TÉCNICO LISBOA

Keeping Master Green

with Machine Learning

João Luís Xavier Barreira Lousada

Thesis to obtain the Master of Science Degree in

Engineering Physics

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

Examination Committee

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

October 2020

Dedicated to someone special...

Acknowledgments

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

Resumo

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

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

Abstract

Insert your abstract here with a maximum of 250 words, followed by 4 to 6 keywords...

Keywords: keyword1, keyword2,...

Contents

Acknowledgments v
Resumo vii
Abstract ix
List of Tables xiii
List of Figures xv
Nomenclature 1
Glossary 1

1 Introduction **1**
1.1 Motivation 1
1.2 Continuous Integration 2
1.3 Repository 2
1.4 Objectives 2
1.5 Thesis Outline 2

2 Background **3**
2.1 Version Control Systems 3
2.1.1 Repository & Working Copy 3
2.2 Mono-repositories vs. Multi-repositories 3
2.3 Software Testing 4
2.4 Regression Testing 5
2.4.1 Test suite minimisation 5
2.4.2 Test Case Selection 6
2.4.3 Test case prioritisation 6
2.5 Approach Exploration 7
2.5.1 Naive-approach 7
2.5.2 Cumulative Approach 7
2.5.3 Change-List Approach 8
2.5.4 Always Green Master Approach 9
2.6 Machine Learning 10
2.7 Supervised Learning 11

2.7.1 Logistic Regression . . . 11
2.7.2 Decision Trees . . . 13
2.7.3 Ensemble Methods . . . 15
2.7.4 Artificial Neural Networks . . . 15
2.7.5 Model Evaluation and Parameter Tuning . . . 22
2.7.6 AutoEncoders . . . 22

3 Implementation **23**
3.1 Synthetic Data Generation . . . 23
3.2 Numerical Model . . . 24
3.3 Verification and Validation . . . 24

4 Results **25**
4.1 Problem Description . . . 25
4.2 Baseline Solution . . . 25
4.3 Enhanced Solution . . . 25
4.3.1 Figures . . . 25
4.3.2 Equations . . . 26
4.3.3 Tables . . . 27
4.3.4 Mixing . . . 28

5 Conclusions **29**
5.1 Achievements . . . 29
5.2 Future Work . . . 29

Bibliography **31**

A Vector calculus **35**
A.1 Vector identities . . . 35

B Technical Datasheets **37**
B.1 Some Datasheet . . . 37

List of Tables

2.1 Comparative analysis between strategies 10

4.1 Table caption shown in TOC. 27
4.2 Memory usage comparison (in MB). 27
4.3 Another table caption. 27
4.4 Yet another table caption. 28
4.5 Very wide table. 28

List of Figures

2.1 Average percentage of fault detection [10] 7
2.2 Code repository and workflow at Google. Each CL is submitted to master branch or HEAD [2] 8
2.3 Speculation tree - builds and outcomes [1] 10
2.4 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) 13
2.5 Perceptron simple example) 16
2.6 Comparative analysis of different activation functions (made in Python)) 17
2.7 Architecture example of a NN [18] 18
2.8 Global and Local minimum determination [19] 19

4.1 Schematic of some algorithm. 26

Chapter 1

Introduction

Nowadays, it has become crucial to have an efficient and reliable way to keep track of software changes, specially in large and fast-paced companies [1]. In many of them, source-code *"repositories"* are the preferred choice: offering the agility of having access to all historical code versions developed so far, keeping track of the changes that were made. Due to the sheer amount of changes that are made daily, in these kinds of environments, the probability that one of them creates a conflict is significant, leading to a breakage. This is an undesirable situation, since it may lead to development over a defective *"master"* (mainline hereafter) [1], so it is critical to rapidly detect and patch what caused the breakage.

A mainline is *"green"*, if all build steps (i.e. compilation, unit tests, User Interface tests) are successfully executed for every change point in history, otherwise it is called *"red"*, and keeping it that way is a key factor to achieve maximum performance [1]. However finding a scalable solution, given a constraint of time and computer resources, is a challenging task in order to find a balance between correctness and speed, so there is a high demand for automated techniques to help developers keep the master green.[2]. The strategy is to collect data from a large company and use heuristics to exploit common patterns that can be learned by machine learning algorithms and then, based on that knowledge, estimate where a breakage, most likely, occurred, and most importantly reduce the time to do so.

There are many ways to configure the architecture of such systems and to manage testing. Therefore, several approaches are compared below. For now, let us start by defining some fundamental concepts needed for the analysis.

1.1 Motivation

Relevance of the subject...

[1] a master represents the main history of code versions - i.e. the main branch (described below)

1.2 Continuous Integration

Continuous Integration (CI) is a popular software development technique that allows developers to easily check that their code can build successfully and pass tests across various system environments.[3] However, it is not straightforward to develop and configure this methodology, there are numerous approaches and several ways to solve the same problem, which are explored below. The key aspect is to navigate through the options of how to commit and how to test, compare them and acknowledge what features can be leveraged at the expense of others, making it possible to find a scalable solution.

