Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique

Université de Carthage

Ecole Polytechnique de Tunisie

وزارة التعليم العالي و البحث العلمي

جامعة قرطاج

المدرسة التونسية للتقنيات

Graduation Project Report

to obtain:

The National Engineering Diploma from the Ecole Polytechnique de Tunisie

Optimizing Continuous Integration Using Artificial Intelligence

Elaborated by:

Maher Dissem

within:

Supported day/month/year before the examination board

Academic Year: 2021- 2022

Acknowledgements

I would like to express my gratitude and thanks to all those who helped me, encouraged me and supported the realization of this project and the smooth running of the internship and especially my advisor, Dr. Mohamed Aymen Saied, for his follow-up, his help and the quality of the supervision he gave me. I would also like to thank all the other members of the laboratory for their help.

As I near the end of my journey as an engineering student at EPT, I would also like to express my gratitude to everyone who made these last three years so memorable and productive, therefore I give my thanks to all the teachers of EPT and especially Pr. Naceur Benhadj Braiek, my academic supervisor, for his guidance and his availability.

Abstract

Continuous Integration is a set of modern Software Development practices widely adopted in commercial and open source environments that aim at supporting developers in integrating code changes constantly and quickly through an automated building and testing process. This enables them to discover newly induced errors before they reach the end-user.

However, this build process is typically time and resource-consuming, specially for large projects with many dependencies where the build process can take hours or even days. This may cause disruptions in the development process and delays in the product release dates.

One of the main causes of these delays is that some commits needlessly start the Continuous Integration process, therefore ignoring them will significantly lower the cost of Continuous Integration without sacrificing its benefits.
Thus, withing the framework of this project, we leverage a novel Deep Reinforcement Learning technique to build an optimal classifier that automatically detects unnecessary commits and skips their execution.

Keywords: Software Engineering, Continuous Integration, Artificial Intelligence, Deep Reinforcement Learning.

Table of contents

List of figures

List of tables

List of Algorithms

General Introduction

Software development practices have advanced and evolved considerably over the years due to the growing demand for more robust products and faster release cycles. And continuous integration (CI) is one of the key modern practices used in software development.
It urges developers to build and test their software whenever a new commit is submitted, in order to discover new bugs as they arise.
While this practice offers many benefits such as faster failure detection, reduced cost and risk of delivering defective changes, and improved code quality, it also induces substantial costs due to the computational resources required to frequently run and test new versions of the software and thus adopting CI can be very expensive.
Long build times can also prevent developers from proceedings further with development, indeed developers may, for instance, work on specific tasks in separate branches using a version control system. If a build fails, it would be cumbersome to switch back to the original branch due to several reasons such as caching, configuration files, and so on. This is why, they tend to wait for the build process to finish, which severely affects both, the speed of software development and the productivity of developers.

Several researchers have proposed solutions to reduce this cost, without sacrificing the advantages offered by CI. This issue has been addressed in different ways, ranging from selecting and prioritizing tests to optimizing builds.
Recently, it has been proven that builds with minor modifications are more likely to pass without errors, therefore omitting them will dramatically lower the cost without sacrificing the benefits of Continuous Integration.
This is why, we aim in this project to optimize the continuous integration process by skipping the execution of builds that do not provide valuable insight to developers.
Thus, the problem lies in correctly predicting whether a new build will fail or succeed without running it.

The report is organized as follows:

- **Chapter 1:** This chapter presents the host organization, the general context of the projects and its goals, as well as the main concepts tackled within this study.
- **Chapter 2:** The second chapter reviews existing research that address our task and similar topics, and explains how we made use of some of their ideas. It also introduced the theoretical details of our own approach.
- **Chapter 3:** This last chapter presents the work done. we start by explaining how our novel model works, and then move on to perform a comparison with state-of-the-art techniques.

Chapter 1

General Framework of the Project

1.1 Introduction

Within a software development context, as projects become more complex with time and the need for more functionalities, the risk of human errors causing consequent defects or software failures grows alarmingly.
Such software flaws or delivery delays can harm a company's reputation, resulting in dissatisfied and lost consumers. Moreover, in extreme cases, a bug can impair interconnected systems or create significant problems.
This is why, it is recommended for companies to set up an automated testing process to identify and address potential flaws in their software before distributing or updating it. This process lets them save millions of dollars per year in development and support, provided they have a good testing and QA processes.

In this first chapter, we will see how Continuous Integration is utilized to ensure software quality, the major drawbacks it induces and how we plan to optimize it. We will therefore start by presenting the host group for this internship. We will then introduce the project as well as the problem it tackles. Moreover, we will explain the basic needed concepts of Software Engineering and Artificial Intelligence.

1.2 Host Organization Presentation

Laval University, located in Québec City, Québec, Canada, is a public research university. it was created by a royal charter issued by Queen Victoria in 1852, making it the oldest center of higher education in Canada and the first North American institution to offer higher education in French.

This project was carried out as part of an end-of-studies research internship within the university's department of Computer Science and Software Engineering whose research areas cover a broad spectrum of computer science topics, ranging from Bioinformatics to Robotics, however our project fits under the Software Engineering research activities of the department.

Fig. 1.1 Logo of Laval University

1.3 Context of the Project

1.3.1 Continuous Integration and its Benefits

Continuous integration (CI) is a cutting-edge software development practice widely used in industry and open-source environments.
In a CI context, developers integrate their changes to the source code frequently and in smaller increments, rather than integrating a large change at the end of a development cycle because error-prone modifications that are smaller and more frequently integrated are easier to debug when an error arises as developers need to deal with a much smaller number of changes.
Each integration is verified by an automated building and testing tool to detect integration errors as quickly as possible. [15]
This approach leads to a lower risk and cost of delivering flawed updates and allows a team to develop cohesive software more rapidly because discovering errors as they arise and knowing how to reproduce the issue thanks to the results of the executed tests, allows them to save significant development time that would otherwise be wasted on debugging an unknown error.

1.3.2 General Workflow of Continuous Integration

Modern software development relies on the use of a Version Control System (VCS) such as Git to record changes made to the code over time. Furthermore, multiple copies

of the codebase or ’branches’ are made in order to work in parallel and to separate in-progress work from tested and stable code.
In Continuous Integration, developers are encouraged to merge their changes to the main branch very frequently.
When a developer is ready to merge a commit (i.e. changes to the project’s files) from his development branch into the main codebase or the master branch, a pull request is created.
At that point, the CI system will proceed to clone the project’s repository into a brand-new virtual environment. After installing all the required dependencies, the code is built, followed by running a test sequence submitted by the QA team.
Moreover, it is a recommended practice to continuously develop and increase test coverage. Each new feature that passes through the CI pipeline should be accompanied by a set of tests that ensure the new code is performing as expected.
If a build fails, the CI system prevents it from moving on to the deployment stage. The team receives a report and attempts to resolve the issue.
However, a successful CI build will lead to the subsequent stages of continuous delivery. It is at this point that the changes will be verified by another developer in a peer review. This developer may then merge the new branch of code into the production branch and thus integrate the new changes into the deployed application. [4]
This way, developers can iteratively build new features and usually discover and fix errors before they reach the end users.
This general workflow is illustrated in Fig. 1.2.

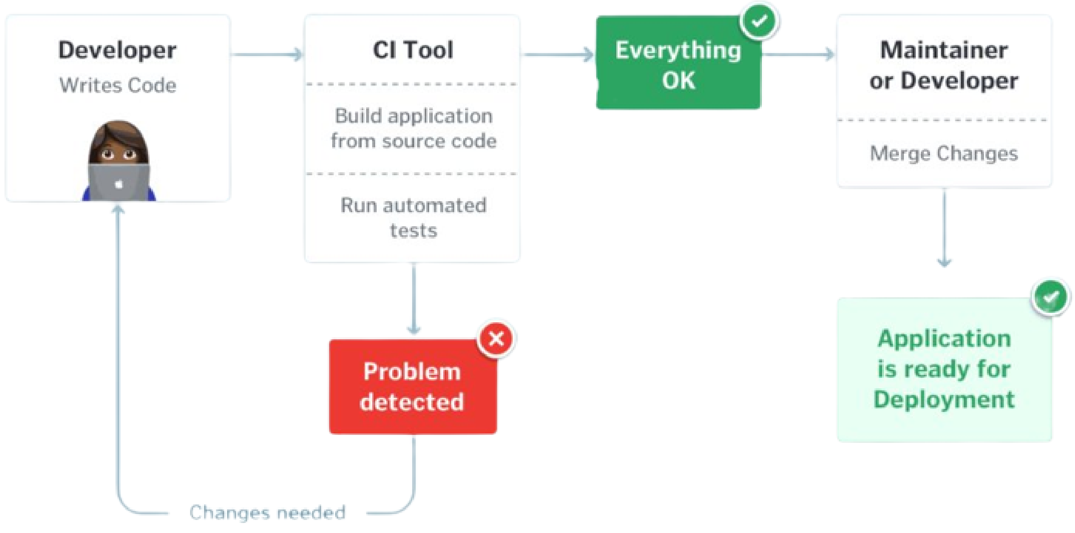


Fig. 1.2 Usual continuous integration workflow

1.4 Description of the Project

1.4.1 Problem Statement

The build process is considered as a serious challenge that hinders CI adoption. In fact, the build process is often time and resource intensive, and it is especially difficult for large projects with multiple dependencies, whose builds might take hours or even days to complete.
The presence of such slow builds has a severe impact on both the speed and cost of software development, as well as the productivity of developers.

1.4.2 Project's Objectives

One of the main reasons for the slow builds and for the delays introduced by CI is that some commits (e.g. adding comments) unnecessarily trigger the CI process. Executing these commits does not provide any particular insight to developers, and vainly uses computation resources.
This is why, this work aims to employ artificial intelligence techniques to automate the process of selecting which commits may be CI skipped while still executing as many failing builds as early as possible to let developers know when a critical change was introduced.

In other words, our goal here is to create an AI system that will be placed before the 'CI Tool' box in Fig. 1.2. It will be triggered when a git commit is pushed to the remote repository. It will then analyze the commit and generate the descriptors needed for the prediction.
If the commit should be CI skipped, this system will prompt the developer to CI-skip the commit giving him an explanation of its decision, if the developer accepts to do so, it will modify the commit message to add the [ci skip] tag into the commit message.
This way, when the CI tool (e.g. Travis CI) is triggered, it will examine the commit message and when it sees this particular tag, it knows that it should skip the build process.

1.4.3 Which Commits to CI skip?

In order to get insight as to which commits developers tend to CI-skip in real world scenarios, we refer to an investigation [2] of 58 open source projects where developers actively apply continuous integration's principles, including the manual use of the [ci

