Testing Faster in Continuous Integration Pipelines. An Industrial Case Study

Diego Joshua Martínez Pineda
diem@itu.dk

Supervisor: Mahsa Varshosaz
Submitted: May 2022
STADS code: KISPECI1SE

IT UNIVERSITY OF COPENHAGEN

SYN/ACK[1]

Oh... so many things to acknowledge and be thankful about. It transcends words really.
Thanks to the kind people and great engineers at Microsoft, it is an honor to be able to learn from all of you. Thanks to the team behind the ITU's HPC, always helpful and effective. And thanks to Mahsa, my supervisor, for her extensive proofreading, and guidance throughout the project, I truly appreciate it.

"Gracias" no son suficientes para las personas que me han ayudado de la manera más profunda. Gracias a mis abuelos Abel y Ana María, por su amor y sabiduría. Gracias a Sofía, Elisa, Miguel y Anabelle, la familia que me ha hecho quien soy y que me ha enseñado lo que significa amar desinteresadamente.
Finalmente, gracias a la familia que me ha dado vital apoyo en esta etapa de mi vida. Gracias a mis primos Karla y Oskar, a mi tía Patricia por toda su entrega, amor y paciencia. A mi tío, Anders Sørensen, osito, que me enseñó tanto en tan poco tiempo. Continuo aprendiendo de ti, te admiro, extraño y llevo en mi corazón por el resto de mi vida.
Más alla de las palabras, los amo. Todos mis logros son en verdad suyos.

[1]A TCP joke, because less people would get a UDP one.[2]
[2]Yes, that was another joke.

Abstract

In this project, we present the evaluation of test selection and prioritization techniques, with data collected from a Continuous Integration (CI) pipeline of the Microsoft *Business Central* project.

Having an effective technique for test prioritization in a project is relevant in Continuous Integration pipelines. Their execution can be time-consuming and energy inefficient. Furthermore, these are typically executed multiple times per day in a large team. Speed and efficiency not only benefit the development cycles but overall development costs and energy impact.

We reduce the test prioritization problem to a ranking problem. This is done by extracting numeric representations of the changes in a codebase, and using them as input vectors to different ranking algorithms. In contrast to existing research, we use different representations; most notably, the usage of test coverage information.

Our results show that for the datasets created, using coverage information for assigning priorities increases the effectiveness of the prioritization used. However, using coverage information for representing changes reduces its effectiveness.

Based on our results, we argue that with this approach to prioritization we would be able to reduce the number of tests that were run by 60% using a conservative estimate of the best performing algorithm, which corresponds to an execution time improvement of 90% in average. This suggests that these approaches are well suited for the pipeline under study.

Furthermore, we also give an outline of how this prioritization could be used in different stages of the existing pipeline.

Contents

Chapter 1

Introduction

Large software systems are a composition of intertwined units of functionality, often related in complex ways. They can be developed across many years and by large teams. It is no surprise that complexity arises quite easily.

It becomes progressively harder to add new features and improvements while being sure that everything else works as it should. That is, the software remains correct. To tackle this problem of complexity, many strategies and different angles have been studied and implemented by large-scale software systems.

Automated regression testing [1] is one of the main strategies widely adopted by the industry [14], to ensure that changes to the code will not negatively affect the users of the system.

Automated regression testing consists of developers writing automated tests: code that executes scenarios that a user would experience and asserts that the desired properties hold. Afterwards, when changes to the code are done, executing successfully the set of automated tests gives confidence that such scenarios will not be impacted by the change. In practice, this is an effective technique to keep the quality of the system and allow for its evolution. [14][1]

Another challenge in software engineering consists in having an effective development process when working with large teams. Many strategies have also been studied, and the industry consensus is to use *Continuous Integration* (CI) processes.

The main idea of the CI software development process is that developers should integrate code changes frequently. When developers integrate their changes, they make other developers on the team aware of their changes. The other developers can then react accordingly if the change conflicts with their work.