1.3 Repository

Provide an overview of the topic to be studied...

1.4 Objectives

The objectives delineated for this work are:

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

Explicitly state the objectives set to be achieved with this thesis...

1.5 Thesis Outline

Briefly explain the contents of the different chapters...

Chapter 2

Background

2.1 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.[3]

2.1.1 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*).[3] When a client *writes* information, it becomes accessible to others and when one *reads*, obtains information from the repository. The major difference from an usual server is the capability of remembering every version of the files. With this formulation, a client has the possibility to request any file from any previous version of the system.

While developing, having multiple versions of a project is not very useful, given that most compilers only know how to interpret one code version with a specific file type. So the bridge linking the repository and the user is the **working copy**. - a local copy of a particular version, containing files or directories, on which the user is free to work on and later on communicate the changes to the repository.[3]

2.2 Mono-repositories vs. Multi-repositories

In order to manage and store amount of new code produced daily, there are two possible ways of organizing repositories: one single giant repository that encompasses every project or assigning one repository for each one. Google's strategy is to store billions of lines of code in one single giant repository, rather than having multiple repositories. [4] A mono-repository is defined such that it encompasses the following properties:

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

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

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

Although electing a preferred candidate is still a matter of debate, mono-repositories are linked to less flexibility and autonomy when compared to multi-repositories, but there is the gain of consistency, quality and also the culture of the company is reinforced, by unifying the tools used to link the source code to the engineer.[5]

2.3 Software Testing

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

Lets take a step back and define what testing is. Testing is a verification method to assess the quality of a given software. Although, this sentence is quite valid, it is vague, in the sense that it does not define what quality software actually is. In some contexts, quality might just refer to simply code compiling with no syntax errors or ”mistakes”. When tests are applied, the outcome obtained is of the form PASS/FAIL, with the purpose of verifying functionality or detecting errors, receiving quick and easy to interpret feedback. However, the connections between testing and quality are thin: testing is very

much like sticking pins into a doll, to cover its whole surface a lot of tests are needed. For example, running a battery of tests (test suite) where every single one of them yields PASS. This can only mean two things: whether the code is immaculate or some scenarios were left out of the process. Usually, test suites are constructed from failed tests. When in a FAIL situation, i.e. fault detection or removal , a new test case is created, preventing this type of error to slip again in the future, incrementing the existing test suite, in a ”never ending” process). So 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. [6]

2.4 Regression Testing

Regression testing is performed between two different versions of software in order to provide confidence that the newly introduced features of the working copy do not conflict with the existing features. In a nutshell, whenever new features are added to an existing software system, not only the new features should be tested, but also the existing ones should be tested to ensure that their behaviours were not affected by the modifications. Usually this is done by applying test cases, to check if those features, in fact work or are still working. Therefore, this field encapsulates the task of managing a pool of tests, that are repeatedly applied across multiple platforms. [7]

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

- **Test Case Selection** - ”do smarter” approach, selecting only relevant tests, given the type of change.
- **Test Suite Minimisation** - ”do fewer” approach, removing possible redundancies.
- **Test Case Prioritisation** - also ”do smarter” approach by running some tests first, increasing probability of early detection. [7]

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.

2.4.1 Test suite minimisation

Definition 2.4.1. 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). [8]

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. [7]

2.4.2 Test Case Selection

Definition 2.4.2. 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 2.4.3. 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 $P(t) \neq P'(t)$. [9]

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. [7]

2.4.3 Test case prioritisation

Definition 2.4.4. 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 [7]. 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) [10].

Definition 2.4.5. 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.[7]

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

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

[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[8]

f

test	fault									
	1	2	3	4	5	6	7	8	9	10
A	x				x					
B	x				x	x	x			
C	x	x	x	x	x	x	x			
D					x					
E								x	x	x

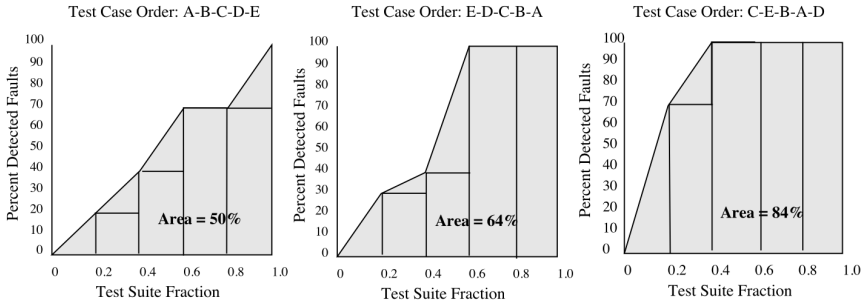

Figure 2.1: Average percentage of fault detection [10]

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.[1]

2.5 Approach Exploration

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.

