementary points of view of these types of systems.

[17] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011. ISBN 0123814790.

[18] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *Software Engineering, IEEE Transactions on*, 39: 757–773, 06 2013. doi: 10.1109/TSE.2012.70.

[19] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26, 2015.

[20] D. H. Moore II. Classification and regression trees, by leo breiman, jerome h. friedman, richard a. olshen, and charles j. stone. brooks/cole publishing, monterey, 1984,358 pages, $27.95. *Cytometry*, 8(5):534–535, 1987. doi: 10.1002/cyto.990080516. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/cyto.990080516`.

[21] M. A. Nielsen. Neural networks and deep learning, 2018. URL `http://neuralnetworksanddeeplearning.com/`.

[22] P. Mehta, M. Bukov, C.-H. Wang, A. G. Day, C. Richardson, C. K. Fisher, and D. J. Schwab. A high-bias, low-variance introduction to machine learning for physicists. *Physics Reports*, 810:1–124, May 2019. ISSN 0370-1573. doi: 10.1016/j.physrep.2019.03.001. URL `http://dx.doi.org/10.1016/j.physrep.2019.03.001`.

[23] K. Eremenko and H. d. Ponteves. *Deep Learning A-Z*. SuperDataScience, 2017. URL `https://www.udemy.com/course/deeplearning/learn/lecture/6753756#notes`.

[24] 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. ISBN 9781450342186. doi: 10.1145/2950290.2983954. URL `https://doi.org/10.1145/2950290.2983954`.

[25] J. Liang, S. Elbaum, and G. Rothermel. Redefining prioritization: Continuous prioritization for continuous integration. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 688–698, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. doi: 10.1145/3180155.3180213. URL `https://doi.org/10.1145/3180155.3180213`.

[26] S. Yoo, M. Harman, and S. Ur. Measuring and improving latency to avoid test suite wear out. *IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2009*, 2009. doi: 10.1109/ICSTW.2009.10.

[27] 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. doi: 10.1145/3092703.3092709. URL `http://dx.doi.org/10.1145/3092703.3092709`.

[28] Z. Wu, Y. Yang, Z. Li, and R. Zhao. A time window based reinforcement learning reward for test case prioritization in continuous integration. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, Internetware '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450377010. doi: 10.1145/3361242.3361258. URL https://doi.org/10.1145/3361242.3361258.

[29] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0262193981.

[30] L. Pyeatt and A. Howe. Decision tree function approximation in reinforcement learning. 07 2001.

[31] A. Geron. *Hands-on machine learning with Scikit-Learn and TensorFlow :*. O'Reilly,, Beijing, first edition. edition, 2017. Includes QR code.

[32] F. Chollet et al. Keras, 2015. URL https://github.com/fchollet/keras.

[33] L. Van Der Maaten. Accelerating t-sne using tree-based algorithms. *J. Mach. Learn. Res.*, 15(1): 3221–3245, Jan. 2014. ISSN 1532-4435.

[34] L. McInnes, J. Healy, and J. Melville. Umap: Uniform manifold approximation and projection for dimension reduction, 2018. URL https://arxiv.org/pdf/1802.03426.pdf. cite arxiv:1802.03426Comment: Reference implementation available at http://github.com/lmcinnes/umap.

Appendix A

Supervised Learning

Supervised Learning is the task of learning by example and it is divided into a training phase and a testing phase. In the first, the learner receives labelled data as input and the output is calculated, then based on a cost function that relates the predicted value with the actual value, the parameters of the model are updated such that the cost function is minimized. Then, in the second phase, the model is tested with new data, that it has never seen before, and its performance is evaluated. Supervised Learning can be divided into two groups: Classification and Regression. In classification, each item is assigned with a class or category, e.g. if an email is considered spam or not-spam, represents a binary classification problem. Whereas in regression, each item is assigned with a real-valued label. [17]

In accordance with Kamei et al. [18] and Yang et al. [19], the typical predictive model used is the Logistic Regression. Here, other models are explored, such as Decision Trees and Artificial Neural Networks.