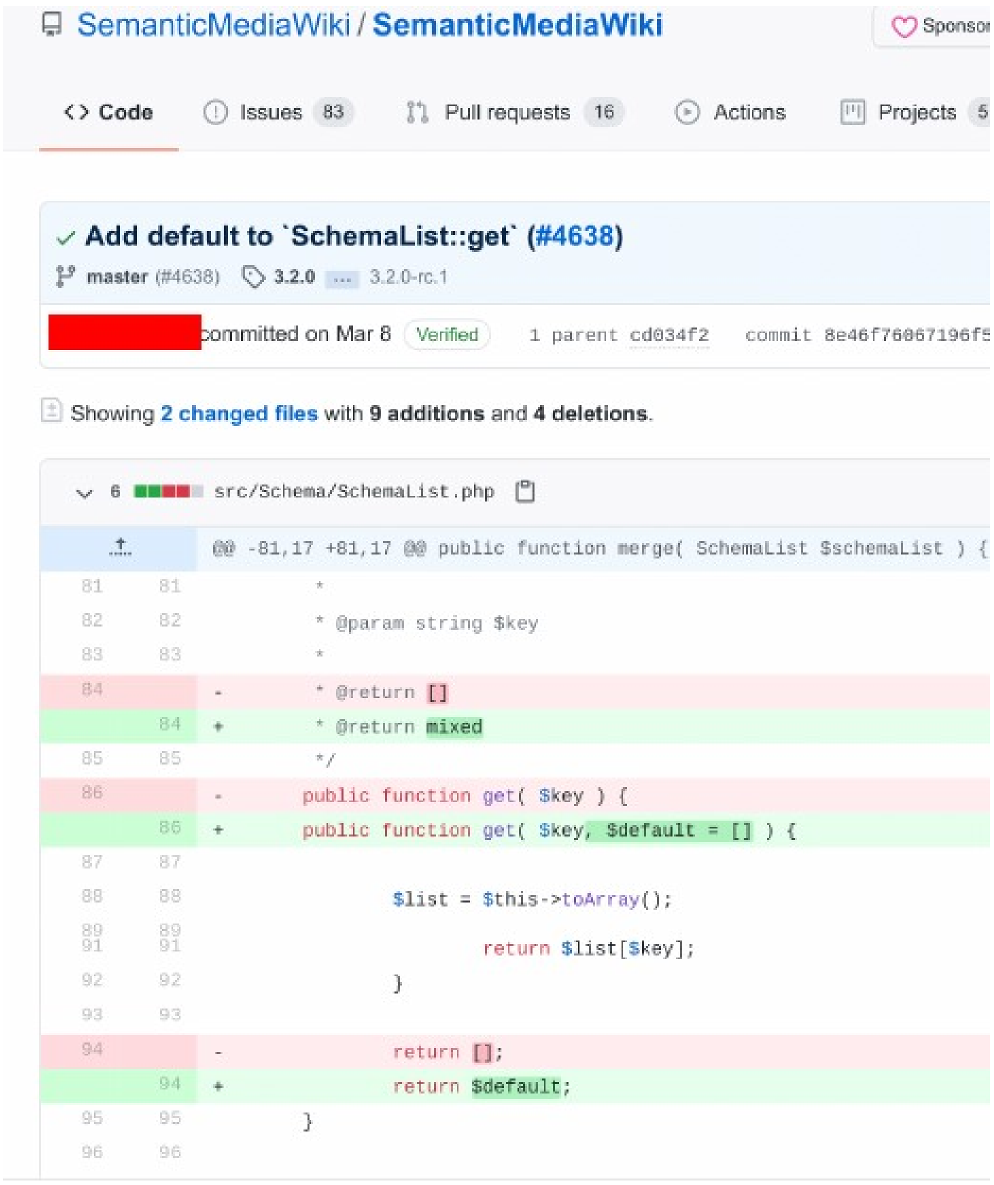
skip] tag to exclude certain commits from the build process.
In fact, the CI-skipped commits within these projects were picked out and by analyzing each commit's meta-data (e.g. commit message, etc.) and the source code changes, the reason for the commit being skipped was identified. These reasons, the percentage of each category as well as the type of information needed to make a decision on whether to CI-skip commits from a particular, are listed in Table 1.1.

Table 1.1 Manually Extracted Reasons of CI Skipped Commits.

Reason	Percentage	Information Source
Non-Source code files	52.01%	Repository
Version preparation	15.11%	Repository
Source code comment	6.01%	Repository
Metal files	3.75%	Repository
Formatting source code	1.16%	Repository
Source code	10.54%	Developer
Build change	6.18%	Developer
Tests	1.00%	Developer
Other	10.48%	Various

Given this data, We may conclude that the most common types of changes that developers tend to skip are changes to non-source code files, version preparation, source code comments, meta files, and source code reformatting.
These categories represent more than 70% of skipped commits, and deciding to CI-skip those do not require an in-depth analysis of the code or specific knowledge from the developer like it does for the case of source code changes. In fact, for these cases, the decision to skip the build can be inferred exclusively from information about the project's repository.
In other words, we can automatically detect when to CI-skip such commits without having to perform an analysis of the specific changes made but only relying on information that can be extracted from the repository and the commit, such as the type of modified files or the number of comments added.

However, developers can also choose to skip testing source code changes when they know that the change made will not impact the execution of the system or when the changes are not covered by CI tests.
A concrete example of this particular case, taken from the GitHub repository of the PHP framework Semantic MediaWiki is shown in Fig. 1.3.



✓ Add default to `SchemaList::get` (#4638)

master (#4638) 3.2.0 … 3.2.0-rc.1

[redacted] committed on Mar 8 Verified 1 parent cd034f2 commit 8e46f76067196f5

Showing 2 changed files with 9 additions and 4 deletions.

6 src/Schema/SchemaList.php

```
@@ -81,17 +81,17 @@ public function merge( SchemaList $schemaList ) {
           *
           * @param string $key
           *
-          * @return []
+          * @return mixed
           */
-         public function get( $key ) {
+         public function get( $key, $default = [] ) {

                  $list = $this->toArray();
                          return $list[$key];
                  }

-                 return [];
+                 return $default;
          }
```

Fig. 1.3 Example of source code modification that can be CI skipped

This commit, despite modifying the source code, can be skipped [19].
Indeed, the developer only added a default parameter $default to the signature of the modified function to be returned if the $key parameter is not passed as an argument. This change has no effect on the behavior of the function or the caller functions.
For our system to be able to account for such commits, it needs to be trained on features that describe the code change and its impact.

1.4.4 Impact of the Project

Economically, there is a lot at stake about optimizing the continuous integration process, as it is a fairly expensive practice. In fact, Google, for instance, estimates the cost of running its CI system in millions of dollars [16] and for smaller-budget companies that have not yet implemented CI, the high cost of constantly running builds can pose a strong barrier to its adoption.
Long build times are also bothersome from the perspective of developers, because often, they would rather focus on a single task and therefore wait for the build results to finish before moving on to the next task or continuing with the present one despite being less productive as it lets them avoid multitasking and context switching which, for complex tasks, can be a drain on the developers' mental energy. [12]

Additionally, we argue that an ideal system would decide to CI-skip any commit that if executed will not trigger a build failure, because what's valuable from a developer's point of view is to know how a commit can lead to a failure, the type of error, and which test covers this particular case. In this context, a survey of the causes of broken builds [6] examined the build outcome of commits from 1,108 Java and Ruby projects that use Continuous Integration. Their results are shown in Fig. 1.4.
The build result distribution is relatively similar for the two programming languages, with build cancels being rare and build failures being caused by infrastructural problems in around 5% of cases. Moreover, failures during the build process are responsible for most broken builds, and the largest portion of built commits succeed.
This data imbalance is an important trait of the CI skip usage as that we will work on because the option to manually decide to CI skip a commit is generally underused by developers, as for projects using a CI system, only 3% of commits include this tag [2]. However, [11] showed that most developers may not know about the different features of CI tools, such as the possibility to skip commits, which explains this low percentage. In sum, the major part of total commits should be CI skipped, and consequently, by

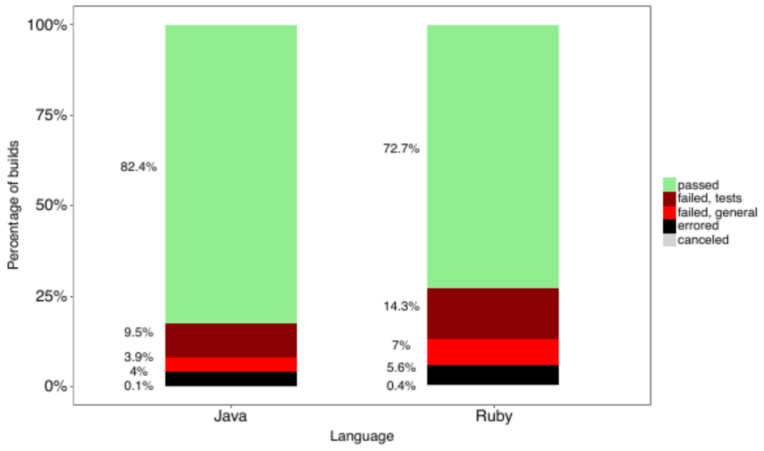


Fig. 1.4 Distribution of build status for over 1000 Java and Ruby projects

successfully skipping the build process of these commits, we can certainty save a lot of computational power, development time and thus financial resources.

1.5 Basic Machine Learning Concepts

Machine learning is a branch of artificial intelligence and computer science built around statistical methods and algorithms which focuses on the use of data to imitate the way that humans learn, gradually improving its performance.
There are three different types of Machine Learning:

Supervised Learning We are provided with a Dataset where we suppose that there is a relationship between the Input (Features) and the Output (Labels). Besides, we already know what our right output should look like. Supervised Learning problems are divided into Regression problems where we want to predict a continuous variable and Classification problems where our goal is to assign a label to a data point.

Unsupervised Learning We are given a dataset in which we have no idea whether the input and outcome are related. Furthermore, we have no notion about how our final product should look like. That is, we can extract relevant information from data even if the importance of its variables is unknown.

Reinforcement Learning RL aims to build a decision-making agent by teaching it through trial and error. Unlike Supervised Learning and Unsupervised Learning, we are not given a dataset.

Reinforcement Learning tries to enhance the model by rewarding positive actions and punishing negative ones. To accomplish this, RL must be able to sense signals from its environment, decide on an action autonomously, and then compare the outcome to a "reward" specification. The RL agent tries then to figure out what to do to maximize the reward, but it does so on its own (with no direct instructions).
More details will be further explained in section 2.5.

1.6 Conclusion

To summarize, through Continuous Integration allows companies to save a lot in terms of development time and cost by avoiding releasing defective software updates. This is done by testing the software build after every modification. However, the testing process itself costs significant amounts of money and computational time. We argue therefore that a promising strategy to optimize this process would be by skipping testing trivial code modifications that will not cause problems when integrated with the main codebase.

Within this chapter, the general framework of the project was introduced. We started by presenting the group where the project was held. We then introduced the problem statement, and the project's objectives and its impact, as well as some basic notions.
In the following chapter, we'll see how other researchers approached this particular problem and how their ideas contributed to shaping our own novel optimization strategy.

Chapter 2

Background and Related Works

2.1 Introduction

In recent years, AI's influence across a wide range of industries has risen significantly, with an increasing number of AI solutions being proposed to handle and automate various tasks. Software engineering and development is one of the sectors that has reaped significant benefits from the rise of AI, as many of its aspects are now being optimized and enhanced using AI to minimize human intervention.
For instance, the way we develop applications is being revolutionized with the addition of code completion and code generation tools such as Kite or Codota that assist developers with trivial tasks and let them focus on more complex ones. [21] Similarly, other tools such as DeepCode are trained to conduct code reviews and identify issues. Hence, it boosts the productivity of the whole team by providing fast and objective recommendations, saving them from having to schedule review meetings. Furthermore, Amazon's CodeGuru, in addition to reviewing code, is able to identify potential security vulnerabilities.
On a related note, AI is also able to capture user data and use Machine Learning to distinguish anomalous behavior from typical behavior. It can therefore observe and detect malware before it infiltrates the system by leveraging AI assisted pattern recognition techniques.
The planning aspect of software development also profits from such new technologies, as AI models trained on data from past projects can be used to provide good estimates on timelines, costs and efforts required and thus aid making strategic decisions.
Besides, as more and companies are modernizing their IT infrastructure, AI-based techniques have been proposed to assist with the migration process. For systems moving from monolithic software architectures to a microservices-based architecture, for

instance, DRL-based models have been proposed to automatically identify suitable microservices and map existing components to them.

Regarding continuous integration, although numerous techniques have been proposed over the time to tackle its optimization, two main strategies dominates the literature. The first one suggests reducing the size of the test set run after building the project by selecting, after each commit, the tests that are most relevant to the changes made so that potential test failures are still caught while reducing the length of the testing process. [22]
Within the scope of our project, however, we are more concerned about the second approach, described in the previous chapter, which resolves around predicting which commits do not induce changes significant enough to re-run the whole build and test process.
We will therefore, in this chapter, review how other researches approached this task. We first mention the works that laid the ground for data-driven research on continuous integration in general by generating commit-level features that may be used as dependent variables for classifying commits. We then move on to explaining the details of some of the most promising work done in predicting the outcome of CI builds. Finally, we delve into the theoretical details and the mathematical foundation of the techniques adopted in our own solution.

2.2 Generating Commit Features

Continuous integration is relatively a modern practice that rapidly spread across the software development realm. However, even today, there is still a lack of quantifiable evidence regarding the implications of introducing and using CI from an academic point of view. [7]
In order to get insight into the impact of CI, three major data sources should be considered as they are the most popular in their respective role and can provide different kinds of information.

- **The GIT version control system:** a tool for tracking changes in any set of files, typically used to coordinate work among programmers who are working collaboratively.
- **The GitHub collaboration platform:** the most popular provider for internet hosting of software development projects, it includes the distributed version control and source code management functionality of Git, plus its own access

control and collaboration features. It can be mined for data using GHTorrent, a queriable, offline mirror of data offered through the GitHub REST API [9].

- **The Travis CI environment:** the most popular CI system, accounting for roughly 50% of the CI market on GitHub [5]. In fact, any GitHub repository can be set up to generate CI builds automatically. It offers both a free service for open source projects and a paid service for private projects.
  Meta-data about CI builds can be accessed using Travis CI's API.

