TÉCNICO LISBOA

Test Case Prioritization Optimization with Machine Learning

A Case-Study at BNP Paribas

João Luís Xavier Barreira Lousada

Thesis to obtain the Master of Science Degree in

Technological Physics Engineering

Supervisor(s): Prof. Doutor Rui Manuel Agostinho Dilão
Dr. Miguel Ângelo Maia Ribeiro

Examination Committee

Chairperson: Prof. Doutor Ilídio Lopes
Supervisor: Dr. Miguel Ângelo Maia Ribeiro
Member of the Committee: Dr. Claude Yves Cochet

December 2020

Acknowledgments

In the first place, I would like to thank everyone at BNP Paribas that was involved in the course of the internship. To Dr. Claude Cochet,Dr. François Girardot and Dr. Pedro Correia, I show my deepest appreciation for the opportunity, interest and time spent from your busy schedule to give me the necessary tools to develop this work.

I would like to acknowledge my two supervisors: Prof. Rui Dilão and Dr. Miguel Ribeiro, for the precious guidance and advice provided throughout the thesis. Prof. Rui Dilão enabled the transfer of knowledge learned in academia to it's application in an industrial environment. Giving me the opportunity to develop a valuable skill, crucial for my future career. To Dr. Miguel Ribeiro, I would like to show gratitude for being tireless, involved in every step of the way and always available for discussion.

To my parents and my family, thank you for all the effort, confidence and support. Without you, I would not be who I am today.

Last but not least, to Beatriz for all the sanity-checks and for being there for me, both in good and bad times, every step of the way.

Resumo

No contexto da engenharia de software, a Integração Contínua tornou-se numa etapa indispensável para a gestão sistemática dos ciclos de vida de desenvolvimento de software. Cada vez mais, grandes indústrias dispendem uma grande quantidade de recursos para manter o sitema atualizado e operacional, devido à grande quantidade de alterações e adição de funcionalidades, que se vão acumulando ao longo do tempo. Portanto, aplicar testes de software de maneira contínua e eficiente constitui uma componente-chave para assegurar a produção de software de qualidade. Ao selecionar os testes mais promissores, o tempo entre comunicar uma alteração e receber feedback é reduzido, aumentando a produtividade e reduzindo custos. Para melhorar a eficiência, foram desenvolvidos dois algoritmos de prioritização.

Em primeiro lugar, estendemos a investigação da aplicação de Reinforcement Learning para otimizar estratégias de testagem. Tendo sido comprovado que esta constituiu uma estratégia tão profícua quanto os métodos tradicionais. Testámos a sua capacidade de adaptação a novos ambientes, utilizando um novo conjunto de dados, proveniented do setor financeiro. Além disso, estudámos o impacto da implementação de um novo modelo competitivo para representação de memória: Árvores de Decisão. Embora sem produzir melhorias significativas em relação às Redes Neurais Artificiais.

Numa segunda fase, criámos o NNE-TCP, que é um modelo original de Machine Learning que analisa ficheiros modificados numa dada alteração, quando ocorre uma transição do estado de um teste. O algoritmo aprende uma relação entre as duas entidades, mapeando-as em vetores multidimensionais, e agrupando-as por similaridade. Quando novas alterações são efetuadas, os testes com maior probabilidade de serem vinculados aos ficheiros modificados são prioritizados. Além disso, o NNE-TCP possibilita a visualização destas entidades em dimensões mais reduzidas, permitindo agrupamentos mais inteligentes. Ao aplicar o NNE-TCP, mostra-se pela primeira vez que a conexão entre ficheiros e testes é relevante e produz resultados de prioritização assinaláveis.

Esta investigação foi realizada no Instituto Superior Técnico em colaboração com o BNP Paribas, no âmbito do Gabinete de Transferência de Tecnologia do IST, tendo beneficiado de uma bolsa de estágio.

Palavras-chave: Integração Contínua, Prioritização de Testes, Reinforcement Learning, Embeddings, Dados.

Abstract

In modern software engineering, Continuous Integration (CI) has become an indispensable step towards managing systematically the life cycles of system development. Large companies spend vast amount of resources 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. Efficient continuous testing is a key component of frequent incremental software releases. By selecting the most promising test cases quicker, the round-trip time between making a change and receiving feedback is shortened, boosting productivity. To improve the efficiency, two Test Case Prioritization algorithms were developed.

First, we extended 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 a financial industry. Additionally, we studied the impact of experimenting a new competitive model for memory representation: Decision Trees, although without producing significant improvements relative to Artificial Neural Networks.

NNE-TCP is an original 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.

This research is carried out for Instituto Superior Técnico in collaboration with BNP Paribas. I have benefited from a fellowship of the BNP Paribas, in the framework of the IST Technology Transfer Office.

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

Contents

List of Tables

List of Figures

Nomenclature

Greek symbols

ϵ	Exploratory parameter.
π	Policy function.

Roman symbols

C	Commit.
CL	Change-List.
R	Reward.
T	Test-Suite
t	Test Case.
A	Action.
S	State.

Subscripts

t	Time instant.

Superscripts

fail	Failed tests.
pass	Passed test.

Glossary

APFD	Average Percentage of Fault Detection is a common evaluation metric for test schedules. Bounded between 0 and 1, upper values show that tests that will fail are prioritized first.
APTD	Average Percentage of Transition Detection is an original evaluation metric for test schedules. Bounded between 0 and 1, upper values show that tests that will transition are prioritized first.
CI	Continuous Integration. The practice of frequently committing software changes.
NAPFD	Normalized Average Percentage of Fault Detection, is the same as APFD but by only running a subset of all tests.
NNE-TCP	Neural Network Embeddings for Test Case Prioritization is the original machine learning framework defined in this work.
RETECS	Reinforced Test Case Selection is a reinforcement learning framework to prioritize relevant tests based on historical features.
TCP	Test Case Prioritization. Finding the optimal permutation of test cases, according to a given criteria

Chapter 1

Introduction

In this chapter, the motivation behind the work is introduced, key concepts are defined as well as a description of BNP Paribas and their testing strategy. Finally, the objectives of the work and thesis outline are presented.

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, especially in large and fast-paced companies. This is why many industries adopt a *Continuous Integration* (CI) strategy, a popular software development technique in which engineers frequently merge their latest code changes, through a *commit*, into the mainline codebase, allowing them to easily and cost-effectively check that their code can successfully pass tests across various system environments [1].

One of the tools used to manage software change is called regression testing. It is critical to ensure that the introduction of new features, or the fixing of known issues, is not only correct, but also does not obstruct existing functionalities. Ergo, regression testing plays a fundamental role on certifying that newer versions adhere to the mainline. Regressions occur when a software bug causes an existing feature to stop functioning as expected after a given change and can have many origins (e.g. code not compiling, performance dropping, etc.), and, as more changes occur, the probability that one of them introduces a fault increases [2].

As software development teams grow, identifying and fixing regressions quickly becomes one of the most challenging, costly and time-consuming tasks in the software development life-cycle, rapidly inhibiting its adoption. Such teams often resort to modern large-scale test infrastructures, like core-grids or online servers [3]. Consequently, in the last decades, there has been intensive research into solutions that optimize Regression Testing, accelerating fault detection rates either by alleviating the amount of computer resources needed or by reducing feedback time, i.e. the time delay between a software change and the information regarding whether it impacts the system's stability [4–8].

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 *apriori*, the likelihood a change has of being merge and/or a limited time-frame [4].

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 [9].

Concretely, in a supervised learning task, given a commit 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.

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 communicate the changes to the repository [1].

1.3 Mono-repositories vs. Multi-repositories

In order to manage and store the 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 [10]. 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, to 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. Also, the inter dependencies are reduced, providing more code stability, possibly accelerating development. However, the major con arises from necessity of synchronization, causing version problems and inconsistencies, for example when using two distinct projects that rely on different versions of 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 [11].

1.4 Continuous Integration

Continuous Integration (CI) is a software development technique where developers make frequent integrations of their work, usually daily. Then, each integration is verified by a build - a set of instructions to compile, test, inspect and deploy software than can be automated, requiring no human-intervention - to detect integration errors as quickly as possible, as other team members rely on the proper functioning of this system. One alternative to CI might be to only care about merging software versions at the very end of a project, but this approach has been proven to develop cohesive software faster and significantly reduce integration problems by dealing with them incrementally, avoiding *merge hell*.

CI is used to minimize risk: by integrating multiple times a day, the detection of defects is facilitated, i.e. there is a greater chance of finding a mistake when it is introduced, rather than in a later stage of the project. Software health attributes and statistics can be measured over time, such as complexity, enabling better project visibility and the number of assumptions can be reduced, because the same processes and scripts are used on a continuous base, e.g. the dependency on third party libraries.

By automating CI, repetitive labour is reduced, allowing developers to focus on more though-provoking tasks and at the same time, establish a consistent safety-net to allow for software cohesiveness.

One of the most important aspects of CI is the possibility to revert changes and to load the last stable version of the system, i.e. a version that had no integration issues, having tremendous impact on software deployment for companies. If malfunction changes in the software are not quickly detected and cannot be *rolled-back* immediately, products may suffer delays or , even worse, be released with defects, staining the company's credibility.

Overall, adopting CI practices provides a path of reduced risk to increase product confidence. By integrating software changes swiftly and detecting errors as quickly as possible, developers are more confident in introducing changes, because if something goes wrong, it is possible to return to the last checkpoint.

On the other hand, maintaining a healthy CI system can be troubling and some developers resist to adopt this practice for the increase of overhead time needed for maintenance. Also, some developers think that incremental changes lead to too much change. Consequently, there are many builds that fail, requiring immediate attention from the person responsible.

Then, finally, as teams as projects increase in size, more changes are made daily and subsequently, more computer power is needed to maintain the short time between making a change and receiving feedback, increasing costs [12].

1.5 Regression 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 other 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 and as the doll increases its size, even more pins are required. For example, running a battery of tests (test suite) where every single one of them yields PASS, it can only mean one of two things: or the code is immaculate or some scenarios were left out of the process. Usually, new test suites are constructed from failed tests. In a FAIL situation, i.e. fault detection or removal , a 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 [13].

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 ones. Usually this is done by applying test cases, to check if those features, in fact, work or are still work-

ing. Therefore, this field encapsulates the task of managing a pool of tests, that are repeatedly applied across multiple platforms [2].

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, hence 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 [2].

1.6 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 do not know which tests are more likely to uncover failures. It could be the case, that there is not even a single failure. Furthermore, a test failing previously is not a crystal clear indicator that it will fail again. Hence, with the rise of data availability and computer resources, there has been a growing interest to investigate solutions that involve learning from data [9].

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.

More and more, software is part of our daily life. Therefore there is the growing need of ensuring software quality. Although software systems are complex and a great variety of factors impact the development of reliable code, testing is the gold standard approach to assess quality. After decades of research, this task still remains a challenge and recently, Machine Learning leveraged strategies have populated the software testing world to circumvent some of the unresolved issues in the literature [9].

1.7 BNP Paribas

This research is carried out for Instituto Superior Técnico in collaboration with BNP Paribas, a leading bank in Europe, operating in 71 countries, employing over 198,000 people. It is one of the largest and first foreign banks to operate in Portugal (since 1985), gradually consolidating its presence. In Portugal, BNP Paribas offers a wide variety of services that cover a full range of areas: capital markets, structured finance, asset management, commercial banking and others [14].

1.8 Testing at BNP Paribas

The testing strategies at BNP Paribas described below are specific to a specific repository within Quantitative Research team. Data was collected over a period of 4 years, in a team that had around 90 developers.

BNP Paribas current test strategy is simple, all test case are run on a loop at regular interval (daily), without any specific test ordering. The specific commit causing a regression is obtained through bisection method with a complexity in $O(\text{nbCommitsPerDay})$. Daily status is usually fine but it might not be enough. On busy days, more than one commit can cause regression on the same tests and the bisection search will only identify the first commit which introduced a change. So there has been the increasing

interest in optimizing BNP Paribas' Testing system, in order to reduce the feedback time between making a change and receiving feedback.

Due to sheer volume of tests and the current testing strategy, to have feedback quickly, a grid of approximately 800 cores is used to execute tests constantly in parallel, being able to apply every single test case on the system within 2 hours.

The current approach to testing does not scale well, due to the $O(\text{NbTests} \times \text{NbDailyStatus} \times log(\text{NbChangePerStatus}))$ complexity, i.e. more and more resources will be needed, as teams and projects grow to obtain the same performance. Therefore, to expand the maximum capacity of the testing system, the goal of optimizing the current test strategy is not only focused on reducing feedback time, but also to achieve the same performance but using much less resources, which will drastically impact the company costs as well as encouraging a greener policy, reducing energy consumption.

Regression Testing optimization is also thought to boost productivity through a human component: if the feedback time is reduced and developers can grasp more quickly were and when a mistake was made, changes can be made more confidently, knowing that your modifications will not affect drastically the overall performance of the team, that is dependant on the common version of the system.

1.9 Repository

For transparency and replication purposes, the work developed throughout this thesis are stored and available online. Two Machine Learning frameworks were developed and a new dataset was made public to contribute to more data availabilty.

The first project is RETECS and it is available at `https://github.com/johnnylousas/RETECS` and the second is for Neural Network Embeddings, available at `https://github.com/johnnylousas/NNE-TCP`.

1.10 Objectives & Thesis Outline

The objectives delineated for this work are:

- Analyse how different system configurations affect Continuous Integration
- Optimize regression testing systems using real world data.
  - Extract relevant features from raw data
  - Build Machine Learning Models to learn from experience
- Improve the overall fault detection rate

The thesis is organized as follows:

- **Chapter 2** depicts the research landscape with the State of the Art: exploring how different CI configurations affect scalability, a strong background on Regression Testing techniques and, finally, how to leverage different Machine Learning techniques in these environments.
- **Chapter 3** corresponds to the first strategy implemented and corresponds to the first paper on Reinforcement Learning.
- **Chapter 4** explores the second study implemented with BNP Paribas data, which corresponds to a Neural Network Embeddings research paper.
- **Chapter 5** concludes the findings, describes the achievements and points to future research ideas.

Chapter 2

State of the Art

In order to depict the research landscape of this topic and due to the exploratory nature of the thesis, a description of how several CI strategies impact other companies is provided, as well as what are the most common regression testing techniques and, finally, the findings of other research publications are summarized, to serve as a starting point to find the most appropriate solution that fits our needs.

2.1 Continuous Integration Strategies

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.

2.1.1 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 Memon *et al.* [15] 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 [3].

2.1.2 Cumulative Approach - Current BNP strategy

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, e.g. 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, taking 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 the progress made during that working day possibly 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. Hence, 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. 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 if 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. Additionally, inside each commit, there can be tests more relevant than others. This information can be extracted and used to apply first the ones that maximize the probability of detecting a fault.

2.1.3 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 2.1.

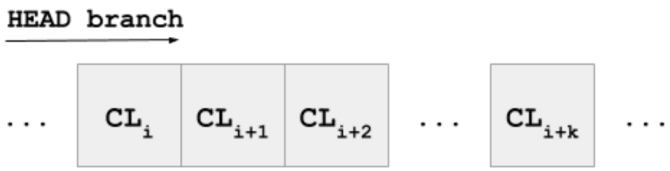

Figure 2.1: Code repository and workflow at Google. Each CL is submitted to master branch or HEAD [3]

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} [3].

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 [3].

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 [5] 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 [3].

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 [5].

2.1.4 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, [5] 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 2.2.