2.5.1 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 [11] stated, computer resources grow quadratically with two multiplicative linear factors, making continuous integration a very expensive and not eco-friendly process in terms of resources and time.

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

2.5.2 Cumulative Approach

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

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

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

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

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

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

2.5.3 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 2.2.

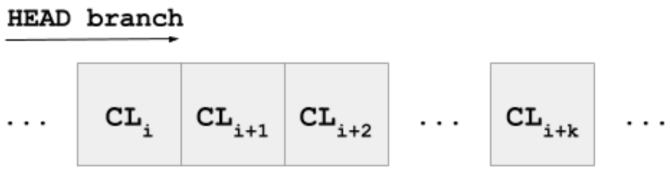

Figure 2.2: Code repository and workflow at Google. Each CL is submitted to master branch or HEAD [2]

Individually, tests validate each CL and if the outcome is pass, the change is incorporated into the master branch, resulting into a new version of the code. Otherwise, a regression is caused. For example, imagine that CL_i introduces a regression, causing failure on the consecutive system versions. Possible fixes and tips: investigate the modifications in CL_i, to obtain clues about the root cause of the regression; Debug the fail tests at CL_i, which is preferable to debug at later version CL_{i+k}, whereas after CL_i code

volume might increase or mutate, misleading conclusions; revert or roll back the repository, until the point where it was ”green”, in this case CL_{i-1}.[2]

The novelty is introduced in test validation:

- **Presubmit tests** - Prior to CL submission, preliminary relevant small/medium tests are run, if all pass the submission proceeds, else it gets rejected. So instead of having to wait until all tests are complete, the developer has a quick estimate of the status of the project.
- **Postsubmit tests** - After CL submission, large/enormous tests are executed and if they indeed cause a regression, we are again in the situation where we have to find exactly what CL broke the build and, possibly, proceed in the same manner as Cumulative Approach to detect what version caused the regression. [2]

Nevertheless there are some major issues: while presubmit testing, the sole focus is to detect flaws on individual changes and not on concurrent one. *Concurrency* is the term referred to builds that run in parallel with each other, meaning that 2 developers can be committing changes of the same file, that are in conflict with each other, or with other files. Therefore, after precommit phase, there is the need to check all the affected dependencies of every commit, quickly identify were the conflict emerged and eventually roll back in the case of a red master, which can only be done in postcommit stage [1] and when a postcommit test fails: a 45 minute test, can potentially take 7.5 hours to pinpoint which CL is faulty, given a 1000 CL search window, so automation techniques, quickly putting them to work and returning to a green branch is pivotal. [2]

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

2.5.4 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, [1] 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 2.3. 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 [1]. 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.

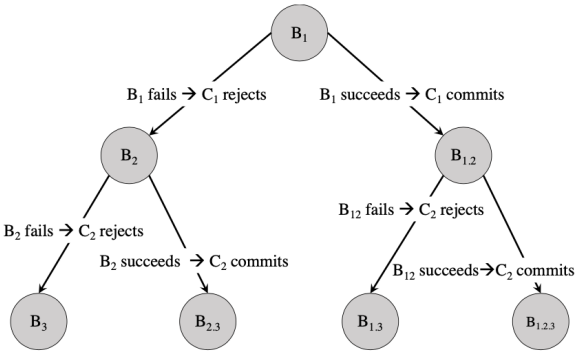

Figure 2.3: Speculation tree - builds and outcomes [1]

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

2.6 Machine Learning

Machine Learning algorithms are at the forefront in achieving effective automatic regression testing. By building predictive models, it is possible to accurately, and most importantly, quickly, identify if a

newly introduced change is defective or not. This helps developers to check and amend faults, when the details still remain fresh on their minds and it allows to prioritize and organize changes by the risk of being defective, this way testing the ones that are more likely to fail first, diminishing the lag-time between committing and feedback.

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

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

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

2.7 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. [12]

In accordance with Kamei et al. [13] and Yang et al. [14], the typical predictive model used is the Logistic Regression. Here, other models are explored, such as Decision Trees, Ensemble Methods and Artificial Neural Networks.

2.7.1 Logistic Regression

In classification, the goal is to come up with a rule to guess an outcome from a set of input variables. More precisely, when the output is binary, i.e. there are only two classes, this rule will predict, for example, if it will rain tomorrow, or not; or if a patient has a certain disease, or hasn't; or if the house mortgage will be payed, or not. Nonetheless, providing a "yes" or "no" answer to every observation is a quite rough estimate. - this type of classifiers are of type "hard" [15]. Especially in the presence of noisy data, adopting a "softer" approach to incorporate such nuances would be useful: like determining the probability that each data point has of being "0" or "1", rather than a binary output.[16] Logistic

Regression corresponds to the corner stone of ”soft” classifiers, which are, sometimes, more useful to resort to. There are situations where it is much more useful to know the likelihood/risk of something, instead of the actual values, thus having the possibility to rank observations in the most convenient way. [15] For example, an international bank desires to investigate which customers are more likely to leave. Considering certain features, like age, gender, country, number of credit cards, amount of debt, credit score, etc, the bank can train a model based on past history, to apply to customers that the model has not ”seen” and rank them by risk. Using such information to adapt and prioritize actions on customers that are more likely to leave.