Frequently integrating changes to the product increases the risk of making mistakes. For that reason, as a safeguard, the set of automated tests should be run before integrating every change. The process of building and testing the code automatically every time a change is meant to be integrated is called a CI pipeline and it is typically enforced in the development processes of industrial software products.

By frequently integrating changes with the regression test suite backing its correctness we ensure quality in every small step.

However, there is a drawback: as the system evolves, more regression tests get added, and eventually, it can become time-consuming to run the complete set of automated tests. Even more, if this is done every time a developer wishes to integrate changes to the

codebase, which is the case in the context of CI.

To address this problem, academic research and industry have focused on several angles. Algorithms on Test Selection and Prioritization (TSP) are techniques that aim to automatically propose a relevant sorted subset of tests to execute. The rationale behind these techniques is that since the goal of executing the test suite is to find failing test cases for the given change, we can avoid executing unrelated tests by selecting a subset of them and executing first the ones that are more likely to fail.

Not only is optimizing CI pipelines relevant for improving the agility of software development, but it also has a positive impact on the energy resources they require. These pipelines are run multiple times a day, and the cost of operating the infrastructure required to keep the system evolving is significant.

For this project, TSP techniques are applied and evaluated for data collected from one of the CI pipelines of the *Business Central* project. *Business Central* is an Enterprise Resource Planning (ERP) software product by Microsoft. The aim is to find an adequate technique for this pipeline and to propose possible directions to improve the CI cycle feedback time and effectiveness.

The techniques explored are based upon previous work by Bertolino et. al. in [2], where the prioritization problem is transformed into a *ranking* problem. We then apply different *Learning to Rank* algorithms and evaluate their results. However, in contrast to their approach, we propose the usage of coverage information as a criterion for these techniques.

For clarity, we will focus on the following research questions:

- **RQ1**: Which of the ranking techniques yields the best prioritization results?
- **RQ2**: How much could we reduce the number of tests and their execution time?
- **RQ3**: What is the effect of using coverage information for *Learning to Rank* techniques?

Throughout this work, we also outline the different challenges and technical difficulties that adapting academic techniques posed when taking into consideration the practical realities of an industrial environment. Additionally, we discuss how these techniques could be used in the existing context of the pipeline.

All the code for the different stages of the project can be found in the accompanying repository.[1]

[1] `https://github.com/mynjj/msc-thesis-public`

Chapter 2

Background

2.1 Business Central

Business Central (BC) is an Enterprise Resource Planning (ERP) software system targeting Small and Midsized Businesses (SMB) from Microsoft. Its functionality spans several areas of a company such as finance, sales, warehouse management, and many others.

BC allows for Microsoft partners to provide custom extensions that suit customer needs as much as required. This is done through application extensions that developers can write in AL: a Domain Specific Language (DSL) for the application logic. These extensions modify the experience end-customers have with the product and enhance the product with any custom logic their business may require. Not only partners provide extensions for BC, all the business logic are first-party extensions that Microsoft developers maintain for the core business functionality of the product.

In this thesis project, we will focus on the CI pipeline and regression tests used for the business logic code (also referred to as *application* code). The project has several other pipelines for the different parts of the product, but for the scope of the project, we focus on optimizing the CI pipeline through techniques for test selection and prioritization targeting changes in the AL DSL.

2.1.1 The *application* CI pipeline and the DME system

The BC *application* code follows a traditional CI cycle for development, which we describe for completeness and clarity. Whenever developers complete tasks by making new changes to the *application* code, they create a Pull Request on the Version Control System (VCS) for the desired target branch. Before this request succeeds and the code is merged, certain checks have to be fulfilled, which include the execution of automated tests.

To execute automated tests, a *job* is started on the Distributed Model Executor (DME) system, the internal build system for the product. A job is defined by a set of tasks, defined by scripts that execute the different required stages. Such specification is called the *model* of the job. This set of tasks may have dependencies between each other. We can represent this set of tasks with dependencies as a Directed Acyclic Graph (DAG), where each task corresponds to a node and an edge corresponds to the target depending on the source.