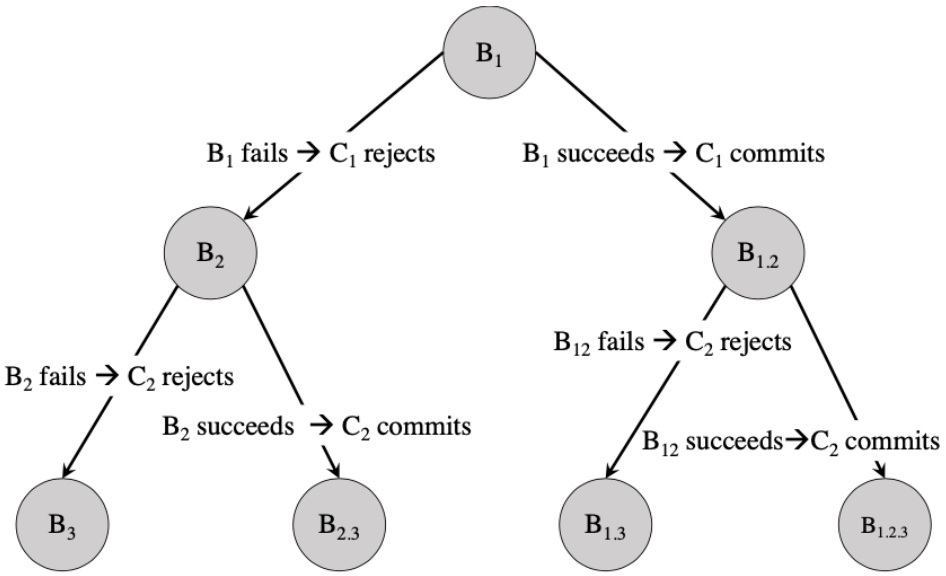

Figure 2.2: Speculation tree - builds and outcomes [5]

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 [5].

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 2.1: Comparative analysis between strategies

2.2 Regression Testing Techniques

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.

2.2.1 Test suite minimisation

Definition 2.2.1. 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) [16] .

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 [2].

2.2.2 Test Case Selection

Definition 2.2.2. 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 2.2.3. 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 $P(t) \neq P'(t)$ [17].

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 [2].

[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[16]

2.2.3 Test case prioritisation (TCP)

Definition 2.2.4. 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 [2]. 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) [18].

Definition 2.2.5. 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.[2]

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

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

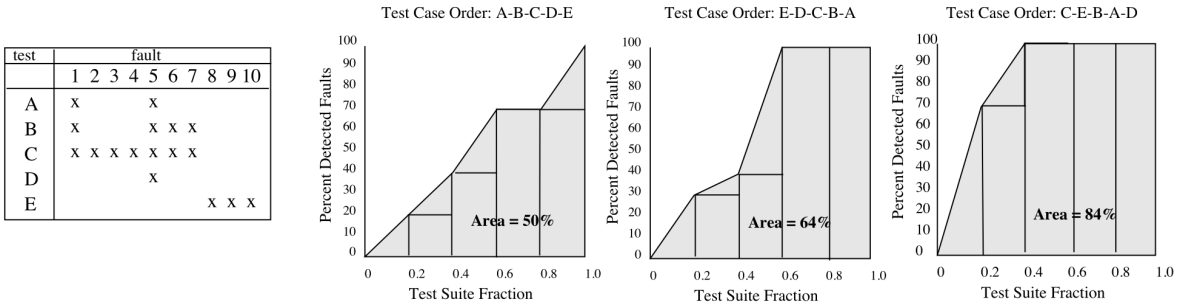

Figure 2.3: Average percentage of fault detection [18]

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 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 [5].

2.3 Test Case Prioritization Approaches

As mentioned before, Test Case Prioritization aims to rank tests according to their probability of detecting faults, such that the maximum number of defects is found in the shortest period of time.There are many approaches to define this probability, some examples rely on code-coverage, historical and similarity-based data. In recent years, with increasing data availability and computer power, Machine Learning approaches have gained popularity in the field of learning how to prioritize test cases. Hereafter, a summary of several techniques is provided to contextualize this work: first, the most important aspect is to identify relevant and expressive features and second, to choose the right Machine Learning Model.

2.3.1 Baseline

Baseline approaches to TCP correspond to simple ordering of test cases without considering any information of past commits and/or test executions, such as Random prioritizations and alphabetical orderings.

2.3.2 Coverage-Based

Code coverage represents a measure of the degree to which a test case exercises a software system [19]. A test case that executes a larger portion of the source code has higher coverage, which may

indicate higher failure probability. This approach was one the first attempts to prioritize test cases and entails a direct inspection of the code to be tested. The fundamental measures used in coverage-based techniques are the functions, branches and lines of code (LOC) [20]. There can be two types of coverage based techniques: total and additional coverage. The first focuses on covering the most portions of the system, according to some criteria, e.g. total faults or least computer resources spent, and the latter's intent is to achieve 100% in the least amount of time as possible.

2.3.3 History-Based

HIstory-based TCP relies on historical data of each test case, so that the ones with shorter time to execute and that have revealed faults in the past are executed earlier, following the hypothesis made by Marijan et. al [21]: a test case that failed in the past is more likely to fail again in the future. Nonetheless, the test case that fails in the current commit might not be the one that has failed previously. For example, test cases can also suffer modifications, if these modifications are small and are made when the test case was failing, there is a high chance that it will fail again, however , because this version of the test case was never run, there is no historical records, although it is quite similar to a recently failing test case. Additionally, more recent history may be more relevant than older history. If a test case has failed a long time ago, it is considered less relevant than one that has just failed [22].

2.3.4 Similarity-Based

Similarity based metrics are intended to prioritize test cases when they are similar to any of the test cases that have previously failed. Noor & Hemmati [23] define similarity based on test execution traces containing sequence of method calls. The methods called from previous test case executions are stored as historical information and it is later compared with test cases in the current commit, constituting the two inputs of the model. Subsequently, the relevance of the test cases is measured based on the returned similarity values and those with higher values are prioritized. Some examples of similarity functions are *Basic Counting* which returns the number of common method calls between two versions; *Hamming Distance* is the minimum number of edit operations, i.e. insertions, deletions and substitutions, required to transform the first sequence into the sequence.

2.4 Machine Learning applied to TCP

2.4.1 Supervised Learning

Supervised Learning represents the largest slice of ML algorithms used in Test Case Prioritization. According to Durelli *et al.* [9], from an universe of 50 different studies, 41 were leveraged by Supervised Learning. The most common type of algorithms that fall into this category are: Artificial Neural Networks, Bayesian, Decision Trees, Ensemble methods, Nearest Neighbors and Regression.

The goal of Supervised Learning is trying to learn from known examples and in the context of TCP, we want to learn which test cases are likely to fail, based on past events.

Busjaeger & Xie [24] were determinant in applying test prioritization in Industrial environments, by resorting to a Support Vector Machine Algorithm that is fed with coverage, text and history features, such as: Code coverage provided by Java, textual similarity of tests, failure history and test age. The authors claim their achievings clearly outperforms previous Machine Learning approaches.

Noor & Hemmati [23] propose a Logistic Regression Model to predict failing test cases based on test quality metrics. They measure code coverage, previous fault history and similarity based quality metrics. After calculating the likelihood of failing for each test case, they are ranked in descending order of probability.

Later, Palma *et al.* [4], extending the work described above, extended the similarity-based features formulated and manually combined them into more expressive ones, showing the importance of data transformation to facilitate ML algorithms learning. Similarly, a Logistic Regression Model was built and the same datasets were studied. The authors claim that the findings of the first research study still hold and that their improvement produced a positive impact in the evaluation metrics.

Oftentimes, Supervised Learning is used as a comparison method to other, more complex, models [4, 6, 25].

2.4.2 Unsupervised Learning

Chen *et al.* [26] claim to be the first to develop a framework to effectively prioritize test cases based on cluster test selection, by grouping tests into clusters and measuring the distance between them. In this context, the authors raise the hypothesis that tests that belong to the same cluster, have similar behaviour and characteristics. Usually, clustering falls in the realm of unsupervised learning, that takes unlabelled data and attempts to organize it. However, labelled data is a crucial step for algorithms to learn, but often limited or unavailable. The authors introduce a semi-supervised clustering approach (semi-supervised KMeans), where only a subset of tests are labelled, to enhance the clustering solution. The features used to build cluster are *pair-wise* links between any two test cases, extracted from previous test case executions.

2.4.3 Deep Learning

Deep learning has been one of the most promising research areas in machine learning, developing efforts in image processing and speech recognition, particularly. In the context of CI, Yang *et al.* [6] bridge the gap between just-in-time defect prediction and deep learning techniques. Rather than focusing on test cases that are likely to fail, the author focus on detecting defective changes, i.e. faulty commits, to ensure software quality. This work is an extension of Kamei *et al.* [27] that implemented a Logistic Regression model to the same datasets.

The authors propose a framework called *Deeper*, that has two phases: first, a feature selection phase with the goal of extracting more expressive features from the initial dataset, by implementing a Deep Belief Network - a complex machine learning algorithm that stacks several Restricted Boltzmann Machines, whose inputs are the initial features and the output are more complicated non-linear combinations of the input. The second phase takes the output of the Deep Belief Network and feeds them to a Logistic Regression Classifier to predict if, whether or not, a given commit is faulty.

Some examples of the features the authors considered relevant are the number of modified files and directories, the lines of code added and deleted, the developer experience and the average time between the last and current change. The results of this work show significant improvements relative to the first approach developed by Kamei *et al.* [27] without Deep Learning.

More recently, Zhou *et al.* [25] proposed *gcForest* - a new deep learning framework that provides an alternative to Artificial Neural Networks - that has a multi-layer structure, where each layer contains a Random Forest. The authors claim that gcForest is able to generate even more expressive features from the initial dataset, taking full advantage of ensemble methods and deep learning.

Yang *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.

2.4.4 Natural Language Processing (NLP)

Lachmann et al. [28] claim to have been the first to employ a supervised learning and NLP technique to rank test cases, without code access. The authors resort to available test case descriptions, written in natural language and create a dictionary containing all words that occur within all test cases. Then, combining features such as: number of failures detected over time, test execution cost (duration) and whether a given requirement is fulfilled and the test case description in natural language are fed to a Support Vector Machine Classifier that learns how to rank test cases, according to the computed probability. The authors claim their framework considerably surpasses random and human prioritizations.

Recently, Sharifi & Hemmati [29] implement a NLP framework that vectorizes the text present in the test case itself and combines this information with history-based features. By using NLP it is possible to extract useful information from free text. Two main techniques in NLP are *Part-of-Speech (POS)* and *Term Frequency-Inverse Document Frequency (TF-IDF)*.

The first focuses on tagging each word in a sentence with respect to a specific syntactic category, e.g. nouns, verbs, adverbs, etc. POS constitutes a crucial preprocessing technique to help algorithms distinguished valuable information. TF-IDF is a common metric to keep track of the frequency a given term appears in a document, with the twist of being normalized by the total number of terms, so that very frequent words are not given to much relevance. Then the features are fed into a Linear Regression and

Artificial Neural Network algorithm to predict the probability a test case has of failing. The authors show this approach is able to predict failures on manual test suites from Mozilla Firefox.

2.4.5 Reinforcement Learning

More recently, Spieker et al. [30] introduced RETECS, which is a Reinforcement Learning agent that selects and prioritizes most promising test cases, based on their duration, last execution and failure history. Initially, the agent has no initial concept of the environment it interacts with, so an exploration policy is encouraged, collecting reward signals as feedback. The agent has online learning properties, i.e. incrementally learns from experience, adapting to the CI environment and scheduling meaningful test executions. RETECS offers a lightweight alternative to similar approaches, while using only one source of data, namely failure execution [30].

Also, Wu et al. [22], inspired by the work described above, developed a novel reward function for Reinforcement Learning in the context of TCP, that can be adjusted according to the frequency of software changes. The work developed by Spieker et al. [30] treats all failure history results similarly, recent or not, meaning that a more recent test execution is as important as one that occurred a long time ago and, therefore, should be less relevant. Wu et al. [22] implemented a parametrized Time-Window reward function that allows to configure a small window, when the development pace is slow, e.g. time in between projects, holidays, and a larger window for fast paced situations, e.g. the delivery of a product.

2.5 Other Approaches

2.5.1 Commit-based Approach

So far, existing prioritization techniques focus on reordering test cases, while maintaining the same order between commits. Liang *et al.* [8] claim that *intra-commit* prioritization - re-order test cases within a commit - rarely produce substantial impacts on fault detection, but rather *inter-commit* prioritization have potential for greater improvements. The authors propose a Continuous Commit-Based Prioritization framework to prioritize and re-prioritize commits, using historical information to calculate expected failing probability. The results obtained by the authors show major improvements, especially in large and fast paced datasets, while being lightweight and straight-forward.

2.5.2 Transition-based Approaches

Until now, previous research tends to focus on detecting failures, rather than trying to detect transitions. A transition is defined as a change in the status of a test case, either from Pass to Fail or Fail to Pass. If transitions are found quickly, failures will also be found quickly. Leong *et al.* [31] developed a fast, lightweight and offline framework powered by historical features to detect transitions on real-world environments. The authors claim that the number of times a test has been triggered, the number of authors committing code that triggers a test and the recent execution history are three strong indicators for test prioritization, although it is said that research remains open. Additionally, it is stated that identifying transitions lies at the core of CI related activities. By informing developers as quickly as possible when committed changes break or fix the last version of the system.

2.6 Test Case Prioritization in Industrial Environments

The gap between academic research and industry is still significant, as implementing test prioritization in industrial environments represents an additional challenge in terms of heterogeneity, scale and cost [24]. There is the additional challenge of data availability and correctness in this type of environments, that, although diverse, may not fit the models proposed in the literature.

Some successful examples of healthy and scalable CI systems belong to Google [3] and Uber [5] (described in detail above). Also, Busjaeger & Xie presented a study applied to SalesForce.com data and Marijan *et al.* [21] applied TCP to an industrial video conferencing system.

Chapter 3

Study I: History-based Reinforcement Learning

In this section, we adapt the algorithm developed by Spieker *et al.*, called Reinforced Test Case Selection (RETECS). First, the background and formalism used throughout the section are defined. Then, a description of how the method works and the modifications made is provided and finally, the experimental evaluation is made with our dataset.

3.1 Background

RL is an adaptive method where an agent learns how to interact with an environment that responds with reward signals, which correspond to the feedback of taking a certain action. For example, when driving a car, the environment is the real-world, the state can be defined as the position, speed of the car and neighbouring cars. Possible actions are turning the wheel and accelerating or braking. The reward function can be calculated by how much time it took to reach the destination, while respecting traffic rules.

These back-and-forth interactions take place recurrently: the agent receives some representation of the environment's state and selects actions either from a learned policy - mapping from perceived states of the environment to actions to be taken when in those states - or by random exploration - randomly choosing an action when in a given state to account for environment changes. Consequently, the environment responds to these actions and presents new situations to the agent, which finds itself in a new state after exercising its action. The main goal of RL is to maximize the cumulative sum of rewards, rather than just consider immediate rewards [32]. Following the example above, a possible action to take is to increase the travel speed of the car. If there are no accidents until the final destination, then the reward receive will be higher.

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

3.1.1 Formalism

This section provides the necessary formalism for the Test Case Prioritization problem studied throughout this work.

The test pool is defined as T and is composed by the set of test cases $\{t_1, t_2, ..., t_N\}$. For each commit C_i, a test suite $T_i \subset T$ can be selected and ordered.When there is no time or resource restriction, T_i can be defined to contain the ordered set of the entire test pool, T_i^{all}. When such restrictions come into play, a subset of the test pool T is selected for T_i. Note that T, being the test pool, 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: $rank : T_i \rightarrow \mathbb{N}$, where $rank(t)$ is the index of test case t in T_i.

Each test case t contains information about its duration, $t.duration_i$, which is known before execution, and its test status, $test.status_i$, only known after execution, which 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 $T_i^{fail} = \{t \in T_i \; s.t. \; t.status_i = 0\}$. It should be noted that in real world scenarios there are situations where the cause of failure of a test case is due to exogenous reasons such as randomness or hardware problems, so tests can have multiple status at C_i, a phenomenon known as *test 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: the failure of a test case can be caused by one or several faults in the SUT, and a single fault can cause several test cases to fail. In this study, because there is no information available on the actual faults, only the status of each test case is considered.

3.2 RETECS

In the context of TCP, RETECS prioritizes each test case individually and, with these prioritizations, a test schedule is then created, executed and finally evaluated. Each state represents a single test case $t_i \in T_i$, and it contains information on the test's duration, when it was last executed and the previous execution results. The set of possible actions corresponds to the set of all possible prioritizations a given test case can have in a commit, which is translated into an integer, e.g. if test A has prioritization equal to one, then it will be the first to be executed. After all test cases are prioritized and submitted for execution, the respective rewards are attributed based on the test case status. From this reward value, the agent can adapt its strategy for future situations: an action yielding positive rewards is reinforced, whereas negative rewards discourage the current behaviour. The agent-environment interface is depicted in Fig. 3.1.