To sum up, for a binary output variable Y, one needs to model the conditional probability $P(Y = 1|X = x)$ as function of x. A good starting point would be to try to do it by using a simple *linear regression*: (1) At first glance, one possible solution would be to let $p(x)$ be a linear function of x. An increment on the value of x would reflect on adding or subtracting a certain quantity to the probability. The drawback comes from the unboundedness of linear functions, and probabilities are bound between 0 and 1. (2) To solve the boundary problem, one can let $log\ p(x)$ be a linear function of x. Incrementing the value of x, will multiply the probability by a fix amount. But logarithms are only bounded in only one direction. (3) Finally, the most straight forward modification of (2) is the **logistic transformation**, $log(\frac{p}{1-p})$, which bounds the probability from 0 to 1, not compromising the interpretability of the result. [16].

Let $x_i = \{x_1, x_2, ...x_m\}$ is defined as the vector representation of features of a data point x, and x_i represents the value of the $i-th$ feature of x, and $W = \{w_0, w_1, w_2, ...w_m\}$ denotes the weight vector associated to the features in x, $w0$ is a bias parameter, and $w_i, i \in \{1, 2, ...m\}$ is the weight of the $i-th$ feature of x (i.e., x_i), and the let $y_i = \{y_1, y_2, ...y_m\}, y_i \in \{0, 1\}$ be the observed class.

Formally, the logistic regression model is given by:

$$y = log\left(\frac{p(x)}{1-p(x)}\right) = w_0 + x \cdot w \tag{2.1}$$

Solving for p, yields:

$$p(x, w_0, w) = \frac{e^{w_0+x\cdot w}}{1+e^{w_0+x\cdot w}} = \frac{1}{1+e^{-(w_0+x\cdot w)}} \tag{2.2}$$

$$p(y = 1) = 1 - p(y = 0) \tag{2.3}$$

Now, what is left is to estimate the parameters w_0 and w and this can be achieved by maximizing the likelihood function, so that the obtained statistical model best fits the observed data. The probability of an observed class y_i is either p, if $y_i = 1$, or $1 - p$, if $y_i = 0$. Then the likelihood is:

$$L(w_0, w) = \prod_{i=1}^{n} p(x_i)^{y_i}(1 - p(x_i))^{1-y_i} \tag{2.4}$$

For convenience, it is usual to write the *log-likelihood*: $log\left(L(w_0, w)\right)$:

$$l = \sum_{i=1}^{n} y_i log(p(x_i)) + (1 - y_i) log(1 - p(x_i)) \tag{2.5}$$

$$= \sum_{i=1}^{n} log(1 - p(x_i)) + \sum_{i=1}^{n} y_i log(\frac{p(x_i)}{1 - p(x_i)}) \tag{2.6}$$

$$= \sum_{i=1}^{n} log(1 - p(x_i)) + \sum_{i=1}^{n} y_i (w_0 + x_i \cdot w) \tag{2.7}$$

$$= \sum_{i=1}^{n} -log(1 + e^{-(w_0 + x \cdot w)}) + \sum_{i=1}^{n} y_i (w_0 + x_i \cdot w) \tag{2.8}$$

Now, to maximize the log-likelihood, one differentiates l, with respect to its parameters w_i. Differentiating with respect to one specific parameter w_j, the result obtained is:

$$\frac{\partial l}{\partial w_j} = -\sum_{i=1}^{n} \frac{1}{1 + e^{-(w_0 + x \cdot w)}} e^{-(w_0 + x \cdot w)} x_{ij} + \sum_{i=1}^{n} y_i x_{ij} \tag{2.9}$$

$$= \sum_{i=1}^{n} \Big(y_i - p(x_i, w_0, w) \Big) x_{ij} \tag{2.10}$$

Equalling the expression to zero and solving for w_j would return the optimized value for the parameter, however this equation cannot be solved analytically, only by a numerical approximation [16] The determination of the optimal parameters is what characterizes the training part of the algorithm.