All these tasks and the job definition are defined within the same Version Control repository as code, giving the application developers complete control over the job's ex-

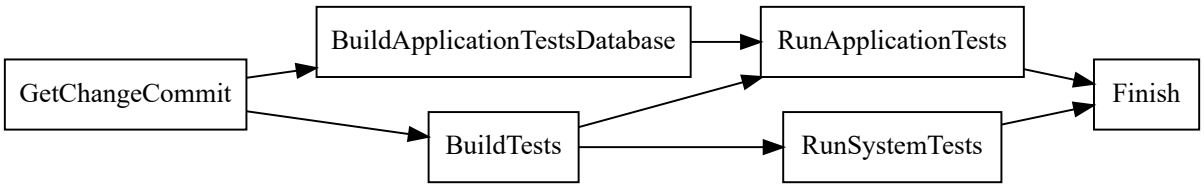

Figure 2.1: Example of tasks that depend on each other for a job execution represented as a DAG.

ecution. A Pull Request triggering the CI pipeline is not the only way a developer can request *jobs*. They can also require them at any point in the development cycle for their convenience.

Some of these tasks correspond to running different groups of tests. Therefore, the cost of executing certain tests belonging to a task depends on the dependencies the task has. We explain this by giving the following example: Suppose a *job* being executed is defined by a model consisting of the tasks shown in figure 2.1 as the associated DAG for such a set of task-dependencies.

We can see that the task `RunApplicationTests` depends on both `BuildTests` and `BuildApplicationTestsDatabase`. On the other hand, the task `RunSystemTests` only depends on `BuildTests`.

In this example, the DME system could assign two different machines to execute `BuildTests` and `BuildApplicationTestsDatabase`. Let us suppose that `BuildApplicationTestsDatabase` takes longer. When the task `BuildTests` is done, the task `RunSystemTests` can already begin as its only dependency has finished. However, `RunApplicationTests` will have to wait for its other dependency to start execution. In this scenario, running tests in the task `RunApplicationTests` has a higher cost than running tests in `RunSystemTests`.

This means that when executing test selection or prioritization considering the complete set of tests. Some of them entail a higher computing cost depending on the dependencies they have.

The DAG of task-dependencies for the *Application* pipeline of the *Business Central* project is not as simple as the given example. In fact, it is so large that we can not show it meaningfully on a page. However, to satisfy the reader's curiosity, figure 2.2 shows a complete diagram for the metamodel of this pipeline.[1]

2.1.2 Test Selection currently on BC *application* tests

In the current CI pipeline of the *application* code of BC, there is a selection method based on test coverage information. We will define more thoroughly the definition of a test selection method in section 2.2, for now, it suffices to think of it as selecting just a subset of tests from the complete set of tests. In this section, we review how tests are defined for *application* code, and what kind of coverage information we have available when executing tests.

[1] A metamodel in this context is a specification for the model that the jobs execute, so it is *smaller* and easier to show.