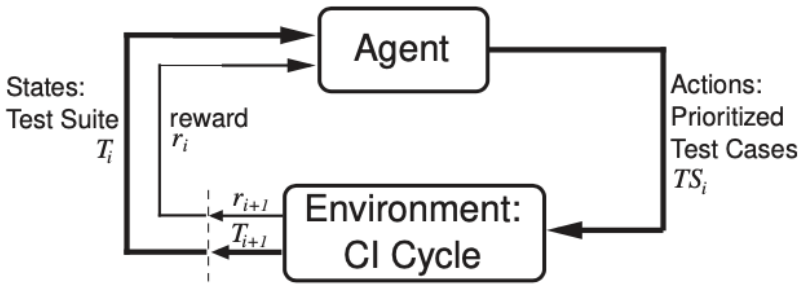

Figure 3.1: Reinforcement Learning applied to TCP cycle of Agent-Environment interactions (adapted from [32])

The RETECS framework has the following characteristics:

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

3.2.1 Reward functions

In a RL problem, the goal consists of collecting numerical rewards that measure the performance of the agent at performing a given task. Hence, properly defining a reward function that reflects these goals will steer the agent's strategy towards optimality. We now use 3 reward functions defined by Spieker *et al.* [30], namely Failure Count, Test Case Failure and Time-Ranked, defined as

Failure-Count Reward

$$reward_i^{fail}(t) = |T_i^{fail}| \quad (\forall_t \in T_i), \tag{3.1}$$

The first reward function awards all test cases with the number of failed test cases that were executed. For this reason, the RL agent is directly encouraged to maximize the number of failed test cases

selected. 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 a given failure, we take the risk of favouring undesirable 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.

Additionally, this reward is only suited when there is a time or computational constraint, in which case we need to do Test Case Selection, because the reward signal is the same for every test case that is scheduled for execution.

Test-Case Failure Reward

$$reward_i^{tcfail}(t) = \begin{cases} 1 - t.status_i, & \text{if } t \in T_i \\ 0, & \text{otherwise} \end{cases}, \tag{3.2}$$

The second reward function bridges the gaps identified with the Failure Count reward function in terms of refinement, by rewarding each test case individually, based on its status after execution. If a test case is scheduled and it's execution produces a fail, a positive reward will be assigned. On the other hand, if a passing test case is scheduled, the reward value is equal to zero, its inclusion in the test schedule is not reinforced nor discouraged.

Finally, it's worth mentioning that 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, which is a limitation, since a failing test might be run only at the end of the test schedule.

Time-Ranked Reward

$$reward_i^{time}(t) = |T_i^{fail}| - t.status_i \times \sum_{\substack{t_k \in T_i^{fail} \wedge \\ rank(t) < rank(t_k)}} 1, \tag{3.3}$$

where $|T_i^{fail}|$ is the number of failing test cases.

The last reward function we consider in this work takes into account the order of test cases. It not only considers a test's status, but also its rank on the schedule. An ideal test schedule would prioritize failing test cases, placing them at the beginning, followed by the passing test cases. This reward function explicitly penalizes passing test cases by the number of failing tests ranked after them, while for failing test cases, a time-ranked reward reduces to the failure count reward.

3.2.2 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, though it is not annulled. The exploration rate can be tuned depending on how dynamic the environment is. This 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.

3.2.3 Memory Representation - Value Functions

In the context of TCP, RETECS prioritizes each test case individually and a test schedule is then created, executed and finally evaluated. Each state represents a single test case and it contains information on the test's duration, when it was last executed and the previous execution results. The set of possible actions corresponds to the set of all possible prioritizations a given test case can have in a commit, assigned by the RL agent. After all test cases are prioritized and submitted for execution, their respective rewards are attributed based on the test case status as feedback, assessing performance. From this reward value, the agent can adapt its strategy for future situations: an action yielding positive rewards is reinforced, whereas negative rewards discourage the current behaviour.

When training RL algorithms, in any given state we need to estimate what rewards will be received in the future if each of the available actions is chosen, so we can select the action that maximizes future rewards [32]. These estimates are calculated by defining a *value-function* $v_\pi(s)$, where s is the state, with respect to the learned policy π and can be learned from experience.

In tabular representation, there are two options: either the agent finds itself in a new state and registers the obtained value of that state or it finds itself in a state where it had been before and an average of the value is calculated. Multiple encounters of the same state will lead to a more accurate representation of that state's actual value. However, with an increasing number of states it may not be feasible to keep track of each state individually. In such a case, $v_\pi(s)$ could be a parameterized function, $v_\pi(s, \mathbf{w})$, where $\mathbf{w} \in \mathbb{R}^n$ is the parameter vector. From now on, we say that $v_\pi(s) \approx v_\pi(s, \mathbf{w})$ for the approximated value of a state s given weight vector **w** that should be adjusted with experience in order to match observations.

These parameterized functions are called *approximators* and, based on the returns of observed states, are able to generalize to unseen ones. This generalization significantly reduces the amount of memory needed to store information about each state, as well as the time required to update it, even though these new values correspond to estimations rather than the actual observed values.

The topic of function approximation is an instance of *supervised learning*, because it gathers observed examples and attempts to optimize parameters to construct an approximation of an entire function [32]. A valid example for such a function are ANN's where the parameters to be adjusted are the network's weights, which from now on will be referred to as Network Approximator,. The downside of Network is that a more complex configuration is required to achieve higher performance [32].

In this work, we additionally implement a DT Agent to approximate the value function. Fig. 3.2 shows how a DT 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. The RL agent is then able to learn the values received from taking each path/actions and where they will lead [33].

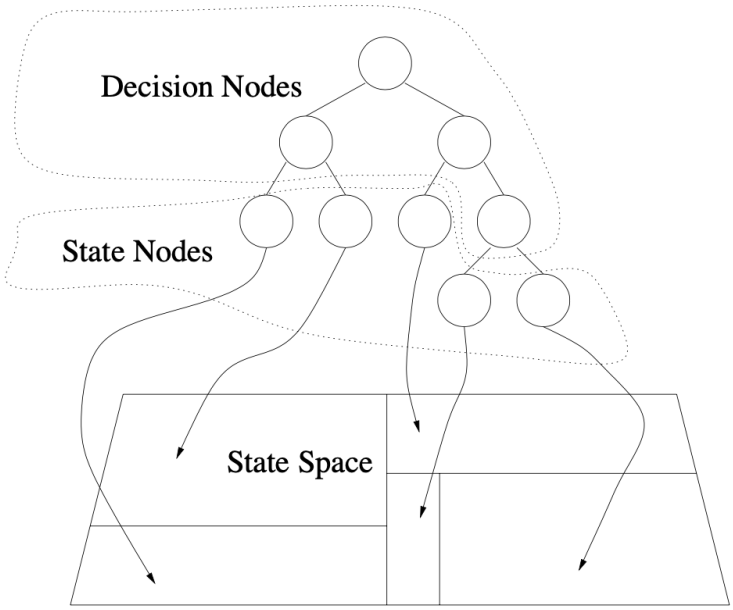

Figure 3.2: Represent state space regions with DT's (Adapted from [33])

By computing the $v_\pi(s)$ with a DT, $\mathbf{w})$ represents all the parameters defining the splitting points and leaf values of the tree. Commonly, the number of parameters n - length of vector $\mathbf{w})$ - is much lower than the number of possible states and tweaking the value of one parameter will affect the value of many states. An action will force the agent to change its state, updating its particular value and consequently, updating **w**, which will generalize from that state to affect the values of many other states.

Finally, the algorithm will generalize better or worse depending on convergence. For example, many supervised learning models' goal is to minimize the root-mean-squared error (RMSE), in this case between $v_\pi(s)$ and $v_\pi(s, \mathbf{w})$ [32].

3.3 Experimental Evaluation

The next section presents the application and evaluation of our framework, describing, first, the setup procedures as well as a description of the datasets used (section 3.3.1). We then provide an overview of possible evaluation metrics to assess the framework's performance 3.3.2. To maximize these metrics, fine-tuning is used to find the best combination of parameters, which is shown in section 3.3.3. Finally, in section 3.3.4, the results obtained are presented and discussed, according to the research questions formulated below. Finally, threats and future work are discussed to close the evaluation process.

RQ1: How will RETECS behave in the presence of a novel dataset with different characteristics?

RQ2: Which function approximator yields better performance? We compare two different models: Artificial Neural Networks and DT's.

RQ3: How is RETECS model performance compared to traditional prioritization techniques, in this new context?

3.3.1 Data Description

The data used in this work corresponds to industrial real-world environments used by Spieker *et al.* and Wu *et al.* [22, 30], from ABB Robotics Norway [1], *Paint Control* and *IOF/ROL* which test complex industrial robots. The novel dataset belongs to an investment bank *Finance*, acting in the financial sector. Each dataset contains historical information of test results, around 300 commits, and has different characteristics. Table 3.1 summarizes the 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 %
Finance	1,379	303	417,837	63.87 %

Table 3.1: Dataset Statistics for *IOF/ROL*, *Paint Control* and *Finance*

The datasets are alike in number of commits, but it is clear that the testing strategy is distinct. 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 Finance 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.

3.3.2 Evaluation Metric

The metric to evaluate the framework's performance is the NAPFD (Normalized Average Percentage of Fault Detection), as defined by Spieker *et al.* [30] and it represents the most common metric to assess the effectiveness of test-case prioritization techniques together with test case selection, that occurs when there is a time limit associated with testing. It is defined as

$$\text{NAPFD}(T_i) = p - \frac{\sum_{t \in T_i^{fail}} rank(t)}{|T_i^{fail}| \times |T_i|} + \frac{p}{2 \times |T_i|}, \tag{3.4}$$

$$\text{with } p = \frac{|T_i^{fail}|}{|T_i^{totalfail}|}, \tag{3.5}$$

where $|T_i|$ is the number of test cases. Furthermore, the higher its value, the higher the quality of the test schedule will be: 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.

In this work, similarly to the original authors, a time limit was imposed, corresponding to 50% of the total time spent if all test cases were applied.

It should be also noted that NAPDF is commonly used in its 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 subset, such that when $p = 1$, all possible faults will be detected and NAPFD reduces to APFD.

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

3.3.3 Fine-Tuning

Parameter tuning allows us to find the best combination of parameters that maximizes 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.* [30] was used, in order to replicate the same results for the Network Agent, which corresponds to using 12 nodes with one hidden layer.

ANN and DT Tune

For the Finance dataset the architecture of the hidden layer for the Network Agent was studied, by calculating the NAPFD with different configurations, as depicted below.

The optimal value used for the other datasets was found by Spieker *et al.* [30] and is equal to 12 nodes with one hidden layer. In Fig. 4.1, 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.

The parameters to be tuned in Decision Trees are: (1) *criterion*, (2) *maximum depth* and (3) *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 of samples 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 Finance dataset.

From Fig. 4.1, 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 a minimum number of samples of 3. This configuration is preserved throughout the following experiments.

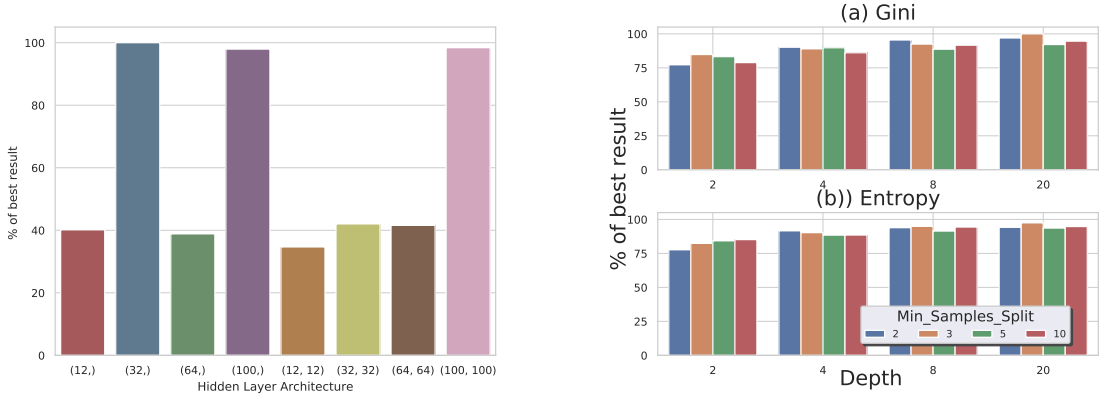

Figure 3.3: Both plots use the Finance dataset and give the $\%$ of best result, i.e best NAPFD. The left plot shows different Hidden Layer architectures and the right plot the Gini and Entropy criterion, as a function of Depth (x-axis) and each bar corresponds to a different value of minimum of samples to split.

History-Length

History-Length parameter 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 to 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. Fig. 3.4 studies how different history length values affect the RL agent, for the Finance dataset. There does not seem to exist any clear relationship between the two quantities depicted in Fig. 3.4. The best result obtained corresponds to a history-length of 25 executions - as well as 29 but the first one is preferable, so that efficiency is not compromised. This is

disparate from the optimal history length obtained by Spieker *et al.* [30] for the other two datasets, which is 4. This reinforces the fact that *Finance* data has dissimilar characteristics from the other datasets.

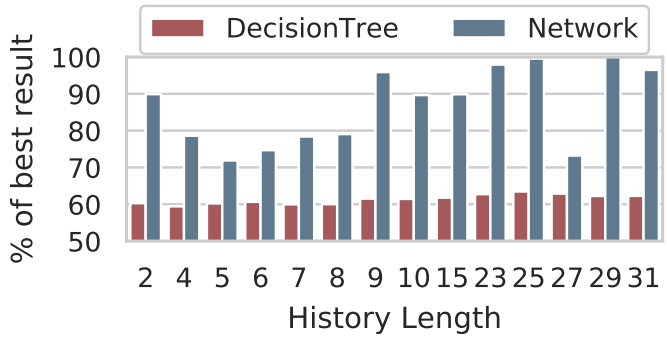

Figure 3.4: History Length Analysis for the Finance Dataset for two ML algorithms. Best Result is 25 executions.

3.3.4 Results

In our experiments, we trained two RL agents. The first resorts to an Network representation of states, while the second uses a DT. On both cases, the reward function varies between failure count, test-case failure and time ranked. For each test agent, test-cases are scheduled in descending order of priority and until the time limit is reached, if there is one. Traditional prioritization methods were included as a means of comparison: *Random*, *Sorting* and *Weighting*.

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

Due to the fact that RETECS learns incrementally as it is trained, the evaluation metric NAPFD is measured on each commit and due to the exploratory nature of the algorithm, we iterate through the experiments 30 times in order to capture the randomness, averaging out the exploratory nature of the algorithm. Unless stated otherwise, reported results show the mean value of all iterations. For reproducibility, our contribution to RETECS is implemented at *https://github.com/jlousada315/RETECS* in a publicly accessible repository, in Python, using *Scikit-Learn's* toolbox for ANN's and DT's.

3.3.5 RQ1 & RQ2

Fig. 3.5 shows a comparison of the prioritization performance between the Network Agent and the DT Agent, with regards to different reward functions (rows), applied to three different datasets (columns). The commit identifier is represented in the x-axis and for each one there is a corresponding NAPFD value, ranging from 0 to 1. (represented as a line plot in red and blue for the Network and DT agents, respectively). The straight lines show the overall trends of each configuration, which is obtained by fitting a linear function - full line for Network and dot-dashed line for DT Approximator. It is worth noting that for *IOF/ROL* and *Paint Control*, with the Network Approximator, we replicate the results from the original paper [30], to make sure no errors were introduced while modifying the code.

It is noticeable that both the approximator used for memory representation and the choice of the reward function have a deep impact on the agent's ability to learn better prioritization strategies. For the Failure Count and Time-Ranked reward functions, both approximators go hand in hand and present

similar trends, i.e. for a given dataset and reward function, both decrease or increase in approximately the same amount.