2.7.2 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[12]:

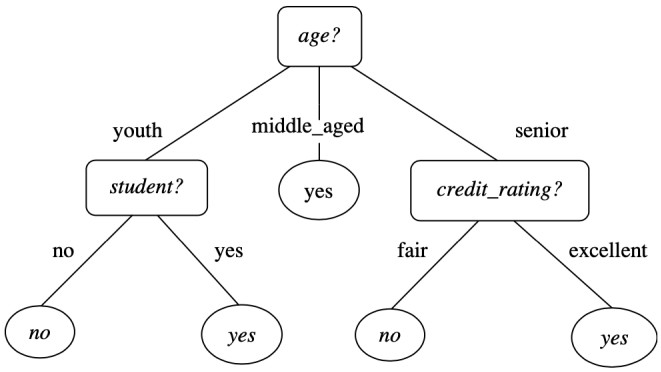

Figure 2.4: Each internal (non-leaf) node represents a test on an attribute. Each leaf node represents a class (if the client is likely to buy the product yes/no)

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:

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 $D_{rigth}(\theta)$.

$$D_{left}(\theta) = (x, y)|x_j \leq t_l \tag{2.11}$$

$$D_{right}(\theta) = D\ D_{left} \tag{2.12}$$

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.

$$G(D, \theta) = \frac{n_{left}}{N_l} H(G(D_{left}, \theta)) + \frac{n_{right}}{N_l} H(G(D_{right}, \theta)) \tag{2.13}$$

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:

$$H(D_m) = \sum_k p_{mk}(1 - p_{mk}) \tag{2.14}$$

as Entropy:

$$H(D_m) = -\sum_k p_{mk}(log(p_{mk}) \tag{2.15}$$

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:

$$\theta^* = argmin_\theta G(D, \theta) \tag{2.16}$$

Finally, recursively apply the same reasoning to subsets $D_{left}(\theta^*)$ and $D_{right}(\theta^*)$, until maximum

tree depth reached, i.e. $N_m < min_{samples}$ [17]

2.7.3 Ensemble Methods

An ensemble combines a series of k learned models (or base classifiers), M1, M2,..., Mk, with the aim of creating an improved composite classification model, M_*.

Ensemble methods are powerful tools for classification, namely *random forests* and *boosting*. An ensemble takes into account a series of k learned models $M_1, M_2, ..., M_k$, intersecting them into one, more robust, model M_*. A good analogy is to think of decision trees as having medical diagnosis from one doctor, that enquires and measures the patient's symptoms and then draws a conclusion. Now, ensembles take into consideration the opinion of several doctors and decides based on this information.

In this work, the ensemble method chosen is the Random Forest Classifier, that is an estimator that fits a various numbers of decision trees. [12]

to be continued

2.7.4 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. [12]

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. [12]

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.

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.

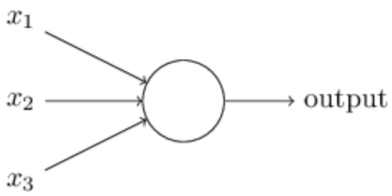

Figure 2.5: Perceptron simple example)

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 [18]. So the output can be defined as:

$$\textbf{output} = \begin{cases} 0, & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1, & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases} \tag{2.17}$$

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.

$$\textbf{output} = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases} \tag{2.18}$$

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.

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:

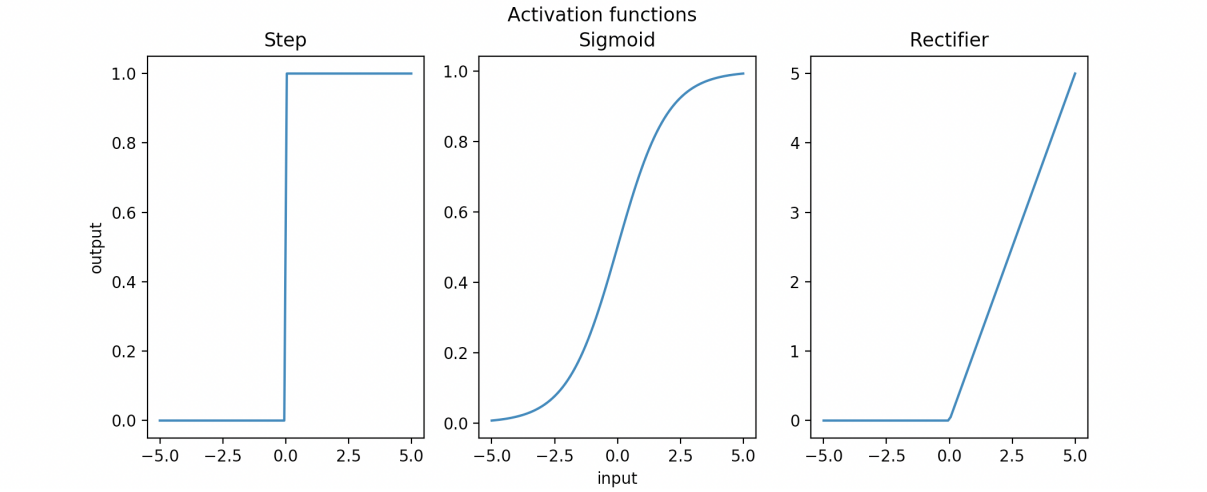

Figure 2.6: Comparative analysis of different activation functions (made in Python))

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.

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. [18].