A.1 Decision Trees

Another potential candidate to treat our data may be a Decision Tree Classifier, which is categorized by having a flowchart-like tree structure, where there are nodes and branches. Each internal node denotes a test on an attribute and the branch represents the outcome of that test, then the nodes at the bottom of the tree, called *leaf-nodes* whereas the topmost node is the *root-node*, holds a class label. An example of such tree can be seen below[17]:

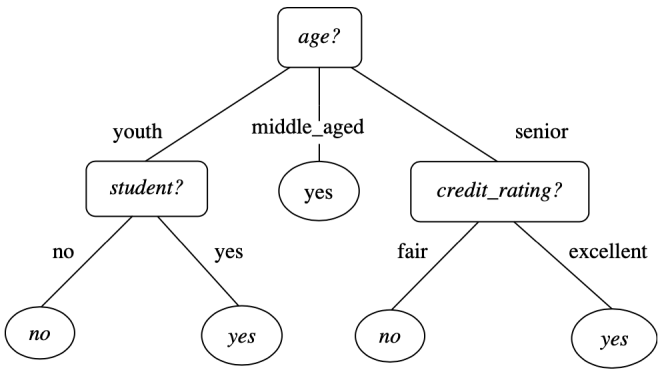


Figure A.1: Each internal (non-leaf) node represents a test on an attribute. Each leaf node represents a class (if the client is likely to buy the product yes/no)

Then after training, when a new element serves as input of a Decision Tree, a path is traced from the root until the leaf node, revealing the prediction of which class that element belongs. Decision tree learning can be based in several algorithms. The most commonly used are ID3 (Iterative Dichotomiser 3) , C4.5 and CART (Classification and Regression Trees). In this work, the chosen algorithm is provided

by the *scikit-learn* package and it uses an optimized version of CART. The mathematical formulation is as follows:

A.1.1 Mathematical formulation

Given training vectors $x_i \in \Re^n, i = 1, ...m$ and a class label vector $y \in \Re^m$, a decision tree recursively partitions the space such that the samples equally labeled are grouped together.

Considering the data at node l be represented by D. For each candidate split $Q = (j, t_l)$,, where j corresponds to the *j-th* feature and t_m the threshold, partition the data into the subsets $D_{left}(\theta)$ and $D_{rigth}(\theta)$.

$$D_{left}(\theta) = (x, y)|x_j \leq t_l \tag{A.1}$$
$$D_{right}(\theta) = D\ D_{left} \tag{A.2}$$

Then, because data is not easily separable,i.e. partitions not often contain elements with the same class label, one defines a criterion called *impurity*, that measures the probability of finding a mislabelled element in the subset. The impurity at m is determined by using an impurity function $H()$, that depends if the task is classification or regression.

$$G(D, \theta) = \frac{n_{left}}{N_l} H(G(D_{left}, \theta)) + \frac{n_{right}}{N_l} H(G(D_{right}, \theta)) \tag{A.3}$$

where n_{left} is the number of attributes partitioned to the left, n_{right} to the right and N_m the total number of attributes in a node. The function H() is commonly defined

as Gini Impurity:

$$H(D_m) = \sum_k p_{mk}(1 - p_{mk}) \tag{A.4}$$

as Entropy:

$$H(D_m) = -\sum_k p_{mk}(log(p_{mk}) \tag{A.5}$$

where p_{mk} is the probability that an item k with label m is chosen.
The goal is to select parameters such that the impurity is minimised, such that:

$$\theta^* = argmin_\theta G(D, \theta) \tag{A.6}$$

Finally, recursively apply the same reasoning to subsets $D_{left}(\theta^*)$ and $D_{right}(\theta^*)$, until maximum tree depth reached, i.e. $N_m < min_{samples}$ [20]

A.2 Artificial Neural Networks

In general terms, a neural network (NN) is a set of connected input/output units in which each connection has a weight associated with it. The weights are adjusted during the learning phase to help the network predict the correct class label of the input vectors.

In light of the knowledge psychologists and neurobiologist have on the structure of the human brain, more precisely on how neurons pass information to one another, this way it was possible to look for methods to develop and test computational analogues of neurons. Generally, a NN is defined as a set of input/output *units* (or *perceptrons*) that are connected. Each connection has a weight associated with it, and these weights are adjusted in such manner, that the network is able to correctly predict the class label of the input data. The most popular algorithm for NN learning is *back-propagation*, which gained popularity since 1980. [17]

Usually, NN training phase involves a long period of time and a several number of parameters have to be set to build the network's structure and architecture. However, there is no formal rule to determine their optimality and this is achieved empirically, by running different experiments. This, nonetheless,

raises an issue of poor interpretability, that makes it hard for humans to grasp what the optimal parameters mean. On the other hand, NN's compensate on easy-going implementation and ability to deal with noisy data and, so far, have been used to solve many real-world problems, such as hand written text recognition, medical diagnosis and finance [17].