However, this behaviour no longer holds when looking at the Test Case Failure reward function. This is the function that evidently produces the best results, in terms of maximizing the slope of the NAPFD trend. When combined with the Network Approximator, this approach proves to be the best configuration overall, for the three datasets, where we see a more significant growth in the trend line, indicating that the algorithm is learning from the data. This implies that attributing specific feedback to all test-cases individually enables the agent to learn how to effectively prioritize them and adapt to heterogeneous environments.

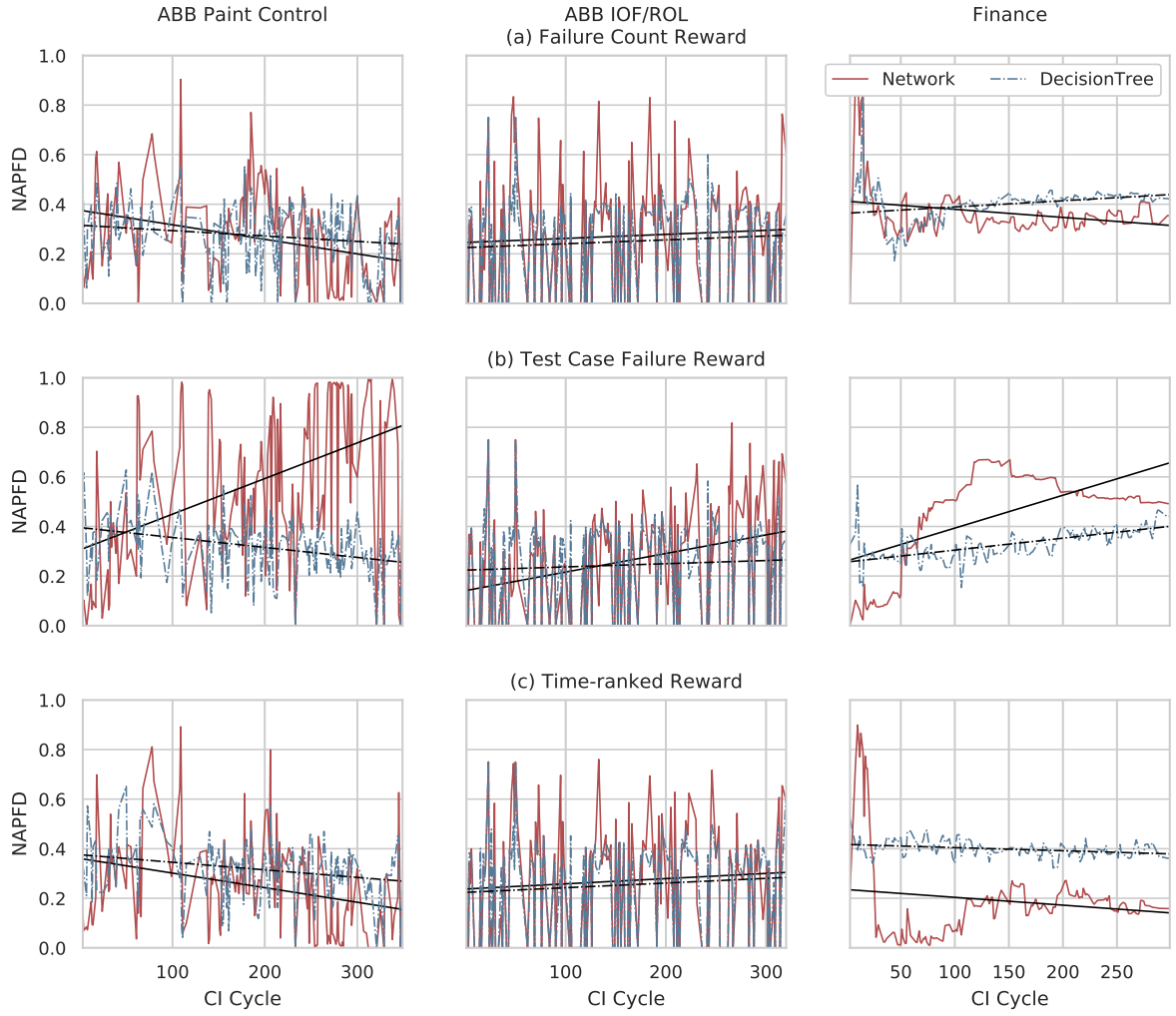

Figure 3.5: NAPFD Comparison with different Reward Functions and memory representations: best combination obtained for Test Case Failure reward and Network Approximator (straight lines indicate trend)

Another aspect reinforcing the notion of heterogeneity is the differences in the fluctuations of each dataset. For the first two datasets, we can clearly see fluctuations in the results. Spieker *et al.* [30] 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 tests that were added manually and produced cascading down failures. Notwithstanding, this behaviour is not observed for the *Finance* dataset, which suggests a much more stable system with less fluctuations, so there is a stronger indication that, with the right set of features, a crystal-clear relation between test cases and their probability 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 DT Approximator was able to surpass its performance by a small amount. If, for example, the *Finance* dataset had more records, it is possible that DT would follow the growing trend and surpass Network by a significant amount. Therefore, the collection of more

data is crucial to correctly evaluate the DT Approximator's performance.

Choosing the best configuration, test case failure reward and the Network Approximator, when RETECS is applied in an environment completely different from Robotics and with different characteristics, it was able to adapt and learn how to effectively prioritize test cases. This shows that the RETECS domain of validity expands to distinct CI environments, which is particularly useful for companies that increasingly rely on the health and proper functioning of these systems.

3.3.6 RQ3

The focus of RQ1 and RQ2 was to discover which combination of components would maximize performance, which we found to be Test Case Failure Reward and the Artificial Neural Network Approximator. Now, with RQ3, our aim is to compare this approach to traditional test case prioritization methods: *Random*, *Sorting* and *Weighting*. The results are depicted in Fig. 3.6 as the difference between the NAPFD for the traditional methods and RETECS over several CI Cycles. Each bar comprises 30 commits. For positive differences, the traditional methods have better performance, and on the contrary negative differences show the opposite. Due to the exploratory character of the algorithm, it is expected that at the beginning, the traditional method will make more meaningful prioritizations and this trend is verified in all datasets, although more evidently in *Paint Control* and *Finance*.

For the *Paint Control* dataset there are 2 adaptation phases: first, there is a steep convergence in the early commits, with the RETECS method needing only around 60 commits to perform as well as or even better than the traditional 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 the *IOF/ROL* dataset, it is evident that the results were inferior to *Paint Control*, having small increments on performance as was expected from the analysis of Fig. 3.5. Comparison methods appear to have slightly better performance during the first 200 CI cycles and afterwards RETECS seems to converge to a similar performance. This is an evidence of slow convergence, as it is clear in the second column of Fig. 3.5, and that more records are needed to possibly surpass the performance these methods.

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

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

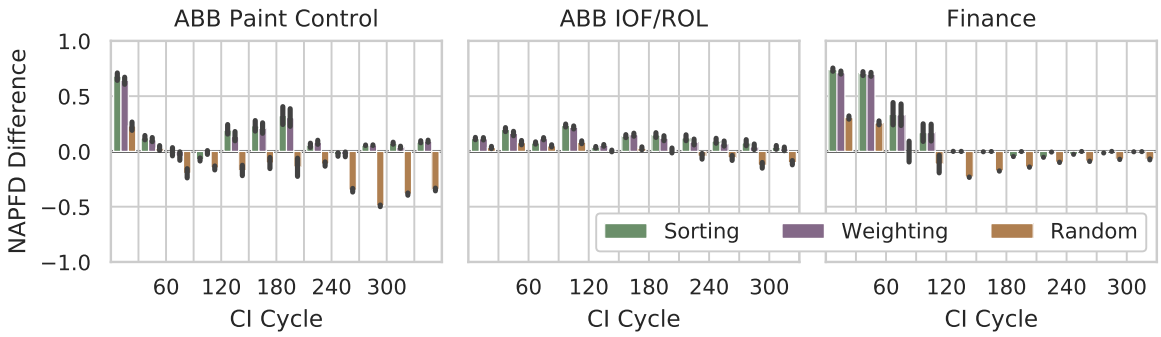

Figure 3.6: NAPFD difference in comparison to traditional methods: Random, Sorting and Weighting. Positive differences indicate better performance from traditional methods and negative differences show better performance for RETECS

3.3.7 Threats to Validity

Internal

RETECS is not a deterministic algorithm and because of its 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 of those results. But it could be the case that a subsequent implementation of the method fails to produce the same results observed in this work.

The second threat is related to parameter selection, that due to limited computer power, should have been more extensive. Thus, the chosen parameters are most likely 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.* [30], there's a threat related to implementation issues. Machine learning algorithms were implemented with the *Scikit-Learn* library and the framework is available online for validation and reproducibility, in the aforementioned *Github* repository.

External

The main gap related with external threats, pointed out by Spieker *et al.* [30], was the fact that the inclusion of only three datasets was not representative of the wide diversity of CI environments. In the original paper three datasets are used, namely *IOF/ROL*, *PaintControl* and *GSDTSR* from Google. The latter was not considered in this study due to its size and our inability to train it in feasible time. 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 faults are considered indistinguishable and interchangeable, yet this is not always true. For example, two test cases can fail due to the same fault, and, vice-versa, one failed test-case can reveal two or more faults. Nevertheless, because information about total faults is not easily accessible, the assumption that each test case indicates a different fault in the system is formulated.

Regarding function approximators, there are many other machine learning algorithms that could be tested and fine-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), it 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, such as 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 considered, including other machine learning methods.

Chapter 4

Study II: Neural Network Embeddings

In this section, our original methodology to find meaningful test case prioritization is presented in detail. First, with the theoretical foundations of embeddings, followed by an extensive step-by-step description of the model - from data cleaning to evaluation. Then, we evaluate it against BNP Paribas' data and compare it to traditional prioritization methods. Finally, the results and threats to validity are discussed.

4.1 Background

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

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

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

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

Embeddings are a very common practice when working with text and modelling language, becoming Word Embeddings - efficient and dense representation where similar words will have similar encodings [34]. When parsing through a document with text, embeddings can be used to determine which is the most likely topic approach in that document - topic modelling - and also to determine the state of mind of someone that wrote it - sentiment analysis. For example, *Word2Vec* word embeddings map each word to a vector, based on a neural network that was already trained on millions of words [35]. Afterwards it is possible to make numerical operations with these vectors, one very common example is the equality: *king - man + woman = queen*.

4.1.1 Neural Network Embeddings

In our case, we take each file and each test and represent them as n-dimensional vectors, with the goal of representing similar files and similar tests as similar vectors. The key aspect of embeddings is that these n-dimensional vectors are trainable, which means that each vector component can be adjusted in order to push vectors representing of related objects together. As a result, after training, the supervised learning task will be able to predict whether two categories are similar. Following the example above, it is possible to learn to map the variable *color* to a lower dimensional space than three dimensions, e.g. *red*$= [1, 0]$, *blue* $= [0, 1]$ and *purple* $= [1, 1]$, by taking advantage of the fact that *purple* is a combination of *red* and *blue*.

4.2 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.

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

The implementation of this algorithm was done with the Python Deep Learning Library - Keras [36]. 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.

Load and Clean Dataset

The dataset should contain records of the files modified in every commit, 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 will inevitably have noise. For example, in a dynamic software development environment, files can become outdated, deprecated, duplicated or renamed.

For this reason, eliminating redundant files and tests as well as removing files and tests that have not been modified or transitioned recently are necessary steps to obtain a cleaner dataset. It's worth mentioning that another source of noise in the data arises from the fact that some files and tests will be linked by chance and will not correspond to actual connections. For this reason, we need to further remove these connections, a step that will be described later.

Create Training Set

After loading and cleaning the dataset, in order for the model to learn the supervised learning task, it needs to be trained. Like any other Machine Learning problem, the dataset must be split into two sets: training, for the algorithm to learn from known examples, and testing, used to obtain an unbiased evaluation of the trained model. The size of each set can vary, depending on the dataset size and characteristics. In this work, we used a ratio of $80/20\%$ between the training and testing sets, in order to maximize the amount of data available for training, without compromising an unbiased evaluation.

The problem we are trying to solve can be stated as: given a file and a test case, predict whether the pair is linked, i.e. predict if the modification of a given file could impact the outcome of a given test case's execution, based on the commit history. Subsequently, the training set will be composed of pairs of the form: $(file, test, label)$. The label will indicate the ground-truth of whether the pair is or is not present in the data. In any of the commits, if there is a test case that suffered a transition where a given file was modified, then that file and test case constitute a positive pair.

To create the training set, we need to iterate through each commit and store all pair-wise combinations of files and test cases. However, there will be commits in which many files were modified, or where many test cases suffered transitions, which means that the training set will contain many false-positive pairs, i.e. positive pairs that do not represent an actual file-test link. This aspect can be mitigated by removing from the training set pairs that only occur a very small number of times, while preserving those that reoccur repeatedly throughout the commit history.

Because the dataset only contains positive examples of linked pairs, in order to create a more balanced dataset we need to generate negative examples, i.e. file-test pairs that are not linked. A negative example constitutes test case that is not linked to a modified file. This occurs when a certain file is modified, but a test case status remained unchanged, i.e. it did not transition. It's important to emphasize that, in a given commit, the files that were not modified in that commit and the test cases that transitioned do not constitute a negative pair, since there is no guarantee that if that file were to be modified, it would not impact the test case in question. For this reason, we only consider negative pairs of the form: (file modified, unaltered test case). Thus, to keep a balanced dataset, for any given commit with a certain number of positive pairs, we randomly pick a proportional number of unique negative pairs.

In our dataset, for the positive samples, the $label$ is set to 1, whereas for negative examples, the $label$ is set to 0 when the task at hand is classification or -1 when we are using regression, for reasons that will become apparent later.

As a result of having to create balanced examples for every commit and specially when dealing with large datasets, it could become unpractical, in terms of memory and processing power, to generate and store the entire training set at once. Consequently, to alleviate this issue, the *Keras* class *Data Generator* offers an alternative to the traditional method, by generating data *on-the-fly* - one batch (i.e. subset of the entire dataset) of data at a time [36].

In our case, each batch corresponds to the pairs that resulted from a single or the association of several commits, where a variable amount of files and test cases are involved. Then, iteratively, each batch is fed to the model, so that the weights can be adjusted to minimize the cost function. Once the weights are updated for every batch, we say that an epoch has passed i.e. after training each batch sequentially, the entire dataset is covered. The Data Generator class has several parameters that need to be adjusted: the batch size, comprising the number of commits in a single batch; the shuffle flag, that, if toggled, shuffles the order of the batches once a new epoch begins; and the negative-ratio, which represents the dataset class balance ratio e.g. if set to 2, there will be twice as many randomly picked negative examples as there are positive.

Build Model

Having created the training set, the next step is to build the learning model's architecture. The inputs of the neural network are (file, test) pairs and the output will be a prediction of whether or not there is a link. The *Keras* Deep Learning model is depicted below in Fig. 4.1 and is composed of the following layers:

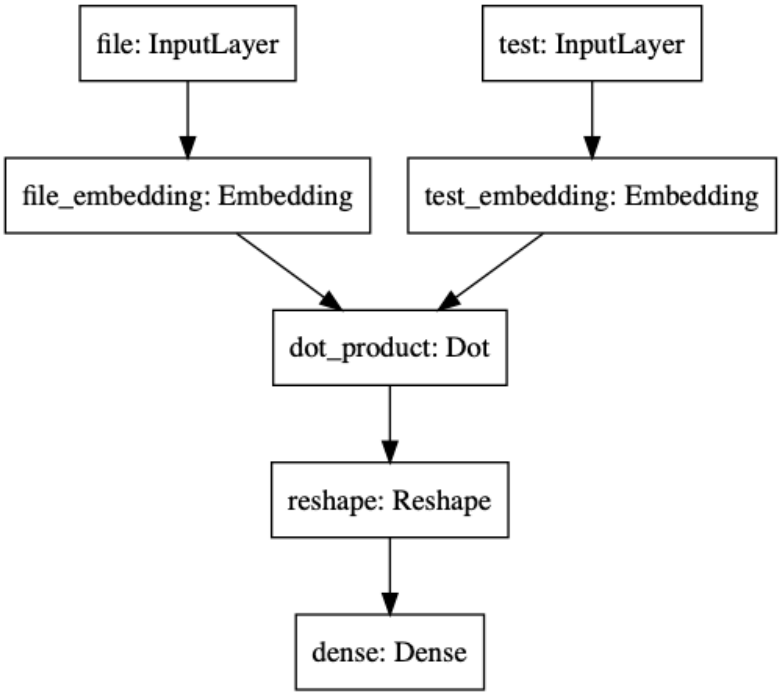

Figure 4.1: Neural Network Embedding Model Architecture

- **Input**: 2 neurons. One for each file and test case in a pair.
- **Embedding**: map each file and test case to a n-dimensional vector.
- **Dot**: calculate the dot product between the two vectors, merging the embedding layers.
- **Reshape**: utility layer to reshape the dot product into a single number. The output of Dot is a 1x1 matrix and needs to be transformed into a single number.
- [Optional] **Dense**: generate output for classification with sigmoid activation function. Commonly known as the logistic function $S(x) = 1/e^{-x} + 1$.