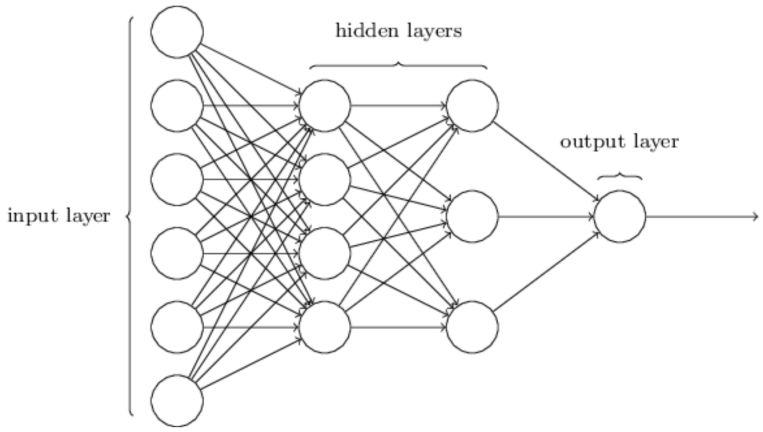

Figure 2.7: Architecture example of a NN [18]

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.[18]

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?

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.:

$$C(Y, \theta) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2 \tag{2.19}$$

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[15]. 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.

$$v_t = \eta_t \nabla_\theta C(Y, \theta) \tag{2.20}$$

$$\theta_{t+1} = \theta_t - v_t \tag{2.21}$$

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

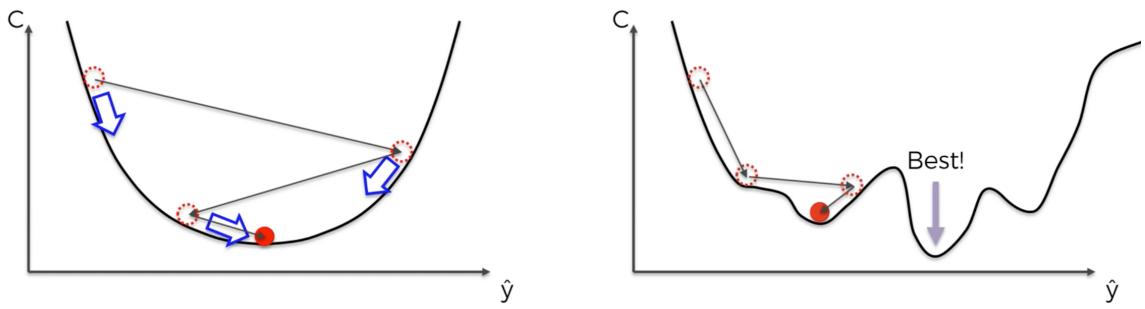

Figure 2.8: Global and Local minimum determination [19]

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. [15]

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. [18] 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:

$$\nabla_\theta C(Y, \theta) = \frac{1}{2m} \sum_{i=1}^{n} \nabla_\theta (\hat{y}^{(i)} - y^{(i)})^2 \rightarrow \frac{1}{2m} \sum_{i \in B_k} \nabla_\theta (\hat{y}^{(i)} - y^{(i)})^2 \tag{2.22}$$

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*. [15]

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.

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. [18]

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_j^l of the *j-th* neuron in the *l-th* layer, by recursively writing the relation to the activation a_j^{l-1} of neurons in the previous layer *l-1*.

$$a_j^l = \sigma\left(\sum_k \omega_{j,k}^l a_k^{l-1} + b_j^l\right) = \sigma(z_j^l) \tag{2.23}$$

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_j^L, 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_j^L:

$$\Delta_j^L = \frac{\partial C}{\partial z_j^L} \tag{2.24}$$

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

$$\Delta_j^l = \frac{\partial C}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l} = \frac{\partial C}{\partial a_j^l}\sigma'(z_j^l) \tag{2.25}$$

Now, one can relate the error function Δ_j^L to the bias b_j^l to obtain the second backpropagation relation:

$$\Delta_j^L = \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial b_j^l}\frac{\partial b_j^l}{\partial z_j^l} = \frac{\partial C}{\partial b_j^l} \tag{2.26}$$

Since, $\frac{\partial b_j^l}{\partial z_j^l} = 1$, from 2.23. Furthermore to find another expression for the error, one can relate the neurons in layer l with the neurons in layer $l+1$:

$$\Delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_k^l} \tag{2.27}$$

$$= \sum_k \Delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_k^l} \tag{2.28}$$

$$= \left(\sum_k \Delta_k^{l+1}\omega_{k,j}^{l+1}\right)\sigma'(z_j^l) \tag{2.29}$$

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

$$\frac{\partial C}{\partial \omega_{j,k}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial \omega_{j,k}^l} = \Delta_j^l a_k^{l-1} \tag{2.30}$$

The combination equations 2.25, 2.26, 2.29 and 2.30, constitutes the backbone of the backpropagation algorithm that is used to train NN's in a very efficient way [15]. 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 2.23, 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 2.26.
4. **Propagate the error "backwards"** - calculate the error Δ_j^l, for all layers l, using equation 2.29.
5. **Determine gradient** - use equation 2.26 and 2.30 to calculate $\frac{\partial C}{\partial b_j^l}$ and $\frac{\partial C}{\partial \omega_{j,k}^l}$.