In the following sections, the reader is guided in detail through the architecture of a NN and given a brief explanation of how back-propagation works.

A.2.1 Perceptron

A perceptron is the most elementary unit of a NN. In similarity to brain cells, that are composed by dendrites, that receive electric signals (input), the nucleus that processes them and the axon, that sends too an electric signal to other neurons (output). In our case, the perceptron receives a vector of inputs $x_1, x_2, ...x_n$ and associated to each one there are weights $w_1, w_2, ...w_n$ that symbolize the influence each input has on the output. As an example, consider a perceptron with an input vector composed of 3 inputs x_1, x_2, x_3.

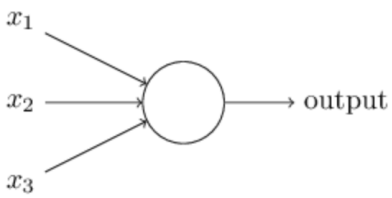


Figure A.2: Perceptron simple example)

Let us say the desired output is either 0 or 1. This value shall be determined by weighing the sum $\sum_j w_j x_j$ and checking if the value surpasses a given threshold [21]. So the output can be defined as:

$$\textbf{output} = \begin{cases} 0, & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1, & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases} \tag{A.7}$$

Also, one can replace $\sum_j w_j x_j$ as the dot product $w \cdot x$ and move threshold to the other side of the equation and call it b, for bias.

$$\textbf{output} = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases} \tag{A.8}$$

Here, the bias term can be thought as the susceptibility of the neuron being activated, the large the value of b, more easily the output is 1.

A.2.2 Activation Functions

There is a problem related to the sensitivity the perceptron has when $w \cdot x + b$ is close to zero. Small changes in the weights may cause a drastic effect on the outcome, because the expression corresponds to a step function. What can be done is define an activation function $\sigma(z)$ that transforms the expression above. Commonly, $\sigma(z)$ is defined as the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$, or the rectifier function $\sigma(z) = max(0, z)$, comparing the three curves for the same values of w_i and b_i:

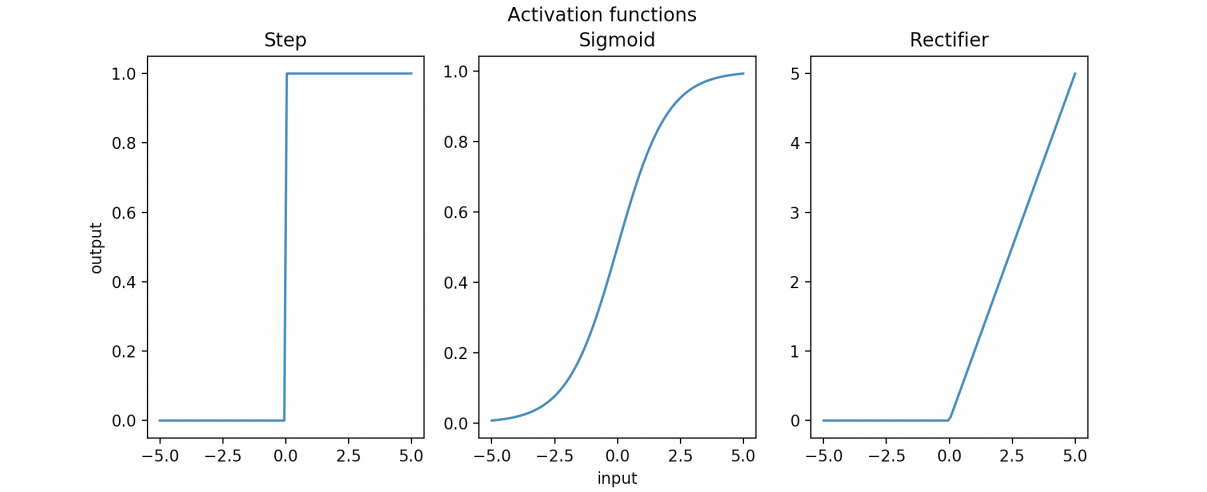


Figure A.3: Comparative analysis of different activation functions (made in Python))

In the first image, the transition from 0 to 1 is abrupt and steep, this way values that are close to the threshold can have dramatic distinctions in output and that is not desirable, because NN's learn by making little adjustments on the values of the weights. By using the sigmoid function values are smoothed out in a way that values no longer have a binary outcome, but rather a continuous and progressive approximation to the values zero or one. For high positive inputs, the output is very close to one, the same happens for high negatives close to 0 (like in the step function case) and in between the curve is softer, also the sigmoid function in differentiable in all its domain. The third case corresponds to the rectifier function that only activates for positive values and due to its easy implementation, its usage because relevant in reaching high performance.