To determine how effective the ML techniques are in detecting CI skip commits, we need to have a labeled test dataset that we can apply the ML techniques on.
We first thought about building our own dataset using these data sources, however, due to the APIs' restrictions on frequently accessing them to collect meta-data and due to the algorithmic complexity of collecting and combining the extracted information for a large number of projects, we were inclined into looking for an existing dataset that we can employ in our research.

This is why we opted to use the dataset proposed in the research paper of Abdalkareem et al. [1] which built a Decision Tree classifier to detect CI-skipped commits on 23 features mined from GitHub repositories.
The process of filtering available data is detailed in Fig. 2.1.
In order to arrive at reliable and generalizable conclusions, it is crucial to operate on a

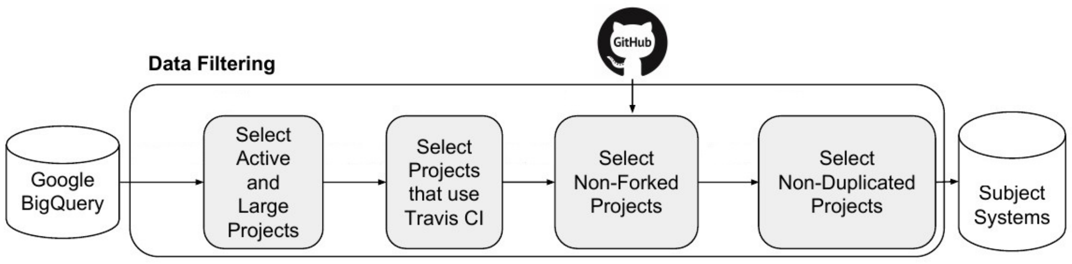

GitHub
Data Filtering
Google BigQuery
Select Active and Large Projects
Select Projects that use Travis CI
Select Non-Forked Projects
Select Non-Duplicated Projects
Subject Systems

Fig. 2.1 An overview of the data filtering steps [add steps]

diverse and large set of software projects. GitHub being the platform hosting the larger number of open-source projects, we naturally choose to mine it for data.
We begin by using Google BigQuery to query the public GitHub dataset, which provides a web-based terminal for running SQL queries on the GitHub data. Our first criteria is to select projects that use Travis CI. This is done by determining projects that contain the .travis.yml configuration file.
Then, we choose to only keep projects that feature the [ci skip] commit message tag

in at least of 10% of their commits so that we ensure that the subject developers are familiar with the CI skip feature.
We then proceed to remove inactive and small projects from the corpus by selecting a minimal threshold that the total number of commits in every project should satisfy. Moreover, users can duplicate a repository on GitHub to make changes without affecting the original project. They later can submit Pull Requests to contribute their changes to the original project. The same development activity will therefore be counted multiple times. This is why the final step is to remove repositories that are flagged as forks according to the GitHub API.

Now that we have a set of projects to study, we need to extract features that describe each commit.
Abdalkareem et al. [13] analyzed the commit history of all the branches for each project and extracted various metrics, which include, the type of change performed in each commit (i.e., does the commit modify source code, modify the formatting of the source code, or modify source code comments) and the type of files modified in each commit (provided by their file extensions).
This is done using Commit Guru [17], a language agnostic, publicly available analytics tool that mines Git repositories and generates metrics at both the repository and the commit level.

An alternative dataset that we can use is the TravisTorrent dataset [7], which synthesizes information from the sources mentioned earlier and provides access to hundreds of thousands of analyzed builds from more than 1,000 open-source projects, where each data point represents a build job executed on Travis CI. This dataset was not created specifically for the CI skip detection task. Instead, by providing general features about the build process, it was used in multiple different tasks such as forecasting the build duration, or the number of builds over a period of time. Nevertheless, since it provides commit-level features, it can be used in our case for a similar aim; predicting the outcome of builds (failing or passing) as we argue that passing build should be CI-skipped.

2.3 Predictive Models

2.3.1 Evaluation Methodology

In order to review and compare research projects found in the literature, it is essential first define how we are able to quantify the performance of a given CI-skip detection solution, and how we should conduct its evaluation.
Ultimately, our goal is to decide, given a certain commit, if the CI system in-use should build the project and run the test set or if we can skip the build execution. This problem can be modelled by a binary classification problem, where we give a label having two possible values to commits. Therefore, we can apply the usual metrics used in classification tasks. However, there are several metrics that can be used, we will therefore focus on metrics that are most relevant to imbalanced classification tasks like ours.
Furthermore, it is important for us to use, if possible, the evaluation metrics that are most widely used in the literature when dealing with CI-skip detection so that we can later make a fair comparison with our solution.

Before we proceed to describe potential performance metrics, let's start by defining the possible classification results and their meaning. A handy way to visualize a classification's model output, on a collection of test data, is to use a Confusion Matrix where we plot the actual values (ground truth) against the predicted values; each value being either positive (CI-skipped commit) or negative (executed commit).
(see Table 2.1.)

The following are the possible categories of predicted labels.

- True Positives (TP): These are the accurately predicted positive values, meaning that we successfully predicted that the commit should be CI-skipped.
- True Negatives (TN): These are the accurately predicted negative values, meaning that we are choosing to execute a commit that should not be CI-skipped.
- False Positives (FP): These are the wrongly predicted positive values, meaning that we are CI-skipping a commit that should be executed by the CI system.
- False Negatives (FN): These are the wrongly predicted negative values, meaning that we are executing a build that ought to be skipped.

Table 2.1 Confusion Matrix

		Predicted Class	
		Positive	**Negative**
Actual Class	**Positive**	True Positive TP	False Negative FN
	Negative	False Positive FP	True Negative TN

Using these definitions, we can further define the following classification metrics.

$$\text{Accuracy } = \frac{TP+TN}{TP+TN+FP+FN} \in [0,1]$$

$$\text{Recall } = \frac{TP}{TP+FN} \in [0,1]$$

$$\text{Precision } = \frac{TP}{TP+FP} \in [0,1]$$

$$F_\beta = \frac{\left(1+\beta^2\right) \cdot (\text{ precision } \cdot \text{ recall })}{(\beta^2 \cdot \text{ precision } + \text{ recall })}$$

$$AUC = \frac{1+\frac{TP}{TP+FN}-\frac{FP}{FP+TN}}{2} \in [0,1]$$

- **Accuracy** refers to the proportion of correct predictions made by the model. This metric, however, is inappropriate for imbalanced classification because a high accuracy is achievable by a no skill model that only predicts the majority class.

- **Recall** is defined as the proportion of correctly categorized CI skip commits over the total number of commits that are skipped.

- **Precision** is defined as the percentage of detected CI skip commits that are really skipped.
- **F1 score** we can also combine these two metrics into a single one; the $F\beta - score$, which evaluates a model using a single score while taking both the precision and recall into account.
  Using the β parameter, we can further decide if we want to give more weight to either recall or precision. With a β value of 1.0, we are calculating the harmonic mean of precision and recall, giving each the same weighting.
  This measure is called the $F1 - score$ and is the most widely used one in literature.
- **ROC curve** (receiver operating characteristic curve) is a graph showing the performance of a classification model. This curve plots two parameters; the true positive rate or recall (y axis) against the false positive rate (x axis) at all possible classification thresholds. To measure the overall performance using a single scalar value, we proceed to calculate the area under this curve.

For evaluation, two approaches can be adopted, the first one is to train and evaluate the detection model on the same software project's commits history. We are therefore considering the case of an existing project with a rich history whose developers want to optimize by adding the automated CI-skip detection feature.
Another scenario we ought to consider is when a development team is starting a new project. Since, in this case, sufficient historical labeled data is not available to build an AI model, the CI-skip detection tool should therefore be trained on data from other projects that, ideally, use similar tools and technologies. We can take into consideration this scenario by performing a project-cross validation, where, given a set of n projects, we train the studied approach on data from $n-1$ projects and use the remaining one project to test the trained model. This experiment is repeated for all the studied projects

2.3.2 Rule-Based Techniques

Abdalkareem et al. [2] are the first to study CI skipping non-essential commits.
In order to assist developers in automatically detecting and labeling such commits, they presented a rule-based technique to flag the commits to be skipped. The following five rules related to non-source code files and to cosmetic changes were after exploring the dataset described in Section 2.2. If any of them if verified, the commit is skipped.

- **Rule1: non-source code files:** for each commit, the modified files' extensions is checked against a predefined list of non source-code files extensions (.md, .txt, .png, etc.)
- **Rule 2: version release:** release preparation commits are detected by checking the commit only modifies the version in the build configuration files (e.g. Maven or Gradle file for Java projects.
- **Rule 3: source-code comments:** regular expression are used on source-code files to remove comments, the resulting file is then checked to see if it has other differences compared to the previous version.
- **Rule 4: meta files:** Meta files are identified by, again, looking at the files' extensions (e.g. .ignore file).
- **Rule 5: source code reformatting:** compare the current version of the file with the previous version of the file after removing all white spaces that are ignored by the programming language's grammar.

This approach, in addition to requiring a huge amount of time and experience to find suitable rules, only reached a moderate average score of 58% in terms of F1-score on a cross-project evaluation done on a corpus of ten open-source Java projects using Travis CI. Furthermore, this technique is highly prone to false negative.

Similarly, Kawrykow et al. [13] developed a rule-based method for detecting non-essential commits. Documentation updates, simple type adjustments, and rename-induced modifications are among the six rules stated by the authors. On average, 92.8 percent of non-essential modifications were accurately categorized by the suggested approach across seven open source projects.
However, we suspect that this method will not be successful in a CI setting because the suggested rules ignore modifications specific to a CI environment, such as those related to release preparation which account to 15% CI-skiped commits according to Table. 1.1. Consecutively, this method will also result in numerous false negatives.

Furthermore, one of the key drawbacks of rule-based solutions is that they, in general, do not take into account the unique characteristics of a project.
For instance, when it comes to identifying CI skip commits, different projects utilize different types of documentation files, and their development context is also distinct.

2.3.3 Evolutionary Search Approach

To address the difficulties associated with manually analyzing the CI commit history and defining rules, we can propose automating this process using various optimization algorithms. However, due to the exponential number of possible combinations of features and their associated threshold values, this is not a simple task. The fundamental difficulty lying in the large and complex search space to be explored.
As a result, the CI skip commits detection problem can be viewed as a combinatorial optimization problem in which we aim to pick the optimal combination of features and thresholds. And due to its complexity we can, instead of exploring the whole search space, propose the use of meta-heuristic algorithms such as genetic algorithms.

In this context, Saidani et al. [19], inspired by the success of evolutionary algorithms in dealing with search-based software engineering tasks [10], proposed to formulate the problem of CI skip detection as a search-based problem. Using a tree-based representation of an IF-THEN CI-skip rule, they adapted a genetic algorithm to find an optimal detection rules.
Their approach consists in generating a binary tree representation of detection rules. As shown in Fig. 2.2, a detection rule is represented as a binary tree where leaves represent a boolean comparison between a feature and a threshold while upper nodes consist of either the boolean 'AND or the 'OR' operator.

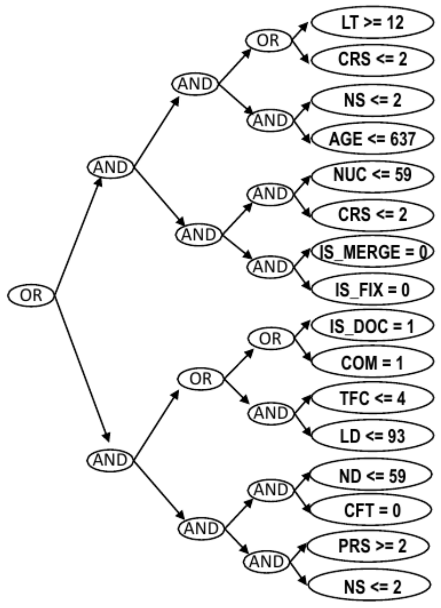


Fig. 2.2 A binary tree representation of a CI skip detection rule

As described in Fig. 2.4, the algorithm starts by generating a random initial population of such rules. Then for each generation, the first step is the fitness assignment step where candidate solutions are compared and ranked according to a quantification of their performance. The final objective is to maximize the coverage of expected CI skipped commits while minimizing non-skipped commits that are incorrectly flagged as skipped.
This is why the following fitness function is used.

$$\text{BestSol } = \min_{i=1}^{n} \sqrt{(1-TPR[i])^2 + FPR[i]^2}$$

Sequentially, only the best solution according to these criteria are selected to breed the next generation, and this is where the binary tree representation comes in handy. First, two parents are selected randomly to breed two new solutions. This is done, during the crossover operation, by randomly selecting a subtree from each parent and exchanging their positions, so that each child consists of a modified copy of the first parent tree having a certain subtree taken from the second parent (Fig.2.3). Secondly, one or more random nodes are randomly selected and both their features and thresholds in the case of a terminal node or the boolean operator in the case of an upper node are replaced by new randomly chosen values in the so-called mutation step (Fig. 2.3).