The file and test case in a pair 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, we combine the two representations into a single number by calculating the dot product between the file and test embedding vectors.

For example, suppose file A and test Z are linked, i.e. the pair (A, Z) has label 1. If file A is represented by the embedding vector $[1, 3, 1]$ and test Z by $[-0.5, 0, 1]$, then the dot product between the two is 0.5. This number will be the output of the model, for a regression task, which will then be compared with the true label of the pair (in this example, 1). The weights (i.e. the components of the embedding vectors) will then be adjusted accordingly, in a way as to increase the dot product to match the label observed.

In order to train the model, we need a metric to measure how close the model's output is to the label, called the loss function. For the regression task, we use the mean squared error (MSE), defined as

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \tag{4.1}$$

where n is the number of predicted samples and where y and $\hat{y}$ correspond to the true and predicted labels, respectively. Our goal is to minimize this loss function by readjusting the embedding vectors' values such that the dot product between the file and test embedding vectors, $\hat{y}$, becomes closer to the correct label, y.

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 (BCE), defined by

$$\text{BCE} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)], \tag{4.2}$$

which measures the similarity between binary classes [36].

Additionaly it is possible, to have an idea of the number of parameters the Keras model has to deal with, during gradient descent. Given a system containing 500 files and 3000 test cases (similar to our system), each one of these entities will be mapped to an n-dimensional vector, e.g. $n = 100$. Yielding for the file embeddings $500 * 100 = 50.000$ parameters, plus for test embeddings $3000 * 100 = 300.000$, ending up with a total of 350.000 parameters, where each number corresponds to a vector component of one entity.

Train Model

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

The algorithm used to train the Neural Network Embedding Model is Stochastic Gradient Descent (SGD) with Backpropagation and it is thoroughly described step-by-step in Appendix A.

Evaluation

After training the model, we are able to make predictions on new, unseen commits. We can validate the true accuracy of our model using the test set, which corresponds to 20% of the entire dataset. To evaluate the model's performance, we measure its APTD as well another metric, which varies depending on the supervised task at hand, and which we define next.

In the literature, APFD is the most common metric for measuring TCP performance across different methods [18]. However, for reasons that will become apparent later, our model detects transitions, not just regressions, so the APFD is unsuitable to assess the performance of NNE-TCP. For this reason, we define, for the first time, a metric for Test Case Prioritization based on test case transitions, the Average Percentage of Transition Detection (APTD).

Definition 4.2.1. APTD Let T be the set of tests containing n test cases and τ the set of m transitions revealed by T. Let $T\tau_i$ be the order of the first test case that reveals the i^{th} transition.

$$\text{APTD} = 1 - \frac{T\tau_1 + \cdots + T\tau_n}{nm} + \frac{1}{2n}. \tag{4.3}$$

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

For regression tasks, the metric used is the Mean Absolute Error (MAE) that calculates the average of the difference between the true and predicted labels and it is defined as

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|. \tag{4.4}$$

For classification tasks, we instead use the Binary Accuracy, that counts the frequency of correctly predicted labels, i.e. how often y_i matches $\hat{y}_i$, and it is defined as,

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}, \tag{4.5}$$

where TP and TN mean true positive and true negative and occur whenever $y_i = \hat{y}_i$, either for positive or negative labels, whereas FP and FN occur when $y_i != \hat{y}_i$.

When evaluating the model with the test set, given that it does not know which tests will suffer transitions, the files modified on each commit will have to be paired with every test. For example, if three files were modified and we have $4,000$ test cases, there will be $3 \times 4,000 = 12,000$ pairs in that commit. Then the algorithm will rank all test cases by the likelihood of them being linked to each modified file, a value that is obtained from the dot product between each of the modified files' embedding vectors and each test's. This will result in a matrix of scores $m \times n$ where m is the total number of test cases and n corresponds to the number of files that were modified in the current commit.

Subsequently, to create a test ordering, we pick the maximum value from each row, resulting in a single vector of size m. Test cases are then ranked by descending order of their score. Following the example above, $m = 4,000$ and $n = 3$ and the result will be a n-dimensional vector of prioritized test cases. From this ordering of tests, we can calculate the APTD metric by applying test cases according to their rank in the prioritization generated.

Visualization of Embeddings

Lastly, a useful application of training embeddings is the possibility of representing the embedding vectors in a reduced dimensional space, providing a helpful intuition about entity representation.

Since embeddings are represented in an n-dimensional manifold, one has to resort to manifold reduction techniques to represent elements in 2D or 3D, in order for the manifold to be understandable by humans. This can be done, for example, by using Uniform Manifold Approximation and Projection (UMAP) [37], a technique used to map high-dimensional vectors to lower dimensional spaces, while preserving the structure of the manifold and the relative distances between elements.

4.3 Experimental Setup

An experimental evaluation of our approach to the algorithm is presented in this section, with a brief description of the dataset used, the corresponding cleaning steps applied, the results obtained and comparison with traditional methods.

4.3.1 Data Description

The dataset used to train NNE-TCP consists of historical data collected from a versioning software log [1] 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 commit identifier, 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 used does not contain any information that corresponds to reality.

4.3.2 Data Cleaning

For the data cleaning process, we took three aspects into account:

- **Date Threshold**: timestamp from which we consider file/test modifications/executions - if a file/test has not been modified/executed for n months, it is considered deprecated and is removed.
  Values considered: $n = [6, 12, 18, 24, 30, 36]$ expressed in months.
- **Individual Threshold**: the individual frequency of each element - files and tests that appear fewer than n times are removed, because they are likely to be irrelevant.
  Values considered: $n = [0, 2, 4, 6, 8, 10]$.
- **Pairs Threshold**: frequency of file/test pairs - pairs that occur fewer than n times are likely to have happened by chance, so they are removed.
  Values considered: $n = [0, 1, 2, 5]$

To remove the noise from the data we need to remove files, tests and pairs that rarely are part of a commit in the dataset. A file/test is said to occur in a commit, if it was modified/executed. We want the average number of occurrences per file/test to be larger, leading to a higher density of relevant files and tests, in order to obtain a higher quality dataset. To remove the noise from the data we need 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 Fig. 4.2.

In the Z-axis of the plots, the density of files/tests is represented, the Number of Modifications per File (MpF) and Transitions per Test (TpT) that is obtained by counting the number of times a positive pair contains a certain file/test and dividing by the total number of pairs. For example, by choosing the combination: 10 months, 5 Individual threshold and 0 pairs threshold, the MpF is near 500, which means that on average each file appears on 500 pairs. The TpT is around 100 with the same combination, meaning that on average each test is present on 100 pairs.

In the first plot, we can see that both thresholds strongly influence MpF, that, for higher values, only preserves the files modified most often. However, from the initial 10800 files, only 300 remain. So there is a significant 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.

It may be counter-intuitive that this density decreases as the pairs threshold increases, if we are trying to eliminate irrelevant pairs. However, due to the shape of the data, there are commits where multiple files are modified at once and multiple tests suffer transitions. This event will generate several pair-wise combinations where each file will be paired with each test, increasing the density. However, there could be the case that only one modified file caused all tests to transition, rather than all of them. It is not possible to distinguish which file modification leads to a transition, in this situations. Therefore we will end up with positive pairs that happened by chance and do not correspond to a true file-test link.

This aspect is mitigated by removing less frequent pairs, keeping the ones that also appear on other commits. The decrease in density as the threshold increases is a possible indicator that situations where

[1] Document that displays commit log messages.

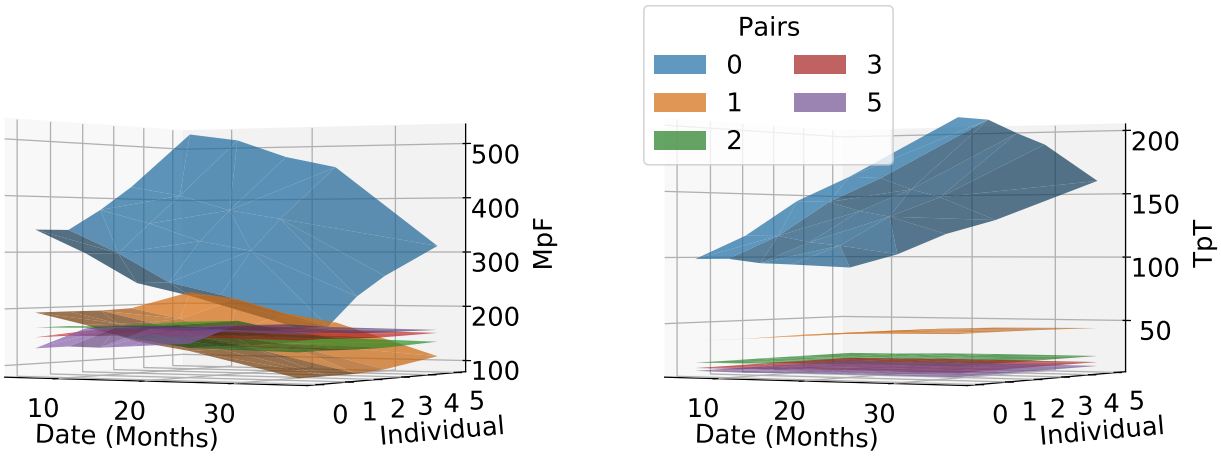

Figure 4.2: Data Cleaning shows Average number of occurrences per files/tests, i.e. Number of Modifications per File (MpF) and Number of Transitions per Test (TpT) as a function of Date, Individual Threshold and Pairs Threshold.

developers commit multiple files at once - considered a bad practice - are frequent and that there is noise present in the data.

Given the description above, in order to have a more expressive dataset without significantly reducing its size, the data cleaning values chosen are:

- **Date Threshold**: 12 months.
- **Individual Threshold**: 5 occurrences.
- **Pairs Threshold**: 1 occurrence.

4.3.3 Traditional Methods

To validate the efficiency of NNE-TCP method, we compare it to three test case prioritization methods:

- **Random** - each test is assigned a random prioritization. This method will serve as a baseline for comparison.
- **Transition** - a fixed prioritization of the test cases is used across every commit. Tests are ranked by their rate of transition, determined by the number of times they have suffered transitions in the past.
- **History** - tests that suffered transitions more recently are assigned a higher rank, i.e. we rank the test cases based on the parameter N_i, which measures the number of commits past since the test case t_i suffered a transition (it is set to 0 whenever the test transitions again). Thus, the smaller the value of N_i, the more recently the test transitioned, and therefore, the higher its rank. This approach is based on two assumptions: regressions are likely to be fixed quickly, i.e. a test that just started failing should transition through a progression soon, and stable test cases, i.e. tests that rarely have any transitions are less relevant. Furthermore, while a project is in state of active development, it is likely that only the same subset of test cases will be involved in transitions, which will be prioritized by *History*. However, this approach does not take into account the fact that a test case that has been failing for some time is more likely to be fixed than a test that had just started failing.

4.4 Results

After framing the problem and having the data cleaned, we can move on to training the model. This training consists of learning the embedding representation, by updating the neural network's weights, through backpropagation. This process is then repeated for 10 epochs, collecting the respective metrics, Accuracy or MAE, depending on the supervised learning task at hand. The algorithm is then evaluated by measuring the APTD for test orderings on the test set.

In this training process, we have some hyper-parameters that influence the model's performance, and these need to be fine-tuned to reach the best results. Some examples of hyper parameters are the embedding size, batch size and negative-ratio. Fine-tuning analysis allows us to find the best combination of parameters that maximize our metrics, by experimenting different values. To determine them, a grid search was conducted, covering different combinations between possible values for each parameter. The choice of optimizer function was not subject to fine-tuning but it is indicated that Stochastic Gradient Descent (SGD) was used.

The parameters that were found to produce the best results are summarized in Table 4.1. Due to limited time and computer power, the values obtained are most likely sub-optimal and more refinement is needed to reach optimality. Additionaly, fine-tuning analysis results are shown in Appendix B in detail. Hereafter, unless stated otherwise, the same set of parameters is used throughout the paper.

Parameter	Best Value	Possible Values
Embedding Size	200	[50, 100, 200]
Negative Ratio	1	[1, 2, 3, 4, 5]
Batch Size	1	[1, 5, 10]
Epochs	10	[10, 100]
Task	Regression	[Classification, Regression]
Optimizer	SGD	SGD
Date (months)	12	[6,12,18,24,30,36]
Individual	5	[0,1,2,3,5,10]
Pairs	1	[0,1,2,5]

Table 4.1: Best combination of parameters obtained after grid-search fine tune analysis and possible values used.

By training the model with the aforementioned combination of parameters, we were able to obtain an APTD $= 0.70 \pm 0.19$. These results are represented in Fig. 4.3, in which we represent a histogram with a Gaussian Kernel distribution - a continuous probability density curve - of the different APTD values obtained for the different commits of the test set.

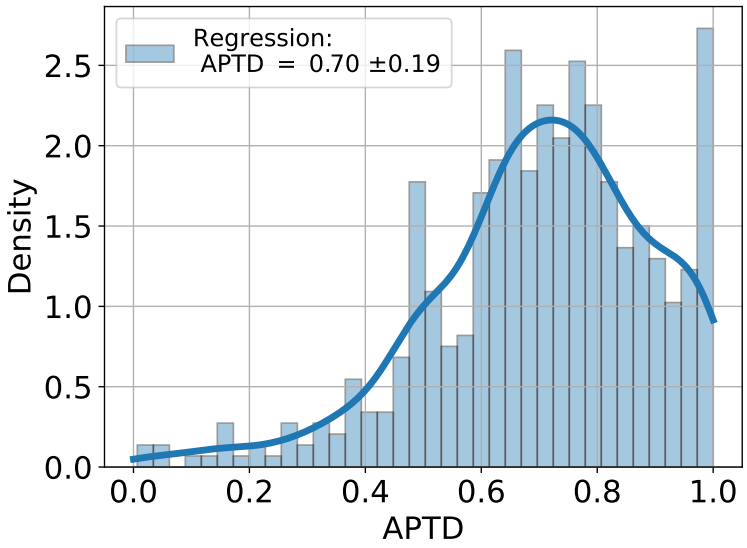

Figure 4.3: APTD histogram and density distribution function (Blue line) for optimal parameter combination

From Fig. 4.3 we can conclude that the model managed to produce a desirable result, with a fairly high APTD. It can also be seen that highest bar in the histogram corresponds to an APTD near 1, which may indicate that, for some commits, the algorithm was able to correctly predict which tests would suffer transitions and prioritized them first.

4.4.1 Cross-Validation

Having fine-tuned the model's parameters to maximize the its performance, the next step is to account for over- and underfitting. The former occurs whenever a model is not able to generalize from the particular set of data, becoming biased. The latter indicates that the model is unable to capture the underlying structure in the data. Our goal is to make sure that, during training, the model can generalize known examples, to predict new information it has never seen before.