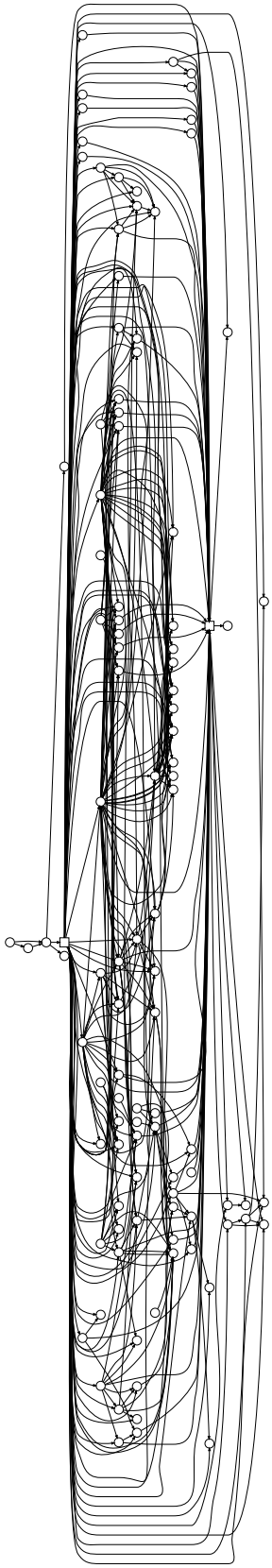

Figure 2.2: Associated DAG of the metamodel of job executions of the *Application* pipeline for the *Business Central* product.

```
codeunit 135203 "CF Frcst. Azure AI"
{
    Subtype = Test;
    TestPermissions = NonRestrictive;
    // ...
    [Test]
    procedure AzureAINotEnabledError()
    // ...
    [Test]
    procedure NotEnoughHistoricalData()
    // ...
    [Test]
    procedure FillAzureAIDeltaBiggerThanVariancePercNotInserted()
    // ...
}
```

Figure 2.3: Excerpt of the test codeunit 135203.

Application Tests in AL

We first give an overview of how tests are defined in AL, *Business Central*'s DSL for business logic. For a more thorough overview of the AL language and its role in the *Business Central* system, we refer to the appendix of this project.

AL organizes the code in *objects* [2], these *objects* represent different units of functionality for the different features of the product, for example, tables in a database, or pages the user can interact with. A common type of *object* is a *codeunit* which is conformed by different procedures that can be called from any other *AL object*[3].

An *application* test is written also in AL, as a *codeunit* with specific annotations for it to be identified by test runner applications. In figure 2.3 you can see an excerpt of a group of test procedures defined under a test codeunit. From a more general perspective, we can think of test codeunits as groups of test scenarios.

Test runners are also implemented in AL. These are programs that run the test codeunits, with different settings, like permission sets to use, or whether or not the test runner should persist changes after test execution, which can be desirable according to the scenarios the developer wishes to test. Currently, the project has two different implementations of test runners, used for different sets of tests, a result of historical legacy. For clarity, we distinguish them as *CAL Test Runner* and *AL Test Runner*.

In the current *application* CI pipeline, an *Application Test* task on the job model can use either of these two test runners depending on their needs, or historical legacy. In the current model definition, roughly half of the test tasks are using *CAL Test Runner* and the other half the *AL Test Runner*.

Aside from historic differences, for our purposes, we highlight a significant difference: The *CAL Test Runner* can produce coverage information of test execution, and the *AL Test Runner* currently does not.

[2]Note that objects in this context do not correspond to objects in the traditional sense of Object-Oriented Programming.

[3]*Modules* are a similar concept analogous to *codeunits* in more traditional languages like Python, NodeJS, Rust, Haskell, among others.

```
"Table","4","17","1"
"Table","4","18","1"
...
"Table","14","112","1"
"Table","14","127","1"
...
"Codeunit","28","111","1"
"Codeunit","28","112","1"
"Codeunit","28","115","1"
"Codeunit","28","116","1"
"Codeunit","28","119","1"
"Codeunit","28","122","1"
...
```

Figure 2.4: Excerpt of the coverage file for the test codeunit 135203. The columns correspond to Object Type, Object Id, Line Number, and Number of Hits

```
1  codeunit 28 "Error Message Management"
...
106    local procedure GetContextRecID(ContextVariant: Variant; ...
107    var
108       RecRef: RecordRef;
109        TableNo: Integer;
110    begin
111        Clear(ContextRecID);
112        case true of
113            ContextVariant.IsRecord:
114                begin
115                    RecRef.GetTable(ContextVariant);
116                    ContextRecID := RecRef.RecordId;
117                end;
```

Figure 2.5: Some of the lines covered by test codeunit 135203.

Coverage information

AL has implemented a language primitive that can be used to record information about which lines of code of which objects were executed. In a broad sense, this is how the test coverage information from the *CAL Test Runner* is produced. After setting up the context for each test, this primitive is used to record activity on each AL object throughout execution.