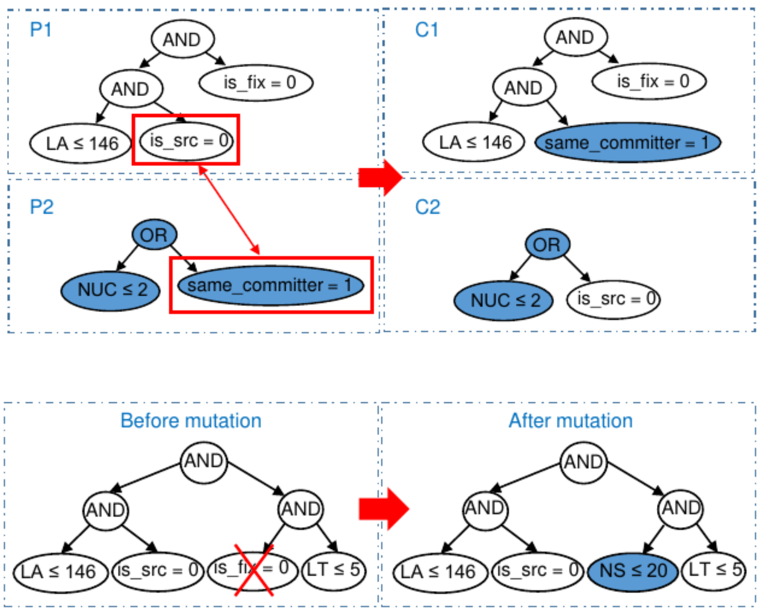


Fig. 2.3 Illustrative examples of the Crossover (top) and Mutation (bottom) operations

This is repeated for all the population in the mating pool, and the resulting population goes on into the next generation to be ranked and to evolve again until a certain number of generations is reached.

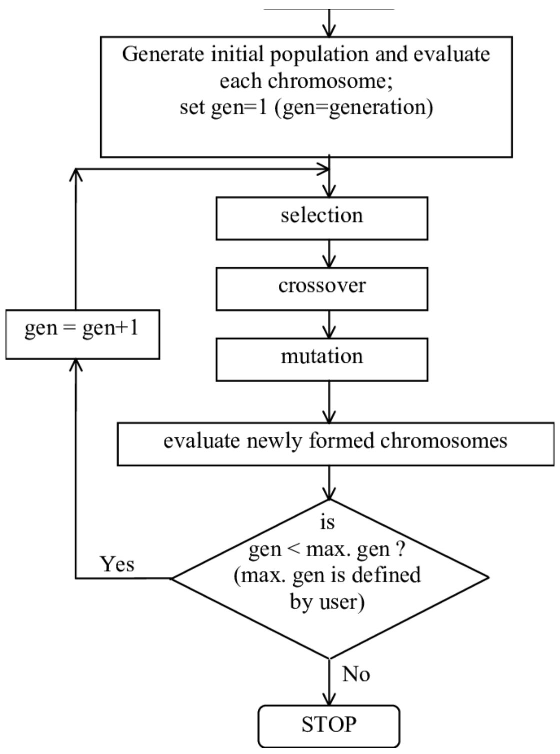

Generate initial population and evaluate each chromosome; set gen=1 (gen=generation)
selection
crossover
gen = gen+1
mutation
evaluate newly formed chromosomes
is gen < max. gen ? (max. gen is defined by user)
Yes
No
STOP

Fig. 2.4 Flow chart of a genetic algorithm [8]

The advantage of this approach, in addition to computation efficiency, is its ability to take into account data imbalance, compared to Machine Learning approaches for instance which may overfit on a particular category of data. And overall, it achieved remarkable results with 92% and 84% in terms of AUC for cross-validation and cross-project validations respectively on a dataset of 15 projects using Travis CI.

Another approach that makes use of a similar genetic algorithm was introduced by Saidani et al. in a research [18] where they were aiming to predict failing builds in a CI context. They proposed to learn the temporal correlation in historical CI build data using Long Short Term Memory (lSTM) to predict the outcome of the next CI build in a given sequence.
However, although LSTM is certainly the best neural network when it comes to exploiting temporal dependencies in past observations, finding the adequate model configuration (hyperparameters) to use may be challenging.

Moreover, the hyperparameters adopted for the model's configuration should also reflect the CI build domain knowledge such as the size of the project, the number of developers, the seasonality of the testing process, etc.
This is why, this research leveraged a genetic algorithm to find the best possible set of hyperparameters to build the model.
A candidate solution, therefore, was considerate as a set of the following parameters:

- number of neurons in each layer
- number of layers
- number of training epochs
- type of optimizer used to update the neural network.
- dropout probability
- number of previous observations used for prediction

The algorithm starts by generating a random population of such solutions, it trains an LSTM-based model using every configuration and then ranks the candidate solution based on the validation loss achieved during evaluation. For mutation, one or more hyperparameters are randomly selected from the set and given new random values and for crossover, randomly selected hyperparameters are exchanged between the two selected parents.
Overall, this approach achieved average results on the dataset it was tested on, with a median F1-score of 49 under a within-project validation and 57 under cross-project validation.
Additionally, we argue here that only considering the outcome of previous builds as a feature is not enough to predict future results and that, hence, considering a larger set of attributes to classify future builds may yield better results.

2.3.4 Machine Learning Approaches

In a related effort to improve CI skip detection, Abdalkareem et al. [1] presented an ML-based solution to address the issue of their initial rule-based approach's moderate performance. They employed a Decision Tree ML model and got a higher F1-score of 79% for within-project validation, however for cross-project validation, the classification performance declined to 55%, which is comparable to the accuracy of random guessing and suggests that this approach cannot be applied to projects with little or no previous commit history.

Nevertheless, they also looked at the most prominent features and discovered that the number of developers who altered the modified files, as well as the terms used in the written commit messages, are the best indicators of CI skip commits.

Besides, many research works have introduced prediction models to predict the build outcome of a commit, this can be reduced to our CI skip detection problem by considering that passing builds should be skipped.
An effort by Xia and Li [28], for instance, to predict the failure of CI builds considered the problem as a binary classification problem, leveraged 9 Machine Learning classifiers and compared their performance under AUC and F1 on both a cross-validation and on-line scenario using commit level features extracted from 126 open-source projects using Travis CI.
From the comparison of these models, the gradient boosting (GRB), logistic regression (LRG), multilayer perceptron (MLP), multinomial Naive Bayes (MNB), the nearest centroid (NRC), nearest neighbors (NRN), random forest (RDF) and ridge regression (RDG), as shown in Fig. 2.5, we conclude that the Random Forest model yields the best results for cross-project validation, whereas the Decision Tree model is the best classifier for the online scenario.

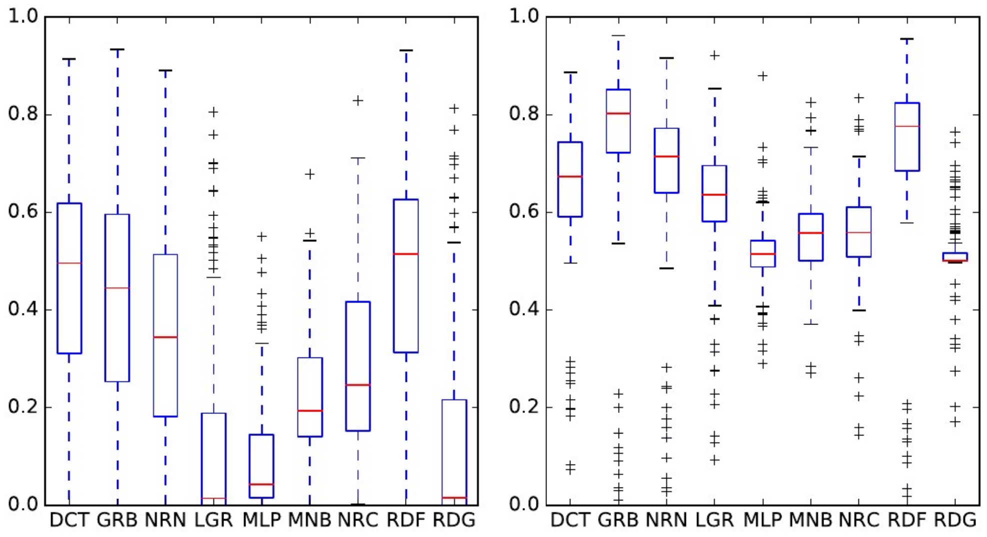


Fig. 2.5 A within-project comparison of ML classifiers on the F1-score (left) and AUC (right)

Luo et al. also employed TravisTorrent dataset attributes to forecast the outcome of a build. [14] They compared Support Vector Machine, Decision Tree, Random Forest, and Logistic Regression. The results of 10-fold cross-validation suggest that LR and SVM were the best performers.

However, during our attempt to replicate the traditional Machine Learning approach, we found out that, in our case, the best performance was achieved by the Random Forest model, shortly followed by the K-nearest Neighbors classifier for both scenarios. This may be explained by the fact that we used a different dataset (more features and fewer projects) for our evaluation and possibly followed different pre-processing steps and opted for different hyperparameters.

Furthermore, an equally important benefit accredited to the use of Random Forest and tree-based models in general is that the classifications they produce can easily be interpreted. Indeed, whereas other ML models such as Support Vector Machines or Multi-Layer Perceptron neural networks produce probability-based black boxes, for tree-based models' classification, we can identify which features are the most important indicator of whether a commit should be skipped or not. This may be done by analyzing the followed branch in the case of a Decision Tree or for a Random Forest, by performing Top Node Analysis for each tree (i.e. the lower is the depth level of a feature in a tree, the more important it is, and results are aggregated for all trees in the forest.)

2.4 Decision Trees

2.4.1 Definition

A Decision Tree is a Supervised Learning method that uses a tree structure to build models for regression and classification tasks. It asks a series of yes or no questions to break down a Dataset into smaller and smaller subsets, then chooses its branches according to the answers.
In a Decision Tree each node is a comparison between a feature and threshold, each branch is a decision made (IF-ELSE rule), and each leaf is a result (discrete or continuous value).
The very top of the tree is called the Root Node, the last nodes are called Leaf Nodes, and the rest of the nodes are called Internal Nodes.

To recall the works mentioned for CI skip detection, the detection rules, whether they were manually designed or found using genetic algorithms, can be seen as a single branch of the decision tree model. Therefore, the key difference is having multiple rules or branches and from this perspective, the essential factor to select which detection rule is applied is the order of the splitting nodes in the tree. Hence, the interpretability of classification remains the same.

2.4.2 Building a Decision Tree

Generally, The construction of a decision tree can be thought of as a greedy algorithm where at each decision node, a locally best attribute is selected to split the data into child nodes. This process is repeated until a leaf node is reached, where further splitting is not possible This procedure is repeated until a leaf node is reached, at which point no more splitting is possible.
We will talk about the two most popular decision tree building algorithms and that are implemented for the building process of the DecisionTree model of SciKit Learn.[1]

Iterative Dichotomizer 3

First, to determine the root node, we must choose the attribute that best classify the Training Data and place it at the top of the tree.
We will then perform a top-down, greedy search through the space of feasible Decision Trees by repeating this approach repeatedly. [20]
The best attribute is the one that yields the highest Information Gain. In order to define the latter precisely, we will begin by defining Entropy, a measure often employed in Information Theory that characterizes the impurity of a collection of samples.
The Entropy $H(S)$ is defined by:

$$H(S) = \sum_{c \in C} -p(c) \log_2 p(c)$$

where:

- S The current set for which the Entropy is being calculated.
- C a set of classes in S, C = skipped commit, executed commit
- $p(c)$ The proportion of the number of elements in class c to the number of elements in set S

[1]Scikit-learn is a key free and open source library for Python that includes mathematical, statistical and general purpose algorithms that form the basis for many machine learning technologies

Consequentially, the Information Gain $IG(A)$ is defined by

$$IG(A,S) = H(S) - \sum_{t \in T} p(t)H(t)$$

Where:

- $H(S)$ The Entropy of set S
- $H(t)$ The Entropy of subset t
- T The subsets created from splitting set S by attribute A such that $S = \bigcup_{t \in T} t$
- $p(c)$ The proportion of the number of elements in t to the number of elements in set S

Now all we have to do is apply these metrics on our Dataset in order to split the data and obtain the appropriate decision tree.