[15]

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. [18]

2.7.5 Model Evaluation and Parameter Tuning

2.7.6 AutoEncoders

this part is a bit experimental, autoencoders can be used for feature extraction. For example logistic regression has the limitation of only being accurate with linearly separable data. Autoencoders and Deep Boltzmann Machines can be used to combine features with each other and produce more expressive and relevant ones, that can later serve as input for a classifier . (Im very exciting about this part)

Chapter 3

Implementation

3.1 Synthetic Data Generation

On a first level, the strategy is focused on trying to mimic a real life environment and try to find correlations between commits and regressions, by creating a data set that contains: (1) A list of commits, containing information of what/when files have been modified, and (2) a list of tests, with the associated duration and files covered. After that, we can experiment the different approaches explored above and combine some aspects to efficiently have a scalable Continuous Integration system. Firstly, the determination of automated techniques that facilitate an early detection, for example:

- **Commit List**

  - when someone makes a change that caused a regression, subsequent commits are prone to affect the same tests

  - some tests might be an indicator of whether other tests will fail

  - keeping a score board, based on past-history, to estimate the probability a given developer has of making a faulty committing.

- **Test List**

  - modifying a file might trigger off faults on files that depend on the first

  - some files affect many tests, but can be very stable and not cause failures often. - (for instance standard libraries)

  - other files, that affect few tests, are harder to cover and validate. - (like an innovative feature)

Finally, a test run list is generated to analyse the results of the approaches, this list is expected to contain a live-estimate of the status of the project as tests are being run. At the beginning, the value for the uncertainty that commit will pass has its maximum value and the goal is to choose what test to apply next, in order to minimize this uncertainty and give the estimate between a confidence value, being more accurate as more tests are applied. This way it is possible to keep up with the projects' progress as time evolves.

3.2 Numerical Model

Description of the numerical implementation of the models explained in Chapter 2...

3.3 Verification and Validation

Basic test cases to compare the implemented model against other numerical tools (verification) and experimental data (validation)...

Chapter 4

Results

Insert your chapter material here...

4.1 Problem Description

Description of the baseline problem...

4.2 Baseline Solution

Analysis of the baseline solution...

4.3 Enhanced Solution

Quest for the optimal solution...

4.3.1 Figures

Insert your section material and possibly a few figures...

Make sure all figures presented are referenced in the text!

Images

Make reference to Figures **??** and **??**.

By default, the supported file types are *.png,.pdf,.jpg,.mps,.jpeg,.PNG,.PDF,.JPG,.JPEG*.

See `http://mactex-wiki.tug.org/wiki/index.php/Graphics_inclusion` for adding support to other extensions.

Drawings

Insert your subsection material and for instance a few drawings...

The schematic illustrated in Fig. 4.1 can represent some sort of algorithm.

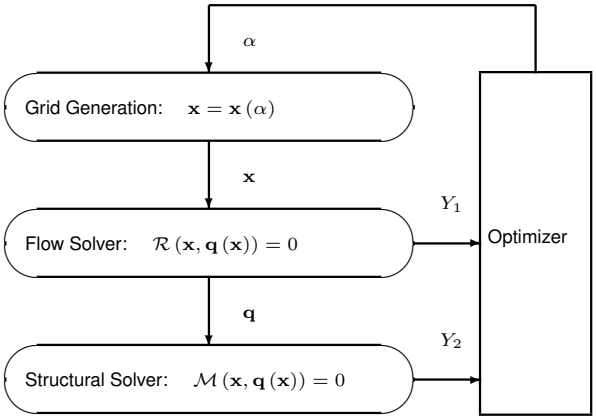

Figure 4.1: Schematic of some algorithm.

4.3.2 Equations

Equations can be inserted in different ways.

The simplest way is in a separate line like this

$$\frac{\mathrm{d}q_{ijk}}{\mathrm{d}t} + \mathcal{R}_{ijk}(\mathbf{q}) = 0\,. \tag{4.1}$$

If the equation is to be embedded in the text. One can do it like this $\partial\mathcal{R}/\partial\mathbf{q} = 0$.

It may also be split in different lines like this

$$\begin{aligned} \text{Minimize} \quad & Y(\alpha, \mathbf{q}(\alpha)) \\ \text{w.r.t.} \quad & \alpha\,, \\ \text{subject to} \quad & \mathcal{R}(\alpha, \mathbf{q}(\alpha)) = 0 \\ & C(\alpha, \mathbf{q}(\alpha)) = 0\,. \end{aligned} \tag{4.2}$$

It is also possible to use subequations. Equations 4.3a, 4.3b and 4.3c form the Naver–Stokes equations 4.3.