The *CAL Test Runner* currently has three different kinds of output relating to coverage information. We will use information that will allow us to: given a test codeunit executed by the test runner, obtain a list of the different *AL objects* and lines within these objects that were run by the test execution.

For concreteness and to further illustrate the available information, we show an example. In figure 2.3 we have an excerpt of the test codeunit 135203, containing several test procedures. In figure 2.4 we can see a corresponding excerpt of its coverage file. This coverage information was collected for a given state of the codebase.

Each row on the coverage file corresponds to a line being executed when executing test codeunit 135203. The first 3 columns identify each line by giving the Object Type and Object Id the line belongs to, and the Line Number to be considered. The last column corresponds to the number of times the line was executed. For example, in figure 2.4, we can see that Codeunit 28 had among others lines 111, 112, 115, and 116 executed once. In figure 2.5 we can see such lines.

Test Selection with coverage information

Currently, the CI pipeline performs test selection for tasks executed with the *CAL Test Runner*, for which coverage information of each executed test codeunit is recorded.

The selection technique is based on obtaining the lines and files affected by integrating a developer's change and finding such lines on the recorded coverage information.

For all those matching lines, the corresponding test codeunit is selected for execution. This effectively selects every test which traversed during its execution the lines being changed by the developer.

However, for tests being run by the *AL Test Runner*, this selection is not available as the corresponding coverage information is not available. We could classify this selection technique as a partial selection based on coverage traces.

As we will explain in section 3.1.1, we will use this coverage information, and add features to the *AL Test Runner* to allow for collecting coverage traces as well.

2.2 Test Selection and Prioritization techniques

Several Test Selection and Prioritization techniques have been studied in the literature both by academia and industry.

In this context, the problem of test selection is defined as: given a change to the codebase, and a complete set of tests; to obtain a subset of tests, such that the capacity of fault detection of the test suite is not lost. This means that if for such change the test suite execution results in failure, this subset of tests should fail as well. A stronger condition is given by a *safe* selection algorithm [15], which requires every failing test case to be included in the selection.

The problem of test prioritization has the same inputs, and it aims to find a sorting for the tests to be executed that prioritizes running the tests that are more likely to fail first.

Intuitively our aim in this problem is that: given a change in the codebase, and a complete test suite to execute; determine a subset (selection) that does not miss any fault-revealing test case, and a sorting (prioritization) that increases the probability of finding failures earlier.

It is worth emphasizing that given a prioritization technique, we can induce selection techniques by selecting the subset of tests that were prioritized the highest. The size of such selection could be determined by a given size, duration, or other criteria. This is how we obtain the selected subsets evaluated in chapter 4.

2.2.1 Related work

Yoo and Harman in their survey [20] present a detailed overview of techniques used for regression test selection, prioritization, and minimization[4]. In this study, several foundational works on this area are presented, including formal definitions, and metrics to evaluate this problem.

[4]Minimization is a problem not dealt with in this project, it consists of removing superfluous tests from a test suite

In the literature, we can find similar techniques to the already implemented selection technique in the pipeline under study. As explained in section 2.1.2, the CI pipeline of interest of the product collects coverage information for some subset of test tasks and uses the information of which files were changed by the developer to run only a subset of tests. This intuitive approach aims to select *modification-traversing* tests [16], which was one of the first approaches studied by Rothermel and Harrold, and widely studied by many others. The approaches differ in how a test is determined to be *modification-traversing*, as some approaches use Control Flow Graphs [15], others use execution traces (Vokolos and Frankl in [18]) and some others use coverage information. Like the work of Beszédes et. al. [3], where they describe populating a coverage database for the C++ components of the WebKit project and identify the changed modules from a given revision.