Algorithm 1: Building a Decision Tree Using Entropy

```
Calculate Entropy for the whole dataset
for each tree node do
    for each feature do
        Calculate Entropy for all categorical values
        Take the average Information Entropy for the current feature
        Calculate the Information Gain for the current feature
    end
    Pick the highest Information Gain feature
end
```

CART (Classification And Regression Trees)

In CART, the tree construction algorithm is the same, with the exception of using the Gini Index instead of Information Gain to select the best attribute.
The Gini Index GI for binary target is defined by:

$$GI = 1 - \sum_{t=0}^{t=1} P_t^2 = 1 - P_{t=0}^2 - P_{t=1}^2$$

It gives an indication of how good a split is because of how mixed the classes are in the two groups formed by the split. Therefore, a Gini Rating of 0 indicates an optimal split.

Applications in Imbalanced Classification

The problem with the previously mentioned methods is that, for imbalanced datasets, the splitting criteria (e.g. Information Gain) is skew sensitive which means that, as the prior probability of class is used to calculate a node's impurity degree, the splitting criteria become biased towards the majority class.
Furthermore, if the minority class is closely correlated with multiple features interacting with one another, it will be difficult for a tree to recognize the pattern and even if it does, it tends to become deep and unstable, prone to over-fitting.

Numerous algorithms have been proposed to address this issue, and they can be divided into two groups: the data level and the algorithmic level.
The main idea of the former is to rebalance the distribution of data by applying re-sampling methods such as random undersampling, random oversampling or synthetic minority oversampling (SMOTE). The concern with these methods is that they may raise the risk of overfitting, as they make exact copies of the minority class' samples. This way, a classifier could create rules that appear to be accurate but only apply to some duplicated samples.
The latter group, on the other hand, tries to tweak the original model by applying weights to distinct classes or samples, decreasing the bias induced by sample size differences between classes.
And in the case of decision trees, ensemble learning algorithms are used, such as bagging where each base classifier is trained on a subset of the initial training set, and new samples are usually predicted by voting which is the principle of Random Forests. Another example is boosting, where a base classifier is trained from the initial training set, and then the next base classifier is trained based on the performance of the previous one. The Gradient Boosting Decision Tree model is based on this idea. [27]

2.5 Reinforcement Learning

2.5.1 Reinforcement Learning Framework

Reinforcement learning (RL) is a general framework where an agent is trained to return an optimum solution by taking a sequence of decisions by itself in a given environment.
The agent continuously interacts with its environment and comes up with solutions all on its own, without human interference.
The two primary components are first, the agent, which represents the decision-making

model that is being trained and second, the environment, which represents the outside world that the agent interacts with and is built based on the problem statement.

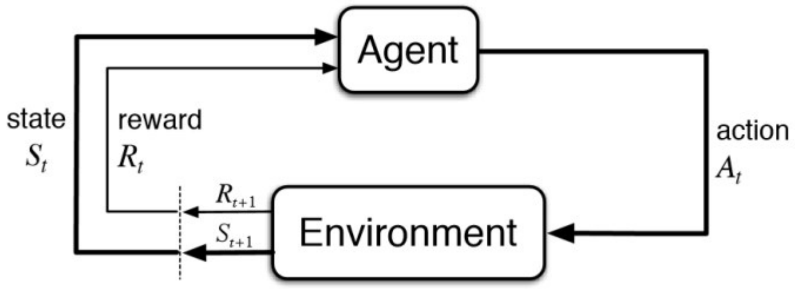


Fig. 2.6 Agent-Environment Interaction in Reinforcement Learning

At each time step t, the agent takes an action A_t on the environment based on its policy $\pi(A_t|S_t)$ and the perceived state of the environment S_t. It must therefore learn how to map the perceived situations to actions. To push it in the right direction, we simply give it a feedback value called the reward, R_t which can be either positive if it performs an action that brings it closer to its goal or negative if it goes away from its goal.
Once an action is selected, a reward is calculated based on the goal of the RL algorithm and the environment transits to the next state S_{t+1} according to the probability $P(S_{t+1} \mid S = S_t, A = A_t)$.
The agent must therefore discover which actions yield the most reward for any given situation by trying them, and hence find and adopt the policy that maximizes the sum of rewards while accounting for subsequent rewards received from the next eventual state if a particular action is taken.
The agent learns automatically by trial and error without being pre-programmed or receiving human intervention. Therefore, it does not require any labeled data, unlike supervised learning.
RL is suitable for solving a specific type of problem where decision-making is sequential, and the goal is long-term.

2.5.2 Formal Definition

Reinforcement Learning can be seen as a framework to solve the certain class of sequential decision-making problems that can be expressed as a Markov Decision Process (MDP).
Thanks to its generality, the MDP formalism can be easily employed for many useful and highly different scenarios and applications, ranging from robotics and machine learning to game theory and economics. It is therefore crucial that we later model the

problem we want to solve using the MDP formalism by defining each of its components. A Markov Decision Process is a tuple (S, A, P, R, γ) where: [23]

- S is a finite set of Markov states, meaning that each state contains all the useful information from the history. $\mathbb{P}[s_{t+1} \mid s_t] = \mathbb{P}[s_{t+1} \mid s_1, \ldots, s_t]$
- A is a finite set of actions
- P is a state transition probability matrix, it returns the probability of moving from state s to a state $s\prime$ when taking the action a. $P^a_{ss'} = \mathbb{P}[s_{t+1} = s' \mid s_t = s, a_t = a]$
- r is a reward function, $r^a_s = \mathbb{E}[r_{t+1} \mid s_t = s, a_t = a]$
- γ is a discount factor $\gamma \in [0, 1]$ which represents the present value of future rewards.
  we can accordingly define the total discounted reward G_t as

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

The ultimate goal of our algorithm is to find the optimal policy (i.e. method used to select an action at a given state.)

$$\pi(a \mid s) = \mathbb{P}[a_t = a \mid s_t = s]$$

In terms of return, a policy π is considered to be better than or the same as a policy π' if the expected return of π is greater than or equal to the expected return of π' for all states.
In other words, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in S$.
Where the value of state s under policy π is the expected return from starting from state s at time t and following policy π thereafter. Mathematically, we define it as

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t \mid s_t = s] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right]. \end{aligned}$$

An important consequence of this formulation is that it implies that a recursive relationship can be defined as follows [3]

$$\begin{aligned}
V_\pi(s) &= \mathbb{E}_\pi\left(r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right) \\
&= \sum_a \pi(a,s) \sum_{s'} p_{ss'}^a \left(r_{ss'}^a + \gamma \cdot \mathbb{E}_\pi\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s'\right)\right) \\
&= \sum_a \pi(a,s) \sum_{s'} p_{ss'}^a \left(r_{ss'}^a + \gamma \cdot V_\pi\left(s'\right)\right)
\end{aligned}$$

In the same vain, the value of action a in state s under policy π is the expected return from starting from state s at time t, taking action a, and following policy π thereafter. Mathematically, we define it as

$$\begin{aligned}
Q_\pi(s,a) &= \mathbb{E}_\pi\left[G_t \mid s_t = s, a_t = a\right] \\
&= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right]
\end{aligned}$$

We are hence interested in finding the optimal state-value function V^* and the optimal action-value Q^*.

$$V^* = max_\pi V_\pi(s) \ \ \forall s \in S$$

$$Q^*(s,a) = max_\pi V_\pi(s,a) \ \ \forall s \in S, a \in A$$

Therefore, V^* is a value function for the optimal policy π^* and it satisfies the following equation known as **the Bellman equation**.

$$\begin{aligned}
V^*(s) &= \max_{a \in A} Q^{\pi^*}(s,a) \\
&= \max_a \mathbb{E}_{\pi^*}\left(r_t \mid s_t = s, a_t = a\right) \\
&= \max_a \mathbb{E}_{\pi^*}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right) \\
&= \max_a \sum_{s'} p_{ss'}^a \left(r_{ss'}^a + \gamma \cdot V^*\left(s'\right)\right)
\end{aligned}$$

Similarly, we can derive the Bellman optimality equation for Q^*,

$$\begin{aligned} Q^*(s,a) &= \mathbb{E}_{\pi^*}\left(r_{t+1} + \gamma V^*(s_{t+1})\right) \\ &= \mathbb{E}_{\pi^*}\left(r_{t+1} + \gamma \max_{a'} Q^*\left(s_{t+1}, a'\right) \mid s_t = s, a_t = a\right) \\ &= \sum_{s'} p_{ss'}^{a}\left(r_{ss'}^{a} + \gamma \max_{a'} Q^*\left(s', a'\right)\right) \end{aligned}$$

This equation provides the optimal Q-function or the maximum return that can be obtained starting from observation s, taking action a and following the optimal policy thereafter.

2.5.3 Q Learning

The basic idea hence is to exploit the Bellman optimality equation in an iterative process, because it can be shown that this process converges Q to the optimal function under the convergence condition of stochastic approximation.

$$Q_i \underset{i \to +\infty}{\longrightarrow} Q^*$$

We can therefore "learn" the Q value using data on the agent's experience following the equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

where $0 < \alpha \leq 1$ is the learning rate, and the larger the learning rate, the stronger is the influence of new data for updating.
The leaning process is as follows.

Algorithm 2: Q-Learning algorithm

```
Set values for learning rate $\alpha$, discount rate $\gamma$, reward function $R$
Initialize $Q(s,a)$ to zeros
for each episode do
    Observe state $s$
    while $s$ is not terminal do
        choose $a$ using $\varepsilon - greedy$ algorithm
        take action $a$
        observe reward $R$ and next state $s'$
        update $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
        set $s \leftarrow$s/
    end
end
```

Besides, to keep track of the states, actions, and their expected rewards across iteration, the traditional approach is to use a Q-table, which is a simple array data-structure that maps a state-action pair to a Q-value that the agent will learn. [25]
Its structure is represented in Table 2.2.

State \ Action	A_1	A_2	$\dots$	A_M
S_1	$Q(S_1,A_1)$	$Q(S_1,A_2)$		$Q(S_1,A_M)$
S_2	$Q(S_2,A_1)$	$Q(S_2,A_2)$		$Q(S_2,A_M)$
$\vdots$			$\ddots$	$\vdots$
S_N	$Q(S_N,A_1)$	$Q(S_N,A_2)$	$\dots$	$Q(S_N,A_M)$

Table 2.2 The structure of a Q-table

At the start of the Q-Learning algorithm, the Q-table is initialized to zeros, indicating that the agent doesn't know anything about its environment yet. As the agent tries out different actions at different states through trial and error, the agent learns each state-action pair's expected reward and updates the Q-table with the new Q-value. Using trial and error to learn about the world is called Exploration, while choosing the best known action at a state s is called Exploitation.
Exploration allows an agent to increase its current understanding of each action, therefore a more accurate action-values estimation. This results, on the long-term, in the agent making more informed decisions.
Exploitation, on the other hand, selects the greedy action that will yield the greatest reward by taking advantage of the agent's existing action-value estimates. However, being greedy with action-value estimations may lead to suboptimal behavior.
When an agent explores, it obtains more precise estimates of action-values. And when it exploits, it might get more reward. However, it has to choose, at each decision, between either, which is also called the exploration-exploitation dilemma.

A common strategy for tackling the exploration-exploitation dilemma is the Epsilon Greedy Exploration Strategy, which lets the agent balance between the two choices by choosing randomly between them.

$$Action \leftarrow \begin{cases} \text{argmax}_a\, Q(a) & \text{with probability } 1-\varepsilon \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$$

2.5.4 Deep Q learning

The major limitation of the table approach to Q-learning is that, as the state space and the set of possible actions grow, this representation quickly becomes unpractical. Indeed, given the fact that the Q-value updates occur in an iterative fashion, the iterative process of computing and updating Q-values for each state-action pair in a large state space becomes computationally inefficient and perhaps infeasible due to the computational resources and time this may take.
Rather than mapping a state-action pair to a q-value, the idea behind Deep Q Learning, is to use a neural network to estimate the Q-values, $Q(s,a;\theta) \approx Q^*(s,a)$.
In a deep Q learning situation, an agent is therefore represented by a neural network that maps input states to (action, Q-value) pairs.