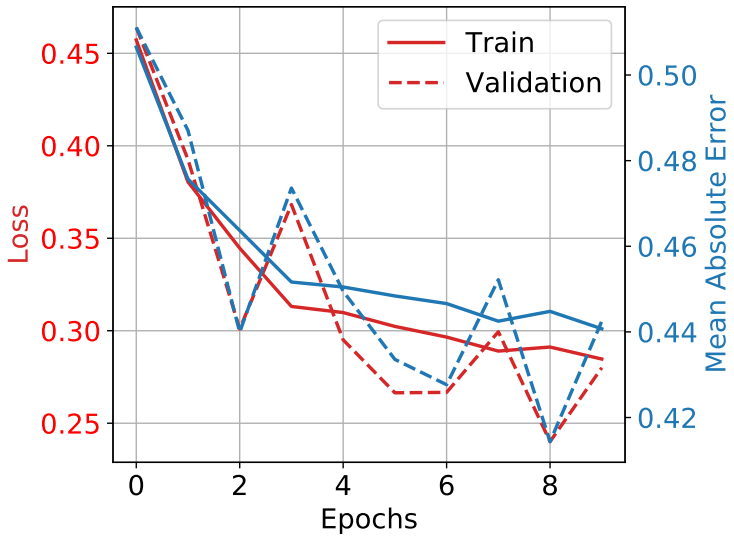

Figure 4.4: Loss (red lines) and MAE (Blue lines) for Cross Validation. Full and Dashed lines correspond, respectively, to the training and validation set. Dashed lines should not deviate from full lines.

Using *Scikit Learn*'s K-Fold cross validation feature [38], the training set can be divided into 10 separated subsets (the gold standard value), called *folds*, used to reduce the bias of a model's fit on the training set. The model is trained ten times, and while nine of those folds are used for training, the one fold is used for validation. Afterwards, we average the ten obtained results for each fold, creating a single model ready to make predictions on unseen data and, hopefully, with enough generalization capability. The one fold is called the *validation-set* and, if the model is not over- or underfitting the data, the loss function - BCE or MSE - and the metric - Accuracy or MAE - of the training set should always be close to that of the validation set.

Fig. 4.4 shows the evolution of the loss function and the metric (in this case, the Mean Absolute Error) for 10 epochs, for the model trained above. If both curves of the validation set (dashed lines in Fig. 4.4) are above the training set's (full lines), then there is underfitting, otherwise there is overfitting. The train and validation curves should ideally stay as close to each other as possible.

It is clear from Fig. 4.4 that there are some discrepancies between the full line (training set) and the dashed line (validation set). However, the variations do not represent a significant difference and it is therefore possible to validate this model and say that there is only slight overfitting, given the amount of noise thought to be present in the data.

4.4.2 Comparison Methods

Having calibrated and trained the NNE-TCP model to produce the best APTD value, we are now able to study how this new approach performs when compared to the three traditional methods described earlier. The results of this comparison are depicted in Fig. 4.5.

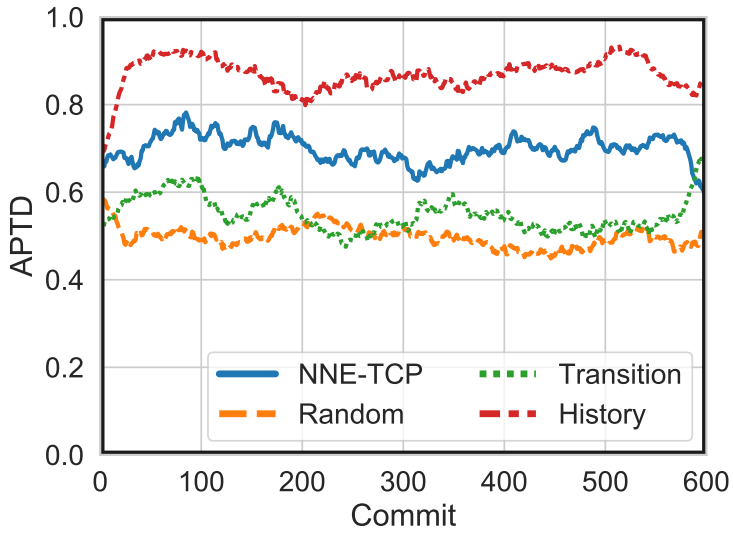

Figure 4.5: APTD Trend of NNE-TCP and traditional methods for the test set. Each line is obtained by calculating the rolling average over a 50 commit window (for a total of 600 commits).

Out of the four methods presented, *History* is the one that shows higher APTD trends, followed by our approach NNE-TCP and, further down, *Transition* and *Random*, respectively. Table 4.2 shows the average value for the APTD and the root-mean-squared error (RMSE) - the average of the quadratic difference between two curves - between NNE-TCP and the other traditional methods.

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

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

The *History* method (red line) was able to achieve the highest trend of the four methods, with the assumption that tests that suffered transitions recently, are more likely to transition again. This assumption, also present in *Transition*, allows for stable tests to be ranked at the bottom of the test schedule, but in a differentiating manner. For example, in *Transition* if two test cases only transition once, but one of them did so very recently, the latter will be more relevant. Hence, stable test are weighted down by the time elapsed since their last execution. In short, the longer a test remains stable, the less relevant it becomes and, consequently the lower priority it has. Additionally, when a regression is detected, it is expected that it will be fixed soon, since the bug-source has clearly been identified, causing a progression. For this reason, a test that transitioned recently through a regression is more likely to transition again through a progression. All these factors help explain the success of the *History* method.

Nevertheless, when regressions occur, the longer a test has been failing, the larger its progression probability becomes, since more time has passed for developers to pin-point and fix the faulty commit. *History* is unable to encapsulate this effect and is also limited in detecting newly introduced regressions, since it does not take into account the available information of each commit (e.g. the modified files), but rather relying only on its heuristic to only prioritize tests that have transitioned recently.

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

Table 4.2: Performance comparison between TCP methods. The RMSE value is calculated in relation to NNE-TCP

As we turn our attention to NNE-TCP (blue line) it is evident that it presents a significant improvement relative to the *Random* and *Transition* prioritization methods, by providing a targeted and commit-focus approach, that directly investigates which files were modified and, from the learned mapping to tests, is able to cherry-pick the tests with the most affinity.

However, it should be said that *History* shows better prioritization capability than NNE-TCP, by only taking into account a test case's history of transitions. Notwithstanding, we have reasons to believe that NNE-TCP has not reached its full potential. Ideally, with a more abundant amount of quality data, a crystal clear relation between modified files and test cases could be found, with minimized uncertainty. With this information, when a commit is made, the algorithm will be fed with the files modified and know exactly which tests are more likely to affected, eliminating unnecessary executions. This way, the problem of not being able to detect newly introduced regressions would be settled, enabling quick transition detections, leading to early fixes.

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

In conclusion, the NNE-TCP approach performs better than *Transition* and *Random*, and worse than *History*. Nonetheless, we strongly believe that there are enough evidences to consider NNE-TCP as viable alternative to address the Test Case Prioritization problem and to be implemented in an industrial environment.

4.4.3 Entity Representation

The advantage of using Embedding Models is that beyond solving a Supervised Learning Problem, the results can be visualized in 2D space by using either TSNE [39] or UMAP [37], shown below and respectively.

These dimensionality reduction algorithms are able to reduce a 200-dim vector down to 3 or 2 and we will use both algorithms for comparison. Usually, TSNE has higher complexity and takes longer to run and it was devised to conservate the local structure within the data. While UMAP is much faster and tries to maintain a balance between local and global structure of the manifold in higher dimensions.

In our case, there are approximately 3,000 test cases, meaning that with One-Hot Encoding we would have 3,000 dimensions. Embeddings allows us to reduce the number of dimensions to 200 and then with Manifold reduction techniques like T-SNE and UMAP we are able represent this entities in 2 dimensions. To facilitate meaningful visualizations, a compromise of the exact distance between embeddings is made, by these manifold reduction techniques that try to still capture those that are closer together.

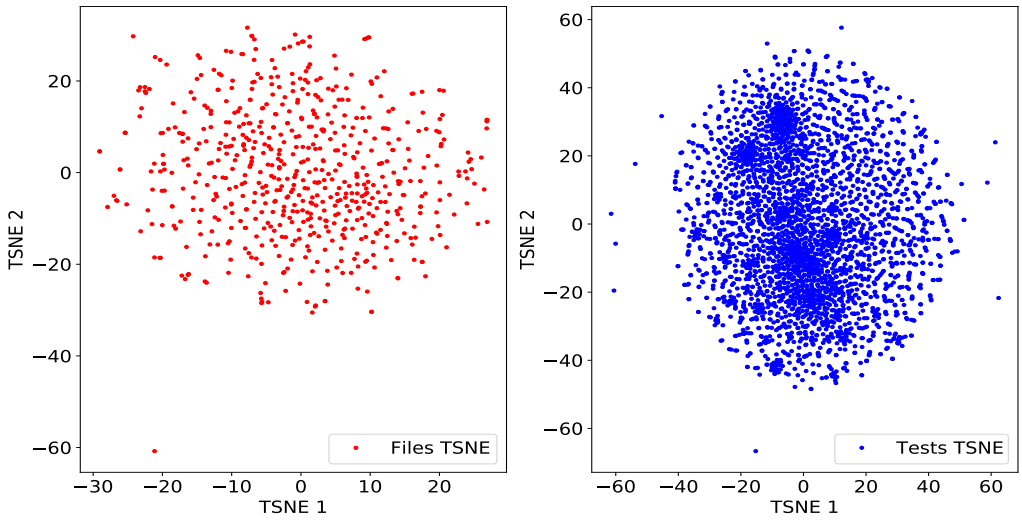

Figure 4.6: TSNE File and Test Case Representation in 2D space

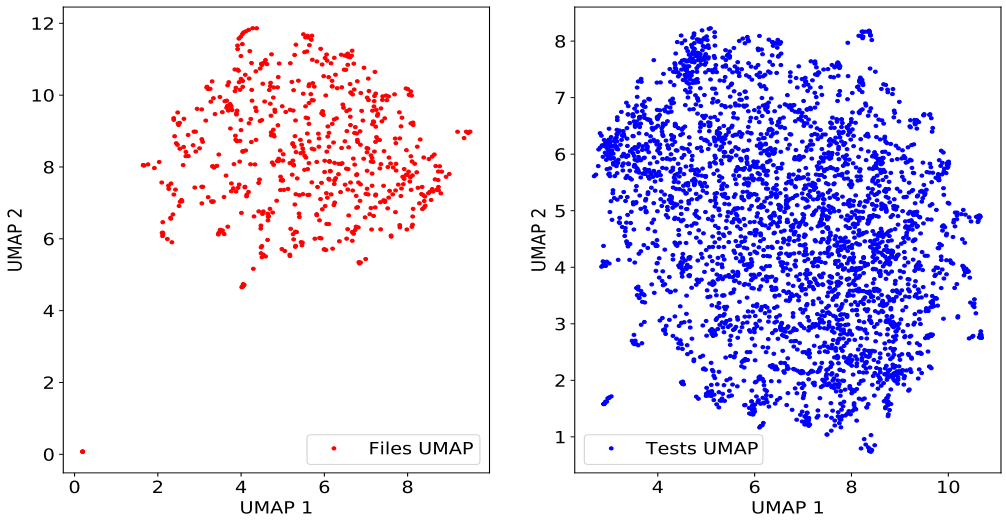

Figure 4.7: UMAP File and Test Case Representation in 2D space

On each axis, one component of either TSNE or UMAP is represented. It is important to point out that the embedding representation of test cases is much more meaningful than the file representation. It is much more relevant to know which test cases will suffer transitions concurrently, such that redundancy is avoided. Nonetheless, information about files that are modified together and trigger the same test cases, is also relevant to find system dependecies, that are hard to find manually.

For modified files, in both graphs the vectors seem to be evenly distributed. For test cases, using T-SNE, there are observable denser areas, pointing to more similarity within those groups. Using UMAP, the same reasoning is not evident, although there is not a homogeneous distribution. The lack of evident more dissociated groups may be an indicator of a small training set, that does not provide enough information to space out each vector and clearly see which test cases trigger at the same time.

Below, Fig. 4.8 and Fig. 4.9 show the embedding representation of labelled files and test cases according to the corresponding folder where they are stored in the system, using TSNE and UMAP, respectively.

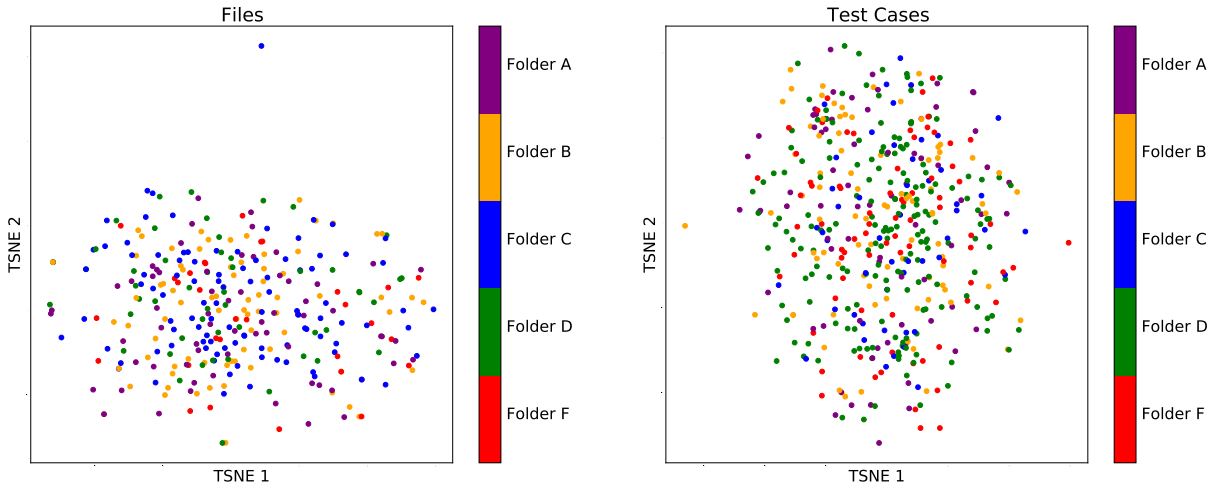

Figure 4.8: Labelled Embeddings with TSNE technique. Labels correspond to the five more populated directories where files/test-cases are stored in the system.

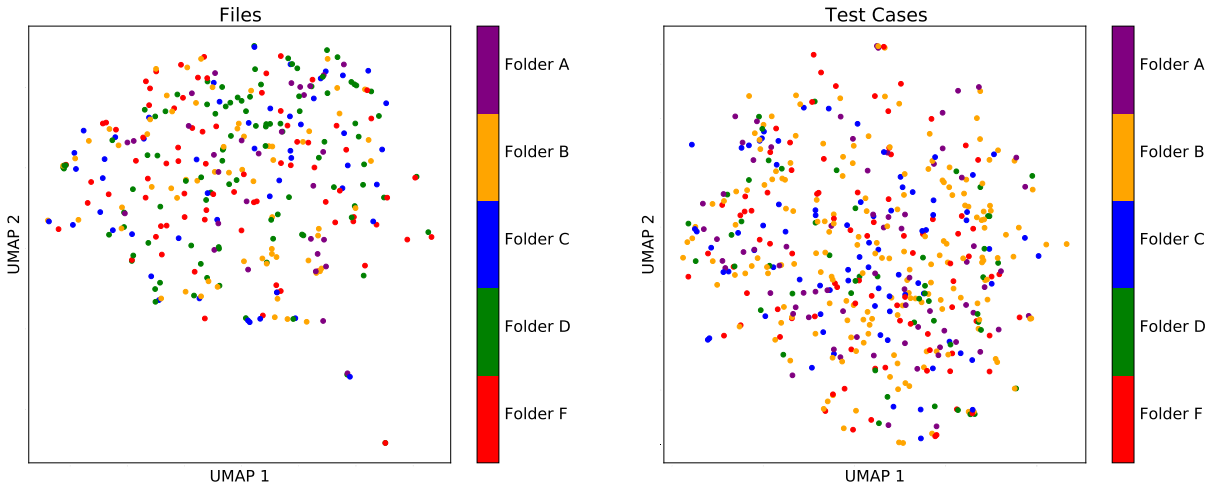

Figure 4.9: Labelled Embeddings with UMAP technique. Labels correspond to the five more populated directories where files/test-cases are stored in the system.