For the techniques evaluated in this project, other literature reviews like [13] by Pan, et. al., present more similar techniques, as they give an overview of Selection and Prioritization techniques using Machine Learning (ML). In this study, they classify the approaches into four groups: Supervised Learning, Unsupervised Learning, Reinforcement Learning, and Natural Language Processing-based.

In our project we focus on Supervised Learning techniques, as found in the work by Bertolino et. al. in [2]; in their work, they build upon previous work by Spieker et. al. in [17] by comparing Reinforcement Learning approaches to Supervised Learning approaches using ranking algorithms. In these two papers, different features are used to characterize test executions, in [17] they only use test history information and duration, and in [2] they use features concerning program size, McCabe's cyclomatic complexity, object-oriented metrics, and test history. In contrast, we propose a different set of features, metrics relevant for the AL DSL, and coverage information available, which is not considered in these works.

Busjaeger and Xie in [5], evaluate applying a ranking algorithm, SVM rank, for prioritization in the case of Salesforce. We highlight that for the features representing the changes a developer made, they use coverage information by proposing a coverage score, to reduce the impact of outdated coverage information. In contrast, we use and compare different ranking algorithms.

2.2.2 The ranking problem in test prioritization

Previous research has focused on interpreting the problem of test prioritization as a *ranking* problem. In this section, we give a brief overview of this problem, and some of the algorithms proposed as a solution. In particular, we explain the algorithms which were the focus of this project and how we will interpret the problem of test prioritization as an instance of a ranking problem.

In the context of Information Retrieval, the goal of the ranking problem is to obtain relevant resources from a potentially large collection of those resources for a given information need (query). Ranking algorithms are relevant to different problems, for example in search engines, or recommender systems.

An approach that has been the subject of extensive research in recent years is *Learning to rank*, a set of ML techniques with the goal of obtaining a *ranking model* out of training data. This model is reused when unseen queries are given, to obtain a similar ranking of the documents as for the training data.

A *ranking model* is a mathematical model that given D a collection of documents and a query q, it returns an ordered set of every element of D. Such ordered set is sorted according to some relevance criterion.

In the case of CI cycle optimization with Test Prioritization, we interpret the query q as the change that the developer wants to commit to the target branch. The set of documents D corresponds to the complete test suite. Our relevance criterion corresponds to sorting the failing tests first (if any) and the rest of the tests can be sorted through different criteria like duration or test coverage as we will explain in section 3.3.

With this interpretation of the Test Prioritization problem, there is still freedom in two aspects: the representation of the query q for a given codebase change, and the relevance criterion to use. As part of our experiments, we will consider variations of these aspects as explained in chapter 3.

We will first describe the different metrics that are used in the ranking literature, and then give a brief overview of the different ranking algorithms used for completeness.

2.2.3 Metrics for the ranking problem

Research on ranking algorithms has given a diverse set of metrics to compare and evaluate rankings proposed by these algorithms.

A common metric used to evaluate ranking algorithms is the Discounted Cumulative Gain (DCG), which was proposed by Järvelin, et. al. [11]. `DCG@k` is defined for k the truncation level as:

$$DCG@k = \sum_{i=1}^{k} \frac{2^{l_i} - 1}{\log(i+1)}$$

Where l_i is the relevance given to the i-th result. We can see that given a ranking to evaluate, this metric increases when the first values of the ranking are given a high relevance. In contrast, high relevance values encountered later are penalized by $\log(i+1)$.

The truncation level just limits the considered documents for this metric.

`NDCG@k` is the Normalized version of `DCG@k`. This metric compares the obtained ranking against the ideal ranking for that query and computes its corresponding `DCG@k` denoted by $IDCG@k$:

$$NDCG@k = \frac{DCG@k}{IDCG@k}$$

Another metric is the Mean Average Precision (MAP) [21], which is based on binary classification metrics. Traditionally, precision and recall are widely used for binary classification. In the context of Information Retrieval, given a query, *precision* refers to how many documents predicted as relevant are labeled as relevant, *recall* refers to how many of the documents were correctly classified from all the documents labeled as relevant.