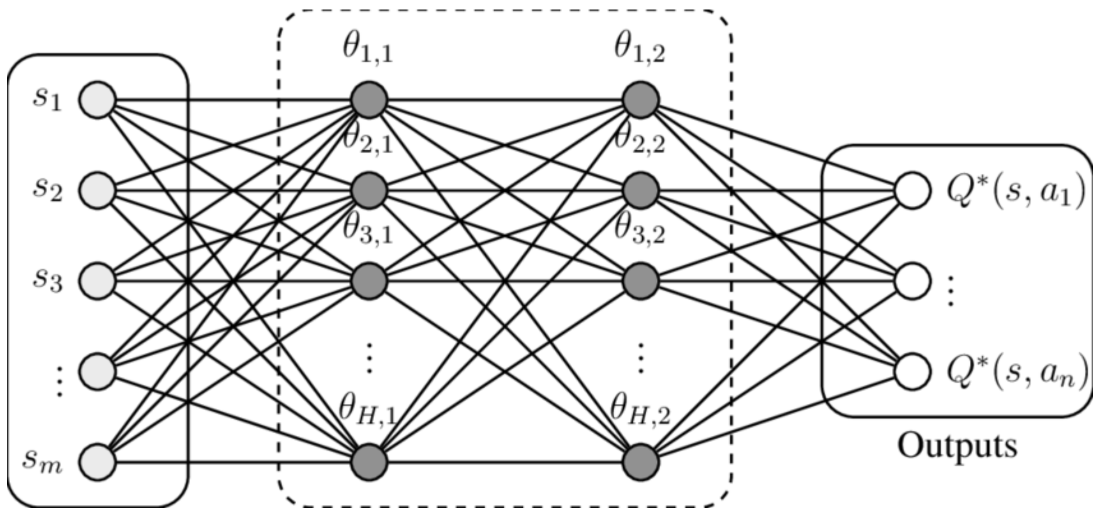


Fig. 2.7 Structure of the neural network used in Deep Q-learning

This can be achieved by minimizing the following loss at each step [24] i:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim \rho(.)}\left[\left(y_i - Q(s,a;\theta_i)\right)^2\right] \text{ where } y_i = r + \gamma \max_{a'} Q\left(s',a';\theta_{i-1}\right)$$

Where:

- y_i is the temporal difference target.
- $y_i - Q$ is the temporal difference error.
- ρ is the distribution over transitions (s,a,r,s') collected from the environment.

Besides, learning can become unstable when trying to find the optimal Q function using non-linear operators like neural networks.

Learning can be stabilized in DQN via a heuristic known as experience replay. It is a technique that saves time-series data in a buffer called replay memory. And, during training, rather than using the latest transition to compute the loss and perform the gradient descent to update the network, we, instead, compute it using a mini-batch of transitions sampled from the replay buffer.
As a result, the correlations between the training samples are minimized, and hence learning is stabilized. Reusing past transition in new updates has also the benefit of learning more from individual transitions multiple times and recalling rare occurrences. The whole process can be summarized in the Algorithm 3.

Algorithm 3: Deep Q-Learning algorithm with Experience Replay

```
Initialize replay memory $D$ to capacity $N$
Initialize the neural network with random weights.
for each episode $t$ do
    Observe state $s_t$
    while $s$ is not terminal do
        choose $a_t$ using $\varepsilon - greedy$ algorithm
        take action $a_t$
        observe reward $r_t$ and next state $s'_t$
        store transition $(s_t, a_t, r_t, s'_t)$ in $D$
        set $s_{t+1} \leftarrow \mathrm{s}_t$
        sample random minibatch of transitions $(s_t, a_t, r_t, s'_t)$ from $D$
        set the target $y_j = \begin{cases} r_j & \text{for terminal } s_{t+1} \\ r_j + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) & \text{for non-terminal } s_{t+1} \end{cases}$
        perform a gradient descent step on $\left(y_j - Q\left(s_t, a_j; \theta\right)\right)^2$
    end
end
```

2.5.5 Use of Reinforcement Learning in Classification

Deep reinforcement learning has attracted much attention from researchers to address sequential decision-making problems in various tasks, including software engineering. Yet, for classification tasks, RL is not as popular. During our literature review, we only encountered few studies that leverage RL in such contexts and none that addressed the CI-skip detection problem.

2.6 Conclusion

To summarize, the CI-skip automation task has been tackled by several researchers and our review of the literature shows that the most common approach involve defining detection rules or applying models that resolve around the use of such rules, such as tree-based classifiers. These methods are generally preferred as they provide the developer with a comprehensible justification for the CI skip decision made, especially when the changes made in the commit are non-trivial.

Moreover, the related works that we reviewed during this chapter helped us better understand our task and inspired us to propose our own approach, where we try to preserve the benefits provided by these works while addressing the performance issues they suffer from.

In the next chapter, we propose a novel approach to tackle our problem. We start by explaining how the ideas introduced in this chapter were adopted to fit our task, we then explain how our approach works and finally we perform an evaluation where we compare our solution to other state-of-the-art techniques.

Chapter 3

Proposed Solution

3.1 Introduction

To recall, our goal is to design a novel solution to detect whether a commit should be skipped from the CI process or if it should trigger a building and testing phase. We model this problem as a binary classification task and aim to develop a solution that would perform as well as existing techniques or outperform them while providing interpretable classification. This solution should also successfully address the major concern about this task, which is the imbalanced nature of the data, as today most commits in open source projects are not CI skipped.

Therefore, in this chapter, we start by explaining the intuition behind our proposed solution, then describe how it works based on the theoretical details explained in the previous chapter. Finally, we perform an evaluation of this approach by comparing it to state-of-the-art techniques.

3.2 Motivation

After reviewing existing works related to CI in the previous work, we noticed that most works agree on using tree-based algorithms to find the optimal classification rules thanks to their good performance and interpretability.
We also argue that, from a developer's point of view, detailed information about what the AI model's decision is based on is needed when prompt to CI-skip a commit. Because, otherwise, a developer may choose to execute the commit regardless of the AI tool's recommendation for a sense of security.
This is why, we chose to address the CI-skip problem using the decision tree model,

which in addition to its promising performance when benchmarked against other ML models, is also easily interpretable by nature. Indeed, a DT's decision, for a given sample, can be explained by recording the nodes that were traversed to choose an outcome.

However, as seen in section 2.4.2, the decision tree model is very prone to overfitting and performs poorly on imbalanced datasets as it is biased towards the majority class. To address this problem, researchers experimented with applying resampling techniques, which do not perform so well on the CI-skip detection context [1] as they tend to increase the likelihood of the model overfitting on replicated or synthesized data. Another approach they tried is to improve the model's performance by applying Ensemble Learning techniques to boost the models' performance, such as using Random Forest or Gradient Boosted Trees for tree-based models.
We, instead, propose to solve this issue, using a single decision tree, by addressing its building process. We propose therefore an alternative Decision Tree building algorithm using Deep Reinforcement Learning based on the work of Wen and Wu [26],

3.3 Solution Details

This decision tree building process is regarded as a multi-step game that can be modeled as a Markov decision process. We therefore start by explaining what each component of the MDP is in our case, and then describe how the training process occurs.

3.3.1 Principle

We start by randomly generating a decision tree (i.e. features and thresholds are chosen randomly among possible values). Then, for each node, the agent observes the state of the tree and chooses to replace the node (both the feature and the thresholds) by a newly generated one so that, the evaluation metrics of the tree when applied to classify the dataset are improved.
To do so, we employ Deep Q Learning using two neural networks. The first one is used to choose what feature to map to the selected node, and the second one chooses which threshold to select for that feature, given the position of the selected node and the state of the tree.

3.3.2 State of The Environment

The state s in time step t needs to be fed to each neural network. It also has to describe the state of the whole tree, so it must contain information about both the attribute and the threshold of all nodes in the tree. However, when more complex models as needed (i.e. larger trees with a longer depth d), the number of nodes N grows exponentially following the formula $N = 2^d - 1$.
Therefore, we implement tree-based convolution in order to reduce the size of the state representation.
Each tree node comparing an attribute k to a threshold x_k is represented by a one hot vector containing the threshold at the index of the given attribute:

$$[0, \dots, 0, x_k, 0, \dots, 0]$$

Hence, at each convolution step, we average the vector representations of each node with its left and right child's therefore, reducing the depth of the tree by one layer.
Then, like convolution in image processing, a one-way pooling operation is performed on the convolution's output to pool all features to a 1-dimensional vector that can be fed to the neural networks.
Finally, a component is added to the vector to indicate which node is being modified at each step. Its value represents the number of the node obtained by following a breadth first traversal of the tree structure.

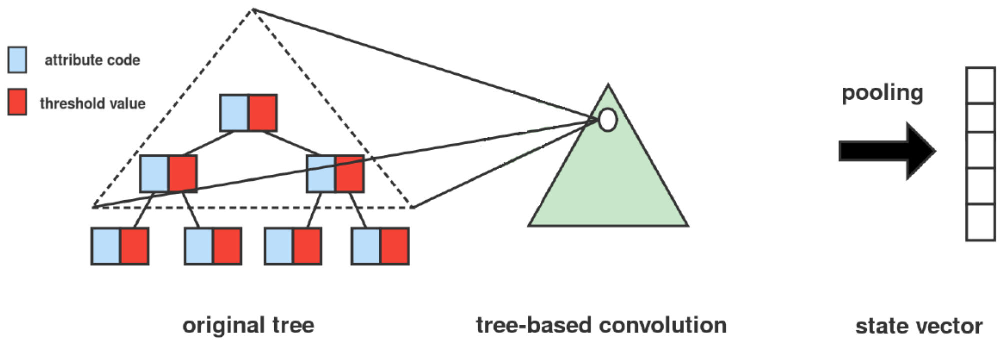


Fig. 3.1 Schematic representation of the transformation process from decision tree to state vector

3.3.3 The Agent's Actions

At each episode m of the game, we build and evaluate a new decision tree by taking a set of discrete actions A. Each action a_t in the set corresponds to the action taken for the node t. It is therefore composed of the attribute k and its threshold, x_k.

$$A = \bigcup_{0 \leq t < N} \{a_t = (k, x_k)\}$$

Here, to address the exploration-exploitation tradeoff, an $\varepsilon - greedy$ policy is employed to choose a random action with probability ε and with probability $1 - \varepsilon$ it selects the action with the highest expected return.

3.3.4 The Reward

After each action, we apply the new decision tree to classify the training set at step t. Then, the predicted results $\hat{Y}_t$ and the ground truth Y_t are used to calculate the $F1 - score$ sc_t.
Next, the reward r_t is obtained according to the following equation.

$$r_t = sc_t - sc_{t-1}$$

Hence, a positive reward r_t means that the action improves the performance of the classification model, whereas a negative reward r_t decreases it. This way, during training, the agent is encouraged to take actions that improve the classification performance, which is measured in a way that accounts for the imbalanced nature of the data.

3.3.5 The Training Process

We first start by building a complete binary tree T_0 with N nodes. To do so, we set a random attribute and a random threshold value for each tree node.
At the same time, the parameters of thresholds-network θ_x and the attributes-network θ_q are initialized.
Now, at each episode m, we build and evaluate a new Decision Tree. Each step t corresponds to modifying the node number t, $0 < t \leq m$
We start by generating the state S_t of the tree T_t at time step t using tree-based convolution.
The agent, consisting of two neural networks, observes the state of its environment s_t and takes an action containing a discrete attribute k and a continuous threshold value x_k.