On both graphs, each dot has a corresponding label, helping us see if the current organization in folders of files and test cases is correlated with the behaviour of transitions. In Fig. 4.9, only five folders - the ones with most frequent files and tests - were considered from the set of all folders, to facilitate visualization. Because we can not observe any single color clusters in the projections, it is possible to conclude from the plot that there is no correlation between the folder where files/tests are stored and whether they cause/suffer transitions together. It should be noted that there were cases where files and tests' folder names were corrupted or unavailable.

Furthermore, there is the possibility to apply a Clustering algorithm to the obtained embeddings, where each cluster contains elements of similar behaviour, unlocking an array of possible improvements for TCP. By selecting one test from each cluster, we could possibly achieve maximum code coverage, i.e. covering most amount of source code with a particular subset of test cases, in the shortest amount of time or consuming the least amount of resources.

Finally, one very important aspect of entity representation is knowing which files cause transitions together. In software development, oftentimes there is a massive interdependence between files, e.g. external libraries, large projects, etc., and, sometimes, these are the source of many introduced regressions [5]. These interdependencies are paper-thin and burdensome to find manually. From the embedding results, if two or more file dots overlap in Fig. 4.9, then there is a strong indicator that the files have a relation between them.

4.4.4 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 trains the model multiple times. Furthermore, the dataset is relatively small, facing the number of test cases and modified files it encompasses. Collecting more data is a crucial step for Machine Learning models to learn better and more complex relations between inputs and outputs.

Another threat can derive from errors associated with our code implementation. Both *Scikit-Learn* [38] and *Keras* [36] are well established frameworks used for Machine Learning. However, our implementation might contain inconsistencies and errors, which could lead to erroneous results. The code used in this work is available online at *https://github.com/jlousada315/NNE-TCP* for validation purposes.

External

This work 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.

As stated previously, the presence of noise in the dataset is a major setback that interferes with the algorithm's learning process. Human factors are one such example, and these can be mitigated by changing workplace practices, such as promoting a higher commit frequency and modifying fewer files on each commit rather than accumulating changes for long periods of time and then committing all at once.

Construct

In real-world complex CI systems, sometimes test cases change status due to extraneous reasons: system dependencies on several platforms can affect the outcome of a test, the infrastructure where tests are executed can suffer critical faults or malfunctions, some tests can fail to produce the same result each time they are executed (i.e. flaky tests). Therefore it is not certain that whenever a test case changed status it was due to a certain file being 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 model's ability to make better predictions such as the commit author, execution history of each test, test age, etc.

Chapter 5

Conclusions

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

Results indicate that RETECS can effectively create meaningful test schedules in different contexts. In the new *Finance* dataset, with a combination of the Test Case Failure reward with the Network Approximator, around 90 commits suffice to reach the performance of deterministic methods and surpass random prioritization of test-cases. Initially, the evaluation metric NAPFD starts at a value of only 0.2 but, as the algorithm progresses, the trend shows values over 0.6.

The inclusion of DT's in the framework failed to produce better results relative to the Network, in the best possible case. However in some cases, with other reward functions, performance is comparable and might not be discarded right away, as it can be useful to apply, in future research, to other CI environments with distinct characteristics.

In this work we presented an original approach to Test Case Prioritization using Machine Learning, called NNE-TCP. It combines Neural Network Embeddings with file-test links obtained from historical data, which, to the best of our knowledge, was done here for the first time. It 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, which is a valuable feature not usually available in other frameworks.

Our results point to a major improvement over some traditional TCP methods, which we called *Random* and *Transition*. However, NNE-TCP was unable to match the performance of a third, more complex traditional method, called *History*. Nevertheless, we strongly believe that NNE-TCP has enough potential to reach higher performance levels. If a mapping between files and tests can be effectively learned by a data-driven approach, then only relevant tests will be executed, reducing feedback time. To validate this hypothesis, further experiments must be conducted on richer and cleaner datasets.

Finally, the ability to visualise embeddings in 2D space represents a valuable improvement over other commonly used methods, providing insights on the structure of the data. We showed that there does not seem to be exist any correlation between the folders where files/tests are stored and the similarity between the files/tests themselves. Notwithstanding, it is already possible to detect possibly redundant tests and discover dependencies between files.

5.1 Achievements

From inception, the nature of this thesis was exploratory. The main achievements can be summarized:

- There was no Test Case Prioritization strategy at BNP Paribas, so the baseline goal of increasing the Average Percentage of Transition Detection considerably, was achieved, bridging the gap between academia and industry.
- Two Machine Learning Models were implemented and ready to be deployed in an industrial environment with immediate impact on developers productivity.
- Two contributions to the scientific community, in the form of paper publication and open-source code available online.
- Make way for others to develop further studies, by presenting future research paths.

5.2 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 higher complexity in relation to traditional methods, unless its performance surpasses these other methods, it's not actually worth implementing. For its performance to increase, it requires more information to formulate better reasoning of expected failures, e.g. links between test cases and modified files.

Regarding Machine Learning models as function approximators, DT's showed a slighty worse performance when compared to Network, however without collecting more records and a more refined parameter tuning analysis to try to find optimal values for all parameters, it is not possible to discard it. Additionally other models, rather than DT's, can and should be considered to represent memory, such as Nearest Neighbours.

Furthermore, in real world environments, test-cases are usually run on a grid that allows for parallelization. In our framework, we assumed that test cases were 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 parallelization, it would be more fruitful to create groups of non-redundant tests to maximize the state-space 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, establishing a solid baseline for further studies.

Due to limited time and computer power, parameter tuning analysis was limited, but it can be further refined by exploring more combinations of parameters and measuring their impact on the APTD metric. Furthermore, our approach should take into consideration additional features, such as the commit author, to allow better understanding of expected transitions, shortening even more the delay between a software change and the information regarding whether it impacts the system's stability.

In terms of entity representation, embedding projection has the potential to provide a valuable insight about the system's structure, giving the chance to reorganize tests in different folders, grouped by similarity. Moreover, a clustering algorithm can be applied to group test cases. Test cases in the same cluster usually fail in similar situations, and therefore applying a subset of each cluster, rather than every test case, would avoid redundancies, without compromising code coverage, quickly grasping the status of the whole system.

Finally, NNE-TCP should be validated and tested against other datasets, so that it can become more flexible and adaptable to different contexts.

Furthermore, one promising future research direction would be to combine both methods into one large prioritization system. There is the possibility of combining historical features - that are good at prioritizing test that were promising in the past - with coverage features - that are good at guessing what tests will be relevant in the nearby future - such as the list of modified files.

Bibliography

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

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] P. Duvall, S. M. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007. ISBN 0321336380.

[13] 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.

[14] BNP Paribas general description. https://www.bnpparibas.pt/en/bnp-paribas/bnp-paribas-group/. Accessed: 2020-12-03.

[15] 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.

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

[17] 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.

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

[19] M. Ivankovic, G. Petrovic, R. Just, and G. Fraser. Code coverage at google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 955–963, 2019.

[20] M. Khatibsyarbini, M. Isa, D. Jawawi, and R. Tumeng. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, 93, 09 2017. doi: 10.1016/j.infsof.2017.08.014.

[21] D. Marijan, A. Gotlieb, and S. Sen. Test case prioritization for continuous regression testing: An industrial case study. pages 540–543, 09 2013. doi: 10.1109/ICSM.2013.91.

[22] 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.

[23] T. Noor and H. Hemmati. Studying test case failure prediction for test case prioritization. pages 2–11, 11 2017. ISBN 978-1-4503-5305-2. doi: 10.1145/3127005.3127006.

[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] T. Zhou, X. Sun, X. Xia, B. Li, and X. Chen. Improving defect prediction with deep forest. *Information and Software Technology*, 114:204 – 216, 2019. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2019.07.003. URL http://www.sciencedirect.com/science/article/pii/S0950584919301466.

[26] 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.

[27] 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.

[28] 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.

[29] H. Hemmati and F. Sharifi. Investigating nlp-based approaches for predicting manual test case failure. *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 309–319, 2018.

[30] 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.

[31] C. Leong, A. Singh, M. Papadakis, Y. L. Traon, and J. Micco. Assessing transition-based test selection algorithms at google. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '19, page 101–110. IEEE Press, 2019. doi: 10.1109/ICSE-SEIP.2019.00019. URL https://doi.org/10.1109/ICSE-SEIP.2019.00019.

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

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

[34] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[35] K. W. Church. Word2vec. *Natural Language Engineering*, 23(1):155–162, 2017. doi: 10.1017/S1351324916000334.

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

[37] 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.

[38] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

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

[40] 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.

[41] 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.

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

[43] 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.

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

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. [40]

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[40]:

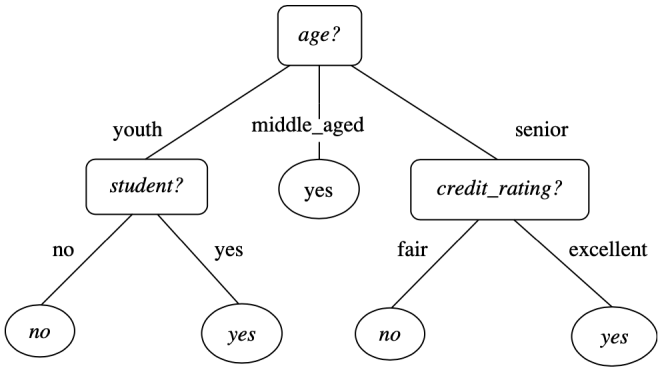

Figure A.1: Each internal (non-leaf) node represents a test on an attribute. Each leaf node represents a class (if the client is likely to buy the product yes/no)

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. $N_m < min_{samples}$ [41]

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. [40]

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 [40].

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.

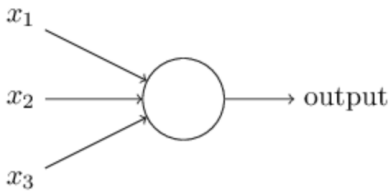

Figure A.2: Perceptron simple example)

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 [42]. 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:

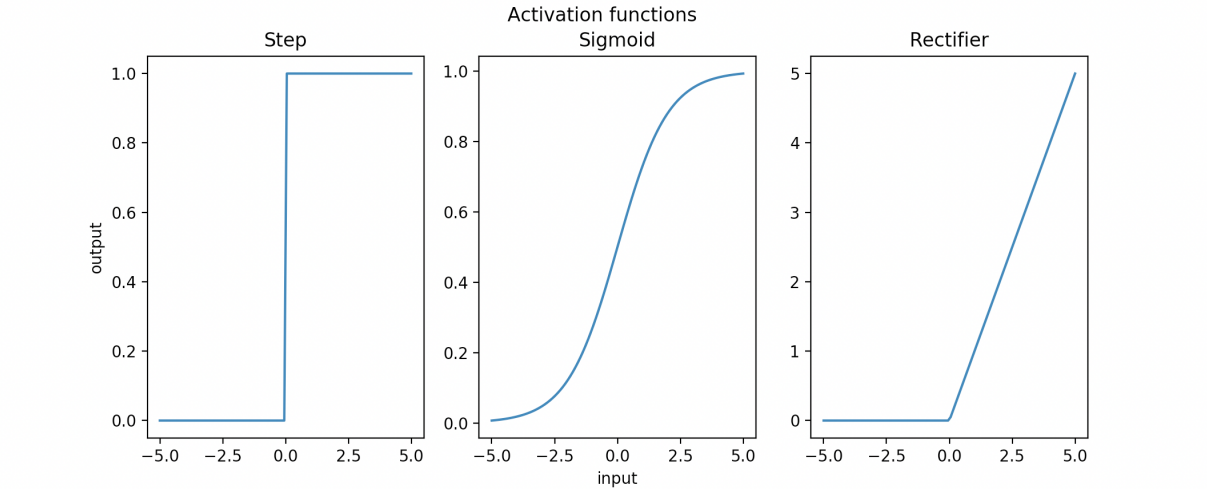

Figure A.3: Comparative analysis of different activation functions (made in Python))

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. [42].

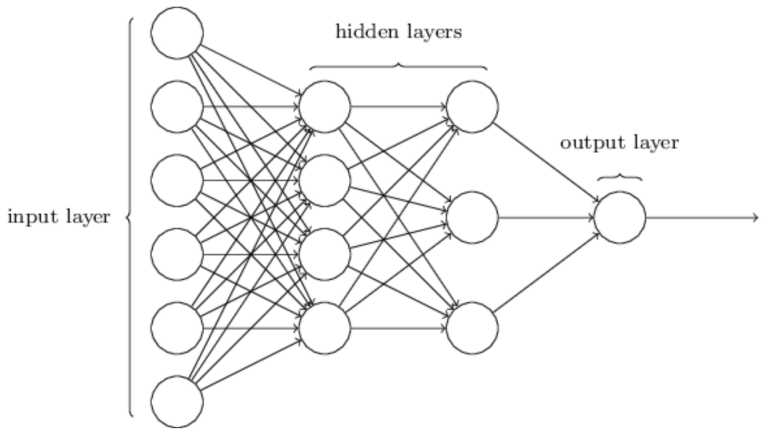

Figure A.4: Architecture example of a NN [42]

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 determine 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.[42]

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[43].

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).[43]. Minimising a cost function looks this:

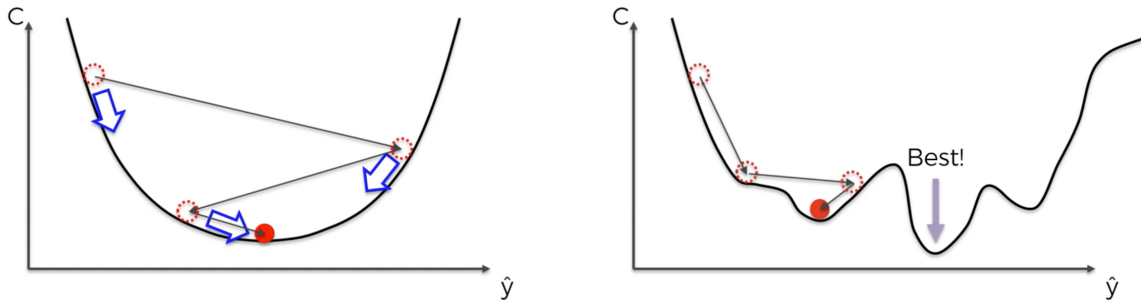

Figure A.5: Global and Local minimum determination [44]

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. [43]

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. [42] 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*. [43]

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. [42]

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_j^L to the bias b_j^l to obtain the second backpropagation relation:

$$\Delta_j^L = \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial b_j^l}\frac{\partial b_j^l}{\partial z_j^l} = \frac{\partial C}{\partial b_j^l} \tag{A.16}$$

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

$$\Delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_k^l} \tag{A.17}$$

$$= \sum_k \Delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_k^l} \tag{A.18}$$

$$= \left(\sum_k \Delta_k^{l+1}\omega_{k,j}^{l+1}\right)\sigma'(z_j^l) \tag{A.19}$$

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

$$\frac{\partial C}{\partial \omega_{j,k}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial \omega_{j,k}^l} = \Delta_j^l a_k^{l-1} \tag{A.20}$$

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 [43]. It described in the following steps:

Backpropagation Algorithm:

1. **Activation at input layer** - compute the activations of the input layer a_j^1.
2. **Forward propagation** - Through equation A.13, determine the activations a_j^1 and the linear weigthed sum z_j^l, for each layer l.
3. **Error at layer** L - calculate the error Δ_j^L, using equation A.16.
4. **Propagate the error "backwards"** - calculate the error Δ_j^l, 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 $\frac{\partial C}{\partial \omega_{j,k}^l}$.

[43]

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. [42]

Appendix B

Fine-Tuning

B.1 Plots

In this section, the grid search results are depicted for every combination of parameters, summarized in the table B.5.

Parameter	Possible Values
Embedding Size	[50, 100, 200]
Negative Ratio	[1, 2, 3, 4]
Batch Size	[1, 5, 10]
Nb_Epochs	[10, 100]
Task	[Classification, Regression]
Optimizer	[SGD]

Table B.1: Grid-Search Values for Fine-Tuning Analysis

Then, each combination of parameters is used train a new model and the APTD metric is measured:

- The three plots shown below are organized by Epochs and Batch size.
  - Each subplot has the negative ratio parameter, corresponding to each row
  - Each subplot has the embedding size parameter, corresponding to each column
- Two curves: the blue line for Classification and orange for Regression.
- The plot legend gives the mean value and standard deviation for each distribution.
- The x-axis represents the values of the APTD metric, ranging from 0 to 1.
- The y-axis represent the density of each bin.
- Grid-Search for 100 Epochs is only calculated at the end, with best obtained combination.