A.2.3 NN architecture

Now, very much alike the human brain, perceptrons are assembled together to form a connected structure called a network. By grouping perceptrons of the same type. - input, output or neither.- layers are form and there are three types: input layer, output layer and *hidden* layer. The neuros that belong to the hidden layer are simply neither input nor output neurons, they serve as a mean of adding more complex relations between the variables input and weights. In terms of configuration, there is only one input layer and one output layer, with variable size. However multiple hidden layers may exist. Let us take the following example with an input vector of size 6, an output vector with size 1 and 2 hidden layers of size 4 and 3. [21].

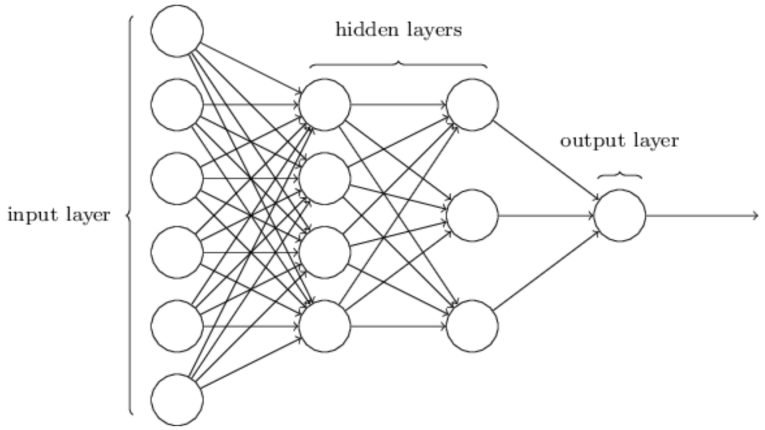


Figure A.4: Architecture example of a NN [21]

Every node as a connection to a node from the next layer, but what is changeable is the number of hidden layers and the number of nodes in each layer. Although there are some heuristics to deter-

mine the best configuration.- like choosing the number of neurons by averaging the number of neurons in the input and output layer.- each dataset has its own characteristics, it becomes difficult to come up with a rule of thumb that works in generality, so one has to determine the parameters by running experiments.[21]

To wrap up the functioning of a NN, from an input vector x, the neurons from the next layer will be activated or not by computing $w \cdot x + b$, then the same process occurs until the output node is reached, in this case yielding the result 1 or 0, this left-to-right process is called *forward-propagation*. Of course, this is not the process of learning, most likely if one compares the predicted result with the actual value, it is a 50% chance of being correct. So how does a NN learn?

A.2.4 Learning with Gradient Descent

As mentioned above, the desired result is achieved once the difference between the predicted value $\hat{y}$ and the actual actual y is minimized. The goal is to find an algorithm that will indicate which values for weights and biases will produce the correct output. In order to quantify the performance achieved so far, let us define a cost function $C(Y, \theta)$,where in the dataset Y, θ are the parameters, , m the total number of samples in the dataset and i its index.:

$$C(Y, \theta) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2 \tag{A.9}$$

In this formula, one sees that it is close to zero, exactly when the predicted value matches the expected value. A possible approach to reach this result could be to brute force the values of the parameters: trying out several combinations of weights and biases, narrowing down in each iteration, eventually converging towards the quadratic value, or the minimum of the cost function. However, this approach does not scale. With just 25 weights and assuming each weight can have $1,000$ different values, there are $1000^{25} = 10^{75}$ possible combinations, which is infeasible to compute, even for modern supercomputers[22].