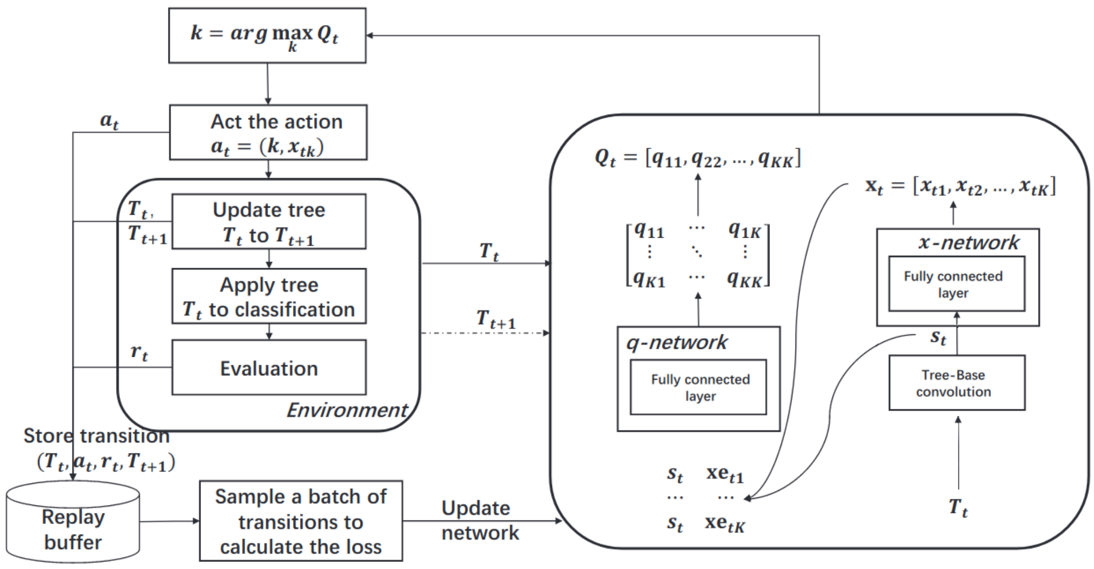

$k = arg \max_k Q_t$
Act the action $a_t = (k, x_{tk})$
a_t
T_t, T_{t+1}
Update tree T_t to T_{t+1}
Apply tree T_t to classification
r_t
Evaluation
Environment
T_t
T_{t+1}
Store transition (T_t, a_t, r_t, T_{t+1})
Replay buffer
Sample a batch of transitions to calculate the loss
Update network
$Q_t = [q_{11}, q_{22}, \dots, q_{KK}]$
$x_t = [x_{t1}, x_{t2}, \dots, x_{tK}]$
x-network
Fully connected layer
q-network
Fully connected layer
s_t
Tree-Base convolution
s_t xe_{t1}
s_t xe_{tK}
T_t

Fig. 3.2 The Framework of the proposed solution

To do so, the thresholds-network $\mathbb{Q}_x(s_t;\theta_x)$ computes a threshold vector X_k containing threshold values for all the sample's attributes, given the tree state.

$$X_t = \mathbb{Q}_x(s_t;\theta_x)$$

Then, given the tree state s_t and the thresholds vector X_t, the attributes-network $\mathbb{Q}_q(s_t, X_t;\theta_q)$ calculates the new Q-values based on the iterative calculation of the Bellman optimality. (section 2.5.3)

$$[q_1, q_2, \dots, q_K] = \mathbb{Q}_q(s_t, X_t;\theta_q)$$

The attribute with the highest Q-value is therefore chosen, and the action a_t is taken.

$$a_t = \begin{cases} \text{random sample} & \text{probability } \varepsilon \\ (k_t, x_{tk}) \text{ such that } k_t = \arg\max_{k\in[K]} \mathbb{Q}_\text{q}\left(s_t, \text{xe}_{tk};\theta_\text{q}\right) & \text{probability } 1-\varepsilon \end{cases}$$

where xe_{tk} is a vector where the k^{th} dimension is equal to x_{tk}, and everything else is zero.
Next, the environment updates the attribute and threshold value of the t^{th} node of the tree T_t into k and x_k so, we transition from T_t to T_{t+1}.
The reward r_t will then be calculated based on the classification result.
We also apply an experience replay mechanism to reduce temporal correlations and increase rare experience efficiency during the update.

As a result, the agent must save the transition (s_t, a_t, r_t, s_{t+1}) in the replay buffer and then sample a random batch of transitions B to update the neural networks.
The target value is therefore,

$$y_b = \begin{cases} r_b & \text{if } T_{b+1} \text{ is terminal} \\ r_b + \gamma \max_{k \in [K]} \mathbb{Q}_\mathrm{q}\left(s_{b+1}, \mathrm{xe}_{b+1,k}; \theta_\mathrm{q}\right) & \text{otherwise} \end{cases}$$

And we finally perform a gradient descent step on the loss functions below to update the two neural networks:

$$L_\mathrm{q}\left(\theta_\mathrm{q}\right) = \mathbb{E}\left(y_b - \mathbb{Q}_\mathrm{q}\left(s_b, \mathrm{xe}_{bk}; \theta_\mathrm{q}\right)\right)^2$$

$$L_x\left(\theta_\mathrm{x}\right) = \mathbb{E}\left(-\sum_{k=1}^{K} \mathbb{Q}_\mathrm{q}\left(s_b, \mathrm{xe}_{bk}; \theta_\mathrm{q}\right)\right)$$

However, if the action taken at a node t reduces the classification score, we will not make the networks learn from it, but instead we move on directly to the next episode. The agent will therefore rebuild the tree using the optimal node values known so far up to that particular node, and try different values in order to improve the score.
The full algorithm can be seen in Appendix B.

3.4 Experiments and Evaluation

3.4.1 Evaluation Methodology

Considered Dataset

To evaluate this novel approach, we consider the dataset introduced in section 2.2 which contains data from open-source Java projects using Travis CI as a CI system and belonging to different fields. For each commit, we have 29 attributes belonging to the three categories below:

- **Statistics about the current commit:** such as features describing the number of lines or comments added, the number and type of files modified, the distribution of the changes across the files or the importance of terms appearing in the commit message.
- **Commit purpose:** these are mainly boolean attributes that describe the goal of the commit; whether it is a formatting commit, a documentation commit, a source code commit and so on.

- **Link to last commit:** contains information about the status of the previous commits and about the developers' experience.

The description of all the 29 considered classification features and the rationale for choosing them is presented in Appendix A.

Validation Approach

Moreover, we conduct the evaluation in two steps.
First, we perform a **within-project validation** where for each project, our model is trained and evaluated on two different chunks of its commit history.
Second, we employ **cross-validation** by iteratively training the model on the merged data of all available projects except one, which will be used to test the performance. This way, we account for the possibility of deploying this solution on new projects where no commit history is available to train on and therefore the model needs to be trained on data from other sources.
Moreover, at each step, we record the area under the ROC curve (AUC) and the F1-score classification metrics.

To put into perspective this new method, we compare it to different state-of-the-art techniques from literature that were evaluated on different datasets with unknown pre-processing steps. Therefore, we select the most recent and performant works on CI-skip detection and evaluate them the same way as our solution by either replicating the work or using provided code.
The following are the detection methods we compare our solution to:

- GAR: Decision rules obtained using a genetic algorithm. [19]
- DT: Decision Tree built using the Gini Index.
- RF: Random Forest classifier, which was shown to be the ML model that yields the highest classification score for our task.

3.4.2 Results and Comparison

Table 3.1 presents a comparison between our approach and several other techniques under a within-project evaluation, while Table 3.2 reports the same comparison but under cross-project evaluation.

Within-project Evaluation

Altogether, the technique we adopted achieves reasonably good results, with an average accuracy of 91%. However, due to the imbalanced nature of our dataset, we are most interested in the following metrics. The f1-score where our model achieved a score of 60% and the AUC where we had a performance measure of 0.7 under a within-project evaluation.

Overall, we also observe that our method yields better results when compared with the two other decision tree building algorithms with an average improvement of 0.17% in terms of F1-score, and is only outperformed once by the Random Forest model on the pghero project data.
These two models, despite having a high accuracy rate on the steve project, yield remarkably low F1-score and AUC. This particular project, having only 32% of its commits labeled as CI-skipped, illustrates how the traditional ML approaches struggle with imbalanced datasets while our approach, by rewarding actions that improve the F1-score, results in a considerably better balance between the detection of the minority class (skipped commit) and the majority class (executed build), and hence a better overall performance.

Table 3.1 Comparison between our approach and state-of-the-art techniques under within-project validation

Score / Project	F1-score				AUC			
	Ours	GAR	DT	RF	ours	GAR	DT	RF
candybar-library	0.79	1	0.75	0.72	0.71	1	0.49	0.47
pghero	0.6	0.85	0.58	0.77	0.71	0.92	0.72	0.85
mtsar	0.71	0.88	0.51	0.55	0.9	0.91	0.63	0.66
steve	0.36	0.62	0.28	0.21	0.6	0.82	0.61	0.56
SemanticMediaWiki	0.49	0.45	0.24	0.04	0.65	0.69	0.54	0.5

Cross-Project Evaluation

However, for cross-project evaluation, the performance is, as expected, a bit lower. This is explained by the fact that under cross-project validation, the target project may have a low collinearity with the features thresholds of other projects as they may have, for instance, different numbers of developers with varying experiences and habits and may use different technologies.
Nevertheless, we achieved relatively high AUC scores with an average of 66% while the values range from 0.59 to 0.76%.

This indicates that our approach can effectively be adopted as a CI skip detection tool within a software development context and may be used for new software projects not having enough history to train ML models.

Table 3.2 Comparison between our approach and state-of-the-art techniques under cross-project validation

Score / Project	F1-score				AUC			
	Ours	GAR	DT	RF	Ours	GAR	DT	RF
candybar-library	0.62	0.92	0.48	0.53	0.7	0.86	0.47	0.61
pghero	0.6	0.8	0.44	0.47	0.76	0.68	0.64	0.65
mtsar	0.54	0.68	0.41	0.37	0.59	0.72	0.58	0.59
steve	0.44	0.52	0.2	0.16	0.68	0.75	0.57	0.54
SemanticMediaWiki	0.33	0.41	0.27	0.19	0.59	0.64	0.54	0.53

Results Interpretation

Concerning the genetic algorithms approach, the results in the table are directly taken from the research paper. However, in order to get more insight into their classification, we did replicate the algorithm as described in the paper. However, we were not able to find the same metrics, undoubtedly because hyperparameters were not shared in the paper. However, thanks to this, we realized that the GA algorithm has one advantage over our method, which may explain the performance difference in some projects. The advantage is the fact that a genetic algorithm may randomly construct and evolve a population of rules with different sizes whereas, for our solution, the tree depth is a hyperparameter that should be selected manually before launching the algorithm.
This is particularly apparent for the ContextLogger project, a java framework for providing contextual human-readable information in log files. Its data can be correctly classified using a simple two comparisons rule: The commit being a documentation commit and the extension of the modified files being Markdown. And, in this case, a larger tree will yield a lower performance.
However, this is a particular case, as under cross-validation, such simple rules generally fail to generalize to other projects and hence perform poorly.

Nevertheless, this suggests that in a future work, we may consider enhancing our model by exploring techniques to find the optimal hyperparameters for both the Decision Tree model (tree depth) and the Neural Networks (number of layers and nodes per layer, optimizer, etc.) because choosing such hyperparameters without prior knowledge about the project is not trivial and often requires significant trial and error efforts.

Moreover, in order to interpret the classifications performed by our model, we conduct a feature importance analysis where we calculate the importance of an attribute as the decrease in node impurity weighted by the probability of reaching the node. The node probability is calculated by the number of samples that reach the node, divided by the total number of samples. The higher this value, the more important the feature. We then aggregate the importance values for the different trees constructed under the cross-project evaluation.
This analysis reveals that the commit message is the most influential feature to decide whether to skip a commit or not, shortly followed by the number of affected files attribute and the *is_source* attribute which indicates whether the commit affects source files.
An interesting observation to make here is that statistics about the current commit are not present in the top-features list, which indicates that these features are less likely to be generalized as they depend on the specific context of the project.

3.5 Conclusion

In this chapter, we employed a novel decision tree building technique based on deep reinforcement learning to build a decision tree classifier for the automatic detection of Continuous Integration commits that should be skipped. We went through the details of this solution and its evaluation methodology, and compared it with state-of-the-art techniques.

Overall, our evaluation indicates competitive performance and the nature of our solution ensures quick and interpretable predictions when deployed in a CI context. This way, developers using our solution will have a justification indicating why they should skip testing the modifications they made. And arguably giving the choice to developers to CI skip a commit while providing a justification would be better than automatically skipping detected commits while risking skipping failing builds (false positive) and especially in more complex scenarios where the developer's expertise is essential. Besides, the developer's feedback can also be utilized to further improve the classifier's performance, as it can be considered to label more data.

Moreover, it is common for software projects to regularly shift their testing priorities as more tests are added, removed or modified. This usually happen when developers are implementing a new feature in an established project and are therefore focused, for a set period of time, on modifying a subset of files and only interested in the result of a few tests. In this situation, our solution provides the benefit of being adaptable in a sense that once more labeled data is collected, the agent can retain its current knowledge while learning from the new information. Therefore, it will be able to modify the decision tree classifier in order to account for priority shift without having to be re-trained on the whole dataset.

Conclusion and Discussion

Continuous integration, despite helping lower the risk and cost of delivering defective software builds, is often seen as an expensive practice because of the computational time required to frequently build and test the project after every modification.
In order to reduce the overall cost and duration of running the build process while preserving Continuous Integration's ability to detect failures, our objective within this project was to automatically detect whether a new commit should trigger the build and test process or if it may be CI skipped.

To address this task, we proposed to employ the Decision Tree Machine Learning model to classify commits based on features concerning the nature and size of modifications, as well as their purpose and temporal context. This choice was made to ensure an easy interpretability of the CI skip decision, which is essential from a developer's point of view. And the features can be extracted using the existing "CommitGuru" tool.