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_j}(\rho u_j) = 0\,, \tag{4.3a}$$

$$\frac{\partial}{\partial t}(\rho u_i) + \frac{\partial}{\partial x_j}(\rho u_i u_j + p\delta_{ij} - \tau_{ji}) = 0, \quad i = 1, 2, 3\,, \tag{4.3b}$$

$$\frac{\partial}{\partial t}(\rho E) + \frac{\partial}{\partial x_j}(\rho E u_j + p u_j - u_i \tau_{ij} + q_j) = 0\,. \tag{4.3c}$$

4.3.3 Tables

Insert your subsection material and for instance a few tables...

Make sure all tables presented are referenced in the text!

Follow some guidelines when making tables:

- Avoid vertical lines
- Avoid “boxing up” cells, usually 3 horizontal lines are enough: above, below, and after heading
- Avoid double horizontal lines
- Add enough space between rows

Model	C_L	C_D	C_{My}
Euler	0.083	0.021	-0.110
Navier–Stokes	0.078	0.023	-0.101

Table 4.1: Table caption.

Make reference to Table 4.1.

Tables 4.2 and 4.3 are examples of tables with merging columns:

	Virtual memory [MB]	
	Euler	Navier–Stokes
Wing only	1,000	2,000
Aircraft	5,000	10,000
(ratio)	$5.0\times$	$5.0\times$

Table 4.2: Memory usage comparison (in MB).

	$w = 2$			$w = 4$		
	$t = 0$	$t = 1$	$t = 2$	$t = 0$	$t = 1$	$t = 2$
$dir = 1$						
c	0.07	0.16	0.29	0.36	0.71	3.18
c	-0.86	50.04	5.93	-9.07	29.09	46.21
c	14.27	-50.96	-14.27	12.22	-63.54	-381.09
$dir = 0$						
c	0.03	1.24	0.21	0.35	-0.27	2.14
c	-17.90	-37.11	8.85	-30.73	-9.59	-3.00
c	105.55	23.11	-94.73	100.24	41.27	-25.73

Table 4.3: Another table caption.

An example with merging rows can be seen in Tab.4.4.

If the table has too many columns, it can be scaled to fit the text widht, as in Tab.4.5.

ABC	header			
	1.1	2.2	3.3	4.4
IJK	group		0.5	0.6
			0.7	1.2

Table 4.4: Yet another table caption.

Variable	a	b	c	d	e	f	g	h	i	j
Test 1	10,000	20,000	30,000	40,000	50,000	60,000	70,000	80,000	90,000	100,000
Test 2	20,000	40,000	60,000	80,000	100,000	120,000	140,000	160,000	180,000	200,000

Table 4.5: Very wide table.

4.3.4 Mixing

If necessary, a figure and a table can be put side-by-side as in Fig.**??**

Chapter 5

Conclusions

Insert your chapter material here...

5.1 Achievements

The major achievements of the present work...

5.2 Future Work

A few ideas for future work...

Bibliography

[1] S. Ananthanarayanan, M. S. Ardekani, D. Haenikel, B. Varadarajan, S. Soriano, D. Patel, and A.-R. Adl-Tabatabai. Keeping master green at scale. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 29:1–29:15. ACM, 2019. This source encompasses Uber's repository configuration and explains what's behind their fault detection process. Very well structured paper, with guidelines of future work that can be explored, mainly power their system with other machine learning techniques.

[2] C. Ziftci and J. Reardon. Who broke the build? automatically identifying changes that induce test failures in continuous integration at google scale. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP '17, page 113–122. IEEE Press, 2017. ISBN 9781538627174. doi: 10.1109/ICSE-SEIP.2017.13. URL https://doi.org/10.1109/ICSE-SEIP.2017.13.

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

[4] 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.

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

[6] 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.

[7] Y. Shin. *Extending the Boundaries in Regression Testing: Complexity, Latency, and Expertise*. PhD thesis, King's College London, 2009. PhD thesis containing refined detail regarding test case management. In a simple, yet very complete way, provides thorough background on several techniques

to reduce, select and prioritise the way tests are applied, which corresponds to a significant part this work intents to achieve.

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

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

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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`.

[16] C. R. Shalizi. Advanced data analysis from an elementary point of view. In *Advanced Data Analysis from an Elementary Point of View*, 2012.

[17] 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`.

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

[19] 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

Vector calculus

In case an appendix if deemed necessary, the document cannot exceed a total of 100 pages...

Some definitions and vector identities are listed in the section below.

A.1 Vector identities

$$\nabla \times (\nabla \phi) = 0 \tag{A.1}$$

$$\nabla \cdot (\nabla \times \mathbf{u}) = 0 \tag{A.2}$$

Appendix B

Technical Datasheets

It is possible to add PDF files to the document, such as technical sheets of some equipment used in the work.

B.1 Some Datasheet