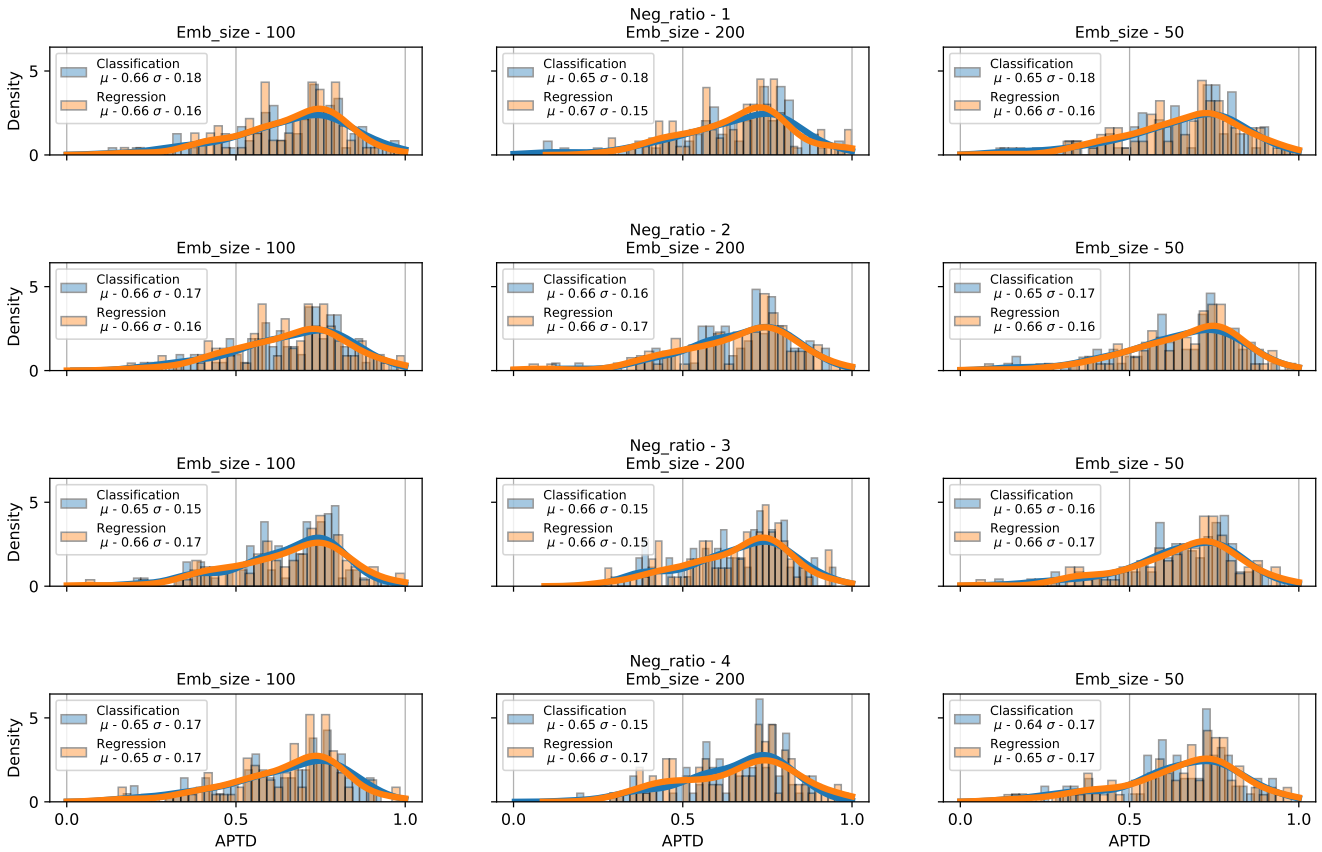

Figure B.1: APTD Parameter Tuning for 10 epochs and Batch Size 1.

Best found distribution, with APTD $= 0.67 \pm 0.15$, is:

Parameter	Value
Embedding Size	200
Negative Ratio	1
Batch Size	1
Epochs	10
Task	Regression
Optimizer	SGD

Table B.2: Grid-Search Values for Fine-Tuning Analysis

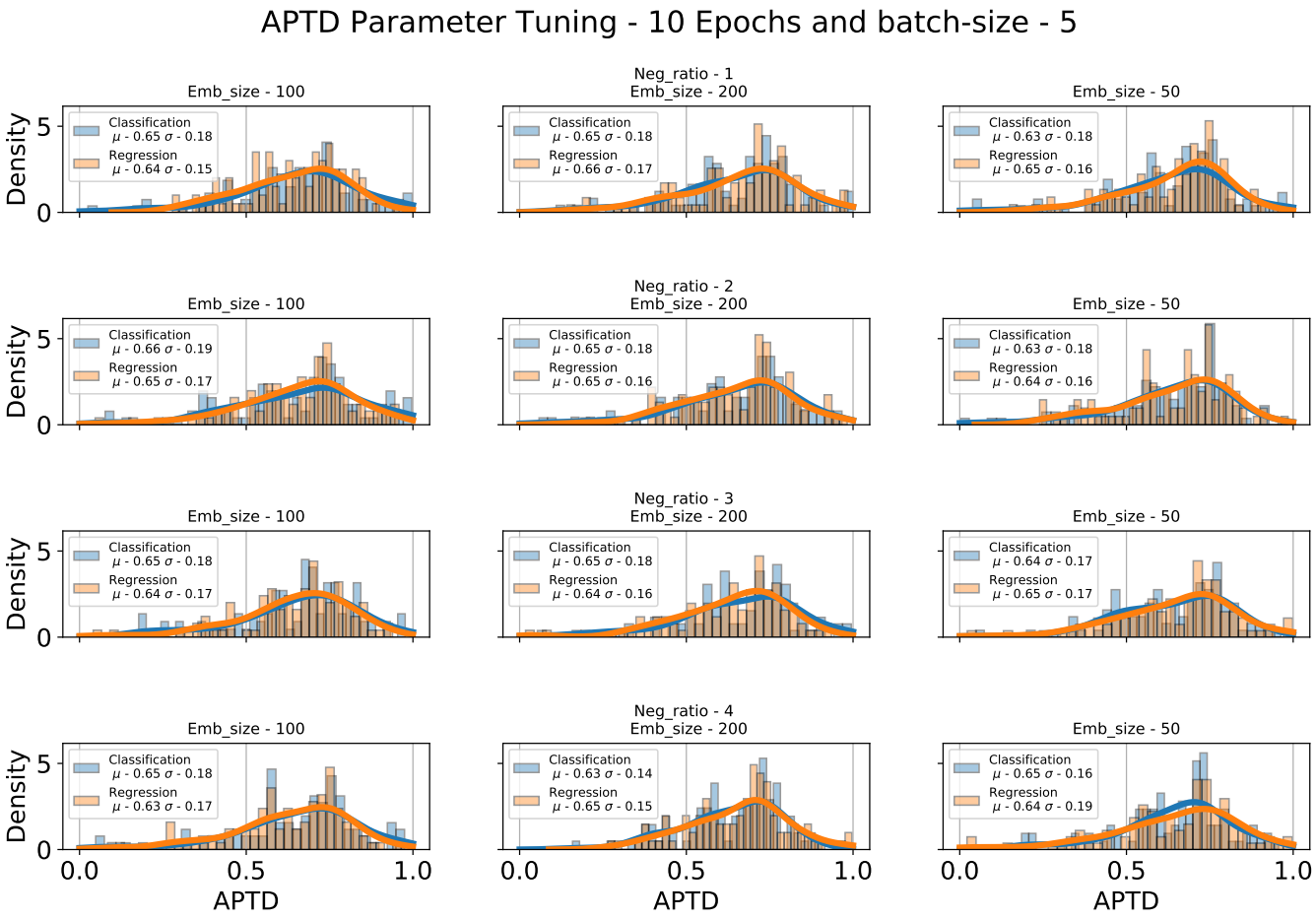

Figure B.2: APTD Parameter Tuning for 10 epochs and Batch Size 5.

Best found distribution, with APTD $= 0.66 \pm 0.17$, is:

Parameter	Value
Embedding Size	200
Negative Ratio	1
Batch Size	5
Epochs	10
Task	Regression
Optimizer	SGD

Table B.3: Grid-Search Values for Fine-Tuning Analysis

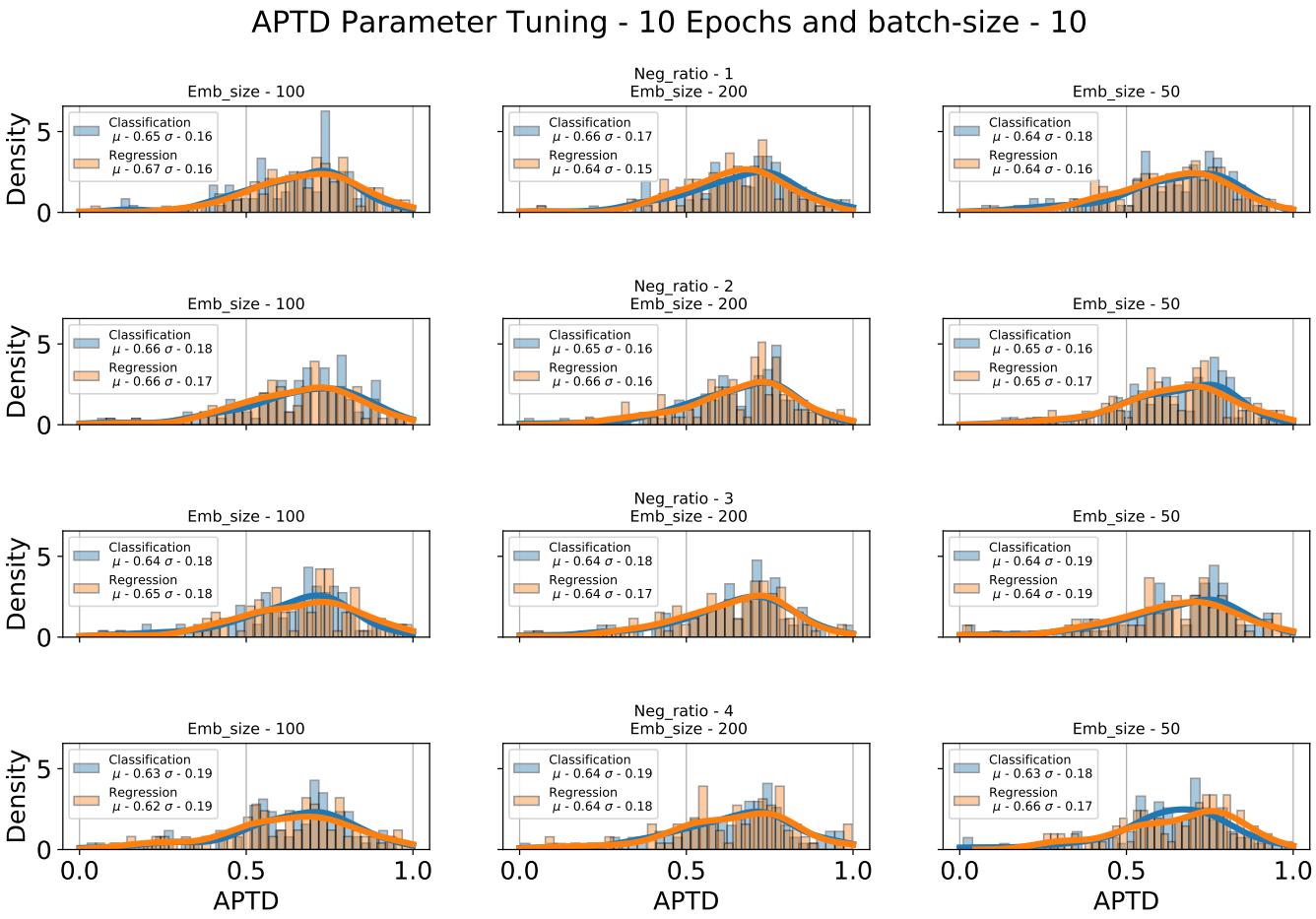

Figure B.3: APTD Parameter Tuning for 10 epochs and Batch Size 10.

Best found distribution, with APTD $= 0.67 \pm 0.16$, is:

Parameter	Value
Embedding Size	100
Negative Ratio	1
Batch Size	10
Epochs	10
Task	Regression
Optimizer	SGD

Table B.4: Grid-Search Values for Fine-Tuning Analysis

After choosing the best distribution overall, the best combination of parameters is used to create a new model that was trained for 100 epochs.

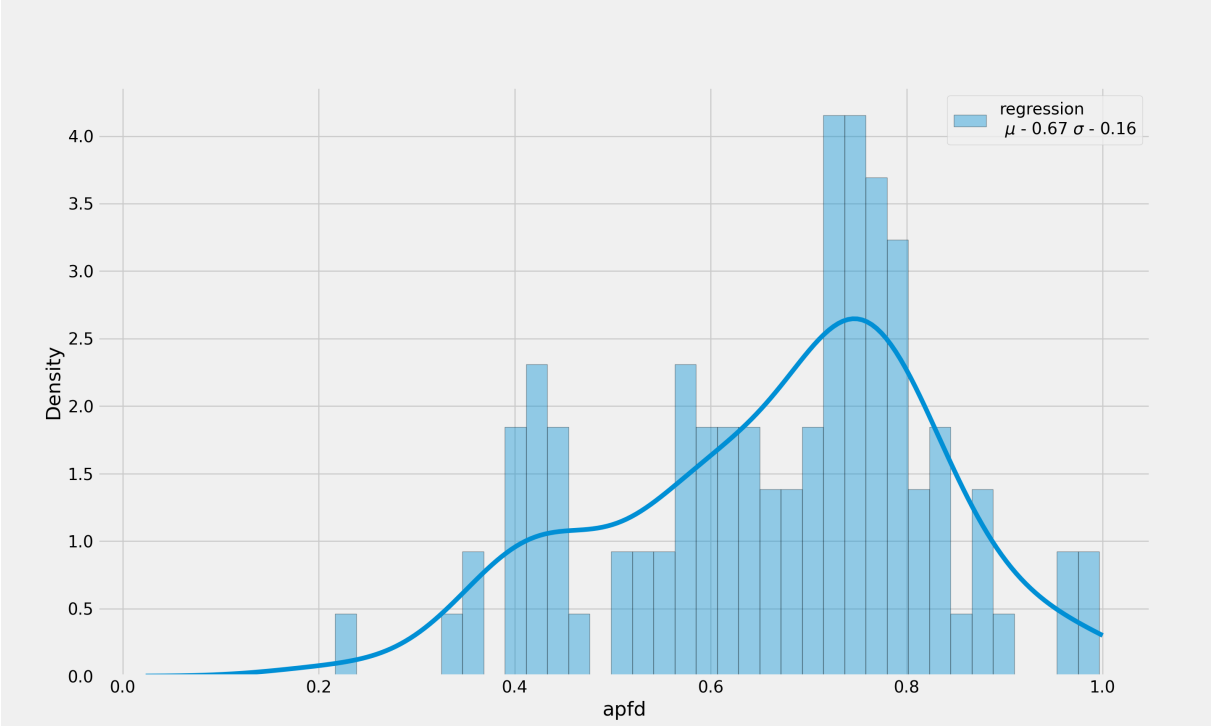

Figure B.4: APTD Parameter Tuning for 100 epochs.

As we can see, the model does not increase performance as the number of Epochs is increased considerably. Indicating convergence at 10 epochs.

Therefore we can conclude that the optimal found parameters, for an APTD $= 0.67 \pm 0.15$, is:

Parameter	Value
Embedding Size	200
Negative Ratio	1
Batch Size	1
Epochs	10
Task	Regression
Optimizer	SGD

Table B.5: Grid-Search Values for Fine-Tuning Analysis