However, decision trees are not suitable for imbalanced classification tasks. And the data available to build a CI skip detection tool is imbalanced by nature, as a minority of commits are skipped while most commits are built and tested because the CI skip option is often unknown and underused by developers.
Therefore, to account for data imbalance, we used a new technique to build this model based on Deep Reinforcement Learning where an agent is tasked to iteratively adjust tree nodes in order to improve the detection metrics.

The effectiveness of this approach was verified by conducting both within and cross-project validation and comparing it to best existing methods. This evaluation showed that our approach outperforms the baseline decision tree building algorithms and other Machine Learning models in terms of F1-score and AUC, which are the most common classification metrics for imbalanced datasets.

Moreover, the major benefit of our solution consists in its ability to provide the user with a comprehensible justification for its choice to CI skip complex commits. Furthermore, once more labeled data is available, the model can retain its knowledge and further adjust its policy to account for a shift in the project's development and testing priorities as more functionalities are added, for example.

As a future enhancement of our technique, we consider investigating possible methods to automatically select the best hyperparameters for the classification model and especially the tree depth, which by controlling the size of the detection rules is directly related to performance of the model. Choosing this parameter is not a trivial task, as different projects require detection rules of different complexities, and therefore we need to be able to automatically choose an optimal parameter for a given project.

In addition, the decision tree construction process can be adjusted to fit other data types and objectives simply by modifying the reward function. Therefore, this technique can be exploited in other classification and detections contexts. and may be utilized to address different classification and detection tasks. Furthermore, this technique can be considered in other engineering areas apart from the software development field and can even be adapted to account for regression tasks.

Besides, as mentioned earlier, this is one possible way to optimize the Continuous Integration process and other strategies can be considered. We can, for instance, propose to optimize this process by developing an intelligent solution that would select a smaller subset of the most promising test cases to detect bugs.
A related idea would be to predict build times within a CI context so that project managers can determine the best build setup to keep the build wait time to an acceptable level and developers can better plan and manage their time and tasks.
Finally, another problem with continuous integration and software testing in general is flaky tests: tests that fails to produce the same result each time the same analysis is run due to their non-determinist nature. Such tests make CI results irrelevant, and therefore a possible optimization approach would be to automatically detect if a test failure is due to a flaky test without rerunning the build.

Finally, this work represents an interesting case study on the usefulness of Deep Reinforcement Learning in the software engineering field, and its remarkable performance should promote its use in a wider range of applications.

References

[1] Abdalkareem, R., Mujahid, S., and Shihab, E. (2020). A machine learning approach to improve the detection of ci skip commits. *IEEE Transactions on Software Engineering*, PP.

[2] Abdalkareem, R., Mujahid, S., Shihab, E., and Rilling, J. (2021). Which commits can be ci skipped? *IEEE Transactions on Software Engineering*, 47(3):448–463.

[3] Alexander Mathis, A. D. (2015). Lecture 8 – Reinforcement Learning III: Bellman equation.

[4] Anastasov, M. (2020). Continuous integration explained. https://semaphoreci.com/continuous-integration. [online; accessed 2022-04-12].

[5] Assaraf, A. (2018). Travis ci vs circleci. https://coralogix.com/blog/travis-ci-vs-circleci/. [online; accessed 2022-05-17].

[6] Beller, M., Gousios, G., and Zaidman, A. (2017a). Oops, my tests broke the build: An explorative analysis of travis ci with github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 356–367.

[7] Beller, M., Gousios, G., and Zaidman, A. (2017b). Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 447–450.

[8] Benhachmi, M., Ouazar, D., Naji, A., Cheng, A., and el Harrouni, K. (2001). Optimal management in saltwater-intruded coastal aquifers by simple genetic algorithm.

[9] Gousios, G. (2013). The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA. IEEE Press.

[10] Harman, M. and Jones, B. (2001). Search-based software engineering. *Information and Software Technology*, 43.

[11] Hilton, M., Tunnell, T., Huang, K., Marinov, D., and Dig, D. (2016). Usage, costs, and benefits of continuous integration in open-source projects. In Khurshid, S., Lo, D., and Apel, S., editors, *ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pages 426–437. Association for Computing Machinery, Inc. Funding Information: This

work was partially funded through the NSF CCF-1421503, CCF-1439957, and CCF-1553741 grants.; 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016 ; Conference date: 03-09-2016 Through 07-09-2016.

[12] Kainulainen, P. (2014). The cost of context switching. https://www.petrikainulainen.net/software-development/processes/the-cost-of-context-switching. [online; accessed 2022-05-24].

[13] Kawrykow, D. and Robillard, M. P. (2011). Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 351–360, New York, NY, USA. Association for Computing Machinery.

[14] Luo, Y., Zhao, Y., Ma, W., and Chen, L. (2017). What are the factors impacting build breakage? *2017 14th Web Information Systems and Applications Conference (WISA)*, pages 139–142.

[15] M. Fowler (2006). Practices of continuous integration. https://www.martinfowler.com/articles/continuousIntegration.html. [online; accessed 2022-04-11].

[16] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig (2016). Usage, costs, and benefits of continuous integration in open-source projects. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.

[17] Rosen, C., Grawi, B., and Shihab, E. (2015). Commit guru: Analytics and risk prediction of software commits. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 966–969, New York, NY, USA. ACM.

[18] Saidani, I., Ouni, A., and Mkaouer, M. W. (2022). Improving the prediction of continuous integration build failures using deep learning. *Automated Software Engineering*, 29.

[19] Saidani, I., Ouni, A., and Mkaouer, W. (2021). Detecting skipped commits in continuous integration using multi-objective evolutionary search. *IEEE Transactions on Software Engineering*, pages 1–1.

[20] Sanjeevi, M. (2017). Chapter 4: Decision trees algorithms. https://medium.com/deep-math-machine-learning-ai/chapter-4-decision-trees-algorithms-b93975f7a1f1. [online; accessed 2022-06-01].

[21] Sharma, G. (2021). How artificial intelligence and machine learning are revolutionizing software development. https://www.computer.org/publications/tech-news/trends/ai-is-changing-software-development. [online; accessed 2022-06-08].

[22] Shi, A., Zhao, P., and Marinov, D. (2019). Understanding and improving regression test selection in continuous integration. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 228–238.

[23] Silver, D. (2015). Lectures on reinforcement learning. URL: https://www.davidsilver.uk/teaching/.

[24] Tensorflow.org (2021). Introduction to rl and deep q networks. https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/. [online; accessed 2022-05-30].

[25] Wang, M. (2020). Deep q-learning tutorial: mindqn. https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc/. [online; accessed 2022-05-30].

[26] Wen, G. and Wu, K. (2021). Building decision tree for imbalanced classification via deep reinforcement learning. In Balasubramanian, V. N. and Tsang, I., editors, *Proceedings of The 13th Asian Conference on Machine Learning*, volume 157 of *Proceedings of Machine Learning Research*, pages 1645–1659. PMLR.

[27] Wen, G. and Wu, K. (2022). Building decision forest via deep reinforcement learning.

[28] Xia, J. and Li, Y. (2017). Could we predict the result of a continuous integration build? an empirical study. *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 311–315.

Appendix A

Features Used for Classification

Table A.1 Features Used for Classification

Feature	Rational
Number of changed sub-systems	Commits that affect several subsystems should usually not be CI skipped.
Number of changed directories	Commits that affect many directories usually should not be CI skipped.
Number of changed files	Usually non-trivial changes affect many files.
Entropy, calculated as the Measure of the distribution of the change across files.	High entropy suggests complex changes.
Number of added lines.	Adding new lines indicates that the code's functionality has changed and should therefore be tested.
Number of deleted lines	Many deleted lines indicate that the change should be tested.
Day of the week when the commit was performed	The routine of developers and can manifest itself in CI skip activity.
The commit message (TF-iDF embeddings)	Commit message may contain useful information about the change.
Type of affected files identified by their extension.	The type of files illustrates the type of modifications.

Commit category	7 commit types are considered: Addition, Corrective, Merge, Perfective, Preventative, Non-Functional, None. Addition commits for example are usually not skipped.
Whether the commit is a bug fixing commit	Fixing bugs means that the code need to be tested.
Whether the commit is a documentation commit,	Commits that only modifies documentation files are likely to be skipped.
Whether the commits contains only build files	If the commit is mainly related to release version, it can be skipped.
Whether the commit a meta commit	Commits that only change meta-files can be skipped usually.
Whether the commit is a merge commit	If the commit is a merging one, it is likely to be CI skipped.
Whether the commit only contains media files	Media-related changes can be CI-skipped.
Whether the commit a source commit	Commits that contain only source files may need to be tested.
Whether the commit a formatting commit	These commits are likely to be CI skipped.
If the commit changes only comments in the source code	These changes should be skipped.
Whether the commit a maintenance commit	Performing maintenance usually requires testing the modifications.
Number of recently skipped commits	Can impact the likelihood of skipping the current commit.
Number of commits recently skipped by the current committer	The current commit may be part of the developer's current task.
Previous commit status	There can be a correlation between the outcome of the previous and the current commit.

Time since these files were lastly modified	Faster modifications may introduce more bugs, necessitating additional testing.
Number of Developers	The risk of failure increases when many developers modify the same file.
The number of commits made by the developer	Developers with smaller experience tend to make simpler commits specially in open-source projects.
Recent experience of the developer	The developer's experience in terms of number of commits weighted by their age.
Number of commits made by the developer to the subject subsystem	Indicate how familiar the developer is with the changed files.

Appendix B

Algorithm of The Proposed Solution

Table B.1 Notation Used in the Algorithm

Notation	Explanation
t	A time step
K	The number of attributes
T_t	The decision tree T at time step t
s_t	A state s at time step t
r_t	A reward r at time step t
X_t	The threshold vector at time step t
x_{tk}	The threshold value corresponds with k_{th} attribute at time step t
xe_{tk}	It is a vector where the k^{th} dimension is equal to x_{tk}, but everything else is zero
$\mathbb{Q}_x$	The thresholds-network
$\mathbb{Q}_q$	The attributes-network
θ_x	The parameters correspond with thresholds-network
θ_q	The parameters correspond with attributes-network

Algorithm 4: Building an Optimal Decision Tree Using Deep Q Learning

```
requirements dataset $D$, max nodes number $N$, max training episode $M$,
  exploration parameter $\varepsilon$, minibatch size $B$, replay buffer $L$, discount rate $\gamma$.
for $m = 0$ to $M-1$ do
    Initialize a full binary tree $T_0$
    for $t = 0$ to $N-1$ do
        Compute the state vetor $s_t$ using tree-based convolution
        Compute the threshold vector: $X_t \leftarrow \mathbb{Q}_x(s_t, \theta_x)$
        Choose an action
          $a_t = \begin{cases} \text{random sample} & \text{probability } \varepsilon \\ (k_t, x_{tk}) \text{ such that } k_t = \arg\max_{k\in[K]} \mathbb{Q}_\text{q}\left(s_t, \text{xe}_{tk}; \theta_\text{q}\right) & \text{probability } 1-\varepsilon \end{cases}$
          where $xe_{tk} = (0, 0, \ldots, x_{tk}, 0, \ldots, 0)$
        Take action $a_t$ , change $T_t$ into $T_{t+1}$
        Apply $T_{t+1}$ to classify on $D$ and observe reward $r_t$
        Store transition $(T_t, a_t, r_t, T_{t+1})$ into $L$
        Sample $B$ transitions $(T_b, a_b, r_b, T_{b+1})_{b\in[B]}$ from $L$ randomly
        Set the target
          $y_b = \begin{cases} r_b & \text{if } T_{b+1} \text{ is the terminal} \\ r_b + \gamma \max_{k\in[K]} \mathbb{Q}_\text{q}\left(s_{b+1}, \text{xe}_{b+1,k}; \theta_\text{q}\right) & \text{otherwise} \end{cases}$
          where $s_{b+1}, \text{x}_{b+1} = \mathbb{Q}_\text{x}\left(T_{b+1}, \theta_\text{x}\right), \text{xe}_{b+1,k} \in \text{x}_{b+1}$
        Perform a gradient descent step on $L_q(\theta_q)$ and $L_x(\theta_x)$
          $L_\text{q}\left(\theta_\text{q}\right) = \mathbb{E}\left(y_b - \mathbb{Q}_\text{q}\left(s_b, \text{xe}_{bk}; \theta_\text{q}\right)\right)^2$
          $L_x\left(\theta_\text{x}\right) = \mathbb{E}\left(-\sum_{k=1}^{K} \mathbb{Q}_\text{q}\left(s_b, \text{xe}_{bk}; \theta_\text{q}\right)\right)$
        if $r_t < 0$ then
            Break
        end
    end
end
```