A more efficient strategy is to use Gradient Descent to minimize the cost function. The first step would be to randomly initialize all the weights and bias, similar to the initial condition of a differential equation, and then calculate the output $\hat{y}$ and compute the cost function. Most likely, it will not yield a result close to the minimum, so now one needs to know is what direction of the curve points towards the minimum and then ”jump” a step towards that direction. This is achieved by calculating the gradient of the cost function, with respect to θ.

$$v_t = \eta_t \nabla_\theta C(Y, \theta) \tag{A.10}$$

$$\theta_{t+1} = \theta_t - v_t \tag{A.11}$$

where η_t is the *learning rate* that defines the length of the step taken in the direction of the gradient , at time step t. By configuring a small learning rate it is guaranteed local minimum convergence, however implying a high computational cost and not assuring an optimal solution by landing on a global minimum. On the other hand, by choosing a large learning rate the algorithm can become unstable and overshoot the minimum. (possible oscillatory dead-end).[22]. Minimising a cost function looks this:

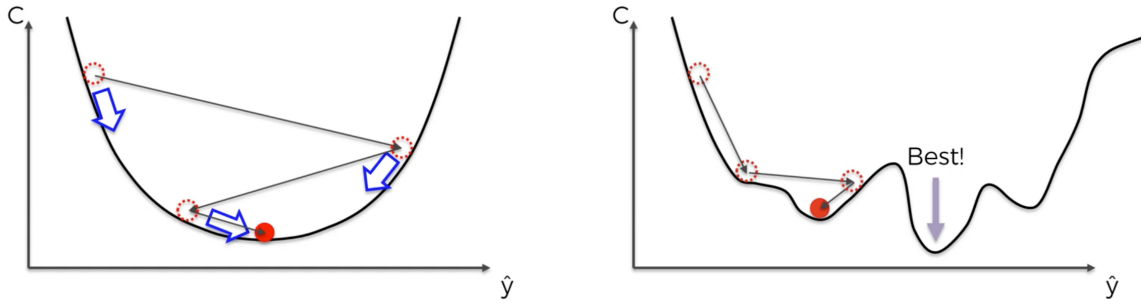


Figure A.5: Global and Local minimum determination [23]

In this one dimensional example, the gradient yields the direction towards the minimum and the learning rate determines the length of the step. However, in the second case, there are some cases where convergence is sub-optimal. This information hints the conclusion that choosing simple gradient descent, as our minimizing algorithm, has many limitations, namely:

- *Finding global minimum* - only local minimum are guaranteed to be found under the gradient descent algorithm.
- *Sensitive to initial conditions* - depending on the starting point, the final location most likely will be a different minimun, hence there is a high dependency on initial conditions.
- *Sensitive to learning rate* - as seen above, learning rate can have a huge impact on the algorithm's convergence.
- *Computationally expensive* - although better than brute force, when handling large datasets, computation rapidly increases cost. Possible solution is to just apply the algorithm to a subset or *"mini-batch"*
- *Isotropic parameter-space* - Learning rate is always the same independent of the "landscape". Ideally, the learning rate should be larger in flatter surfaces and smaller in steeper ones. [22]

A.2.5 Stochastic Gradient Descent (SGD)

To handle the limitations described in the section above, a novel, more efficient algorithm is proposed: the Stochastic Gradient Descent (SGD). The advantage is that, not only the method is more computationally efficient, but it also deals with non-convex cost-functions, on the contrary of simple gradient descent, also called *batch* gradient descent method, because it plugs every item of the dataset into the neural network, obtains the predicted values, calculates the cost function by summing the square differences to the expected value and only then the weights are adjusted. [21] SGD works in a different way, here the batch is divided into n/M subsets or *mini-batches* $B_k, k = 1, ..., n/M$, where n is the total number of data points and M the mini-batch size. The gradient now takes the following form:

$$\nabla_\theta C(Y, \theta) = \frac{1}{2m} \sum_{i=1}^{n} \nabla_\theta (\hat{y}^{(i)} - y^{(i)})^2 \rightarrow \frac{1}{2m} \sum_{i \in B_k} \nabla_\theta (\hat{y}^{(i)} - y^{(i)})^2 \tag{A.12}$$

Each mini-batch B_k is plugged into the NN, the cost function is calculated and the weights are updated. This process repeats k times until the whole data set is covered. A full run over all n points is denoted as an *epoch*. [22]

In sum, SGD has two very important advantages: not only eliminates the local minimum convergence problem, by introducing stochasticity , but also it is much more efficient in computational power, because only a subset of n data points has to be used to approximate the gradient.

A.2.6 Backpropagation

So far, a review of the basic structure and training of NN has been provided: starting from the elementary unit.- the perceptron - up to how many of them can be assembled, covering the most common types of activation functions. Then the concept of forward propagation was introduced along with a cost function that allows to judge whether the model created explains the observations.