The average precision (AveP) represents the area under the curve of a precision-recall plot when considering the first ranked elements:

$$\text{AveP} = \sum_{k=1}^{n} P(k)|R(k) - R(k-1)|$$

Where $P(k), R(k)$ are the precision and recall obtained for the first k results.

While our proposed prioritizations to label the dataset are not binary, they can be considered binary by giving a cutoff point for the assigned relevance.

The Expected Reciprocal Rank (`ERR@k`) metric, was proposed in [6] by Chapelle, et. al. .It is designed to take the relative ordering between ranked results into consideration. In contrast to `DCG`, it does not give the same gain and discount to a fixed position. It is defined by:

$$\sum_{r=1}^{k} \frac{1}{r} R_r \prod_{i=1}^{r-1} (1 - R_i)$$

Where R_i is defined as:

$$R_i = \frac{2^{l_i} - 1}{2^{l_m}}$$

Where l_m denotes the maximum relevance value of the ranking, and l_i is the relevance given to the i-th result.

While this is the least intuitive of the metrics, it correlates better with empiric measurements of search engine applications[6]. It is based on modeling the probability of a user finding its query at a given document position.

As part of our experiments described in chapter 3, we evaluate different metrics used to train the ranking algorithms.

2.2.4 *Learning to Rank* algorithms

We will now briefly describe the different ranking algorithms explored for this project.

Coordinate Ascent

Coordinate Ascent is a general *optimization* method, it is based on iterations defined by maximizing the given function f when fixing all coordinates but one. Formally, the $(k+1)$-th iteration, has as i-th component:

$$x_i^{k+1} = \arg\max_{t \in \mathbb{R}} f(x_1^k, ..., x_{i-1}^k, t, x_{i+1}^k, ..., x_n^k)$$

This method has similar convergence conditions as gradient descent. It was first proposed as a ranking method by Metzler and Croft in [12], and it has successfully been applied to different ranking problems. We give a short overview of how this optimization method is used as a ranking method, but we refer to the above-mentioned article for a detailed explanation.

The optimization is used for maximizing the different ranking evaluation metrics presented in section 2.2.3. Metzler, et. al. propose a linear model to predict the score of each document and add constraints to the parameter space being optimized.

To make it more concrete, the ranking is induced by a scoring function S, for a document D and query Q of the following form:

$$S(D;Q) = \Omega \cdot f(D,Q) + Z$$

Where $f(D,Q)$ denotes the feature vector for such document and query, Z denotes a fixed constant, and Ω denotes the free parameters to be optimized. In the ranking library used for the implementation in this project, called Ranklib, Z is set to zero.

MART

Regression algorithms using gradient boosting were first proposed by Friedman in [9], Multiple Additive Regression Trees (MART) is a gradient boosting technique further developed by Friedman and Meulman in [10].

As a general introduction, we provide a high-level overview of the method. We refer the reader to [9] for a more detailed explanation.

In general, this is a regression technique that approximates a function by minimizing some related loss function. The idea is to use a linear combination of M different models called weak learners:

$$F(x) = \sum_{i=1}^{M} \gamma_i h_i(x)$$

Where h_i denotes each of the weak learners, and γ_i are constants found during training.

The idea is to fit a regression tree to approximate the target function and use the next regression tree to approximate the residuals of the first. Afterwards, greedily compute a scale (γ_i above) for the weak learner that minimizes the loss function.

These residuals approximate the gradient of the loss function, effectively making this method follow the same rationale as gradient descent to minimize the loss function.

Regression algorithms such as MART can be used to rank by regressing a relevance function for each of the documents to rank, minimizing some of the different training metrics presented in section 2.2.3.

Methods for ranking induced by regression algorithms such as MART are called *pointwise* ranking algorithms because they only consider the information of a single document to determine its relevance.

LambdaMART

LambdaMART[4] takes its name from its constituents: LambdaRank and MART. In the previous section, we explained how MART works for general regression and ranking.

In contrast to *