The goal is to minimise the cost function, resorting, in a first approach, to the gradient descent method which, as seen above, has severe limitations, concluding that the Stochastic Gradient Descent algorithm is most suited for this task. However, SGD still requires us to calculate the derivative of the cost function with respect to every parameter of the NN. So a forward step has to be taken to compute these quantities efficiently. - via the **backpropagation** algorithm. - to avoid calculating as many derivatives as there are parameters. Also backpropagation enables the determination of the contribution of each weight and bias to the cost function, altering more, the weights and biases that contribute the most, thus understanding how changing these particular values will affect the overall behaviour. [21]

The algorithm can be summarized into four equations, but first, let us define some notation. Assuming a network composed of L layers with index $l = 1, ...L$ and denote by $w^l_{j,k}$ the weight connecting from the *k-th* neuron of layer *l-1* to the *j-th* neuron in layer *j*. The bias of this neuron is denoted b^j_l. By constructing, one can write the activation function a^l_j of the *j-th* neuron in the *l-th* layer, by recursively writing the relation to the activation a^{l-1}_j of neurons in the previous layer *l-1*.

$$a^l_j = \sigma\left(\sum_k \omega^l_{j,k} a^{l-1}_k + b^l_j\right) = \sigma(z^l_j) \tag{A.13}$$

When calculating the cost function C, one only needs to take directly into account the activation from the neurons of layer L, of course the influence of the neurons from previous layers is underlying. So one can define the quantity, Δ^L_j, which is the error of the *i-th* neuron in layer L , as the change it will cause on the cost function C, regarding weighted input z^L_j:

$$\Delta^L_j = \frac{\partial C}{\partial z^L_j} \tag{A.14}$$

Generally, defining the error of neuron j of any layer l and applying the chain rule to obtain the first equation of the algorithm:

$$\Delta^l_j = \frac{\partial C}{\partial z^l_j} = \frac{\partial C}{\partial a^l_j}\frac{\partial a^l_j}{\partial z^l_j} = \frac{\partial C}{\partial a^l_j}\sigma'(z^l_j) \tag{A.15}$$

Now, one can relate the error function Δ^L_j to the bias b^l_j to obtain the second backpropagation relation:

$$\Delta^L_j = \frac{\partial C}{\partial z^L_j} = \frac{\partial C}{\partial b^l_j}\frac{\partial b^l_j}{\partial z^l_j} = \frac{\partial C}{\partial b^l_j} \tag{A.16}$$

Since, $\frac{\partial b^l_j}{\partial z^l_j} = 1$, from A.13. Furthermore to find another expression for the error, one can relate the neurons in layer l with the neurons in layer $l + 1$:

$$\Delta^l_j = \frac{\partial C}{\partial z^l_j} = \sum_k \frac{\partial C}{\partial z^{l+1}_k}\frac{\partial z^{l+1}_k}{\partial z^l_k} \tag{A.17}$$

$$= \sum_k \Delta^{l+1}_k \frac{\partial z^{l+1}_k}{\partial z^l_k} \tag{A.18}$$

$$= \left(\sum_k \Delta^{l+1}_k \omega^{l+1}_{k,j}\right)\sigma'(z^l_j) \tag{A.19}$$

This is the third relation. Finally, to derive the final equation, one differentiates the cost function with respect to the weights $\omega^l_{j,k}$:

$$\frac{\partial C}{\partial \omega^l_{j,k}} = \frac{\partial C}{\partial z^l_j}\frac{\partial z^l_j}{\partial \omega^l_{j,k}} = \Delta^l_j a^{l-1}_k \tag{A.20}$$

The combination equations A.15, A.16, A.19 and A.20, constitutes the backbone of the backpropagation algorithm that is used to train NN's in a very efficient way [22]. It described in the following steps:

Backpropagation Algorithm:

1. **Activation at input layer** - compute the activations of the input layer a^1_j.
2. **Forward propagation** - Through equation A.13, determine the activations a^1_j and the linear weigthed sum z^l_j, for each layer l.
3. **Error at layer** L - calculate the error Δ^L_j, using equation A.16.
4. **Propagate the error "backwards"** - calculate the error Δ^l_j, for all layers l, using equation A.19.

5. **Determine gradient** - use equation A.16 and A.20 to calculate $\frac{\partial C}{\partial b_j^l}$ and $\frac{\partial C}{\partial \omega_{j,k}^l}$.

[22]

In sum, the application of this algorithm consists on determining the error at the final layer and then ”chain-ruling” our way back to the input layer,find out quickly how the cost function changes, when the weights and biases are altered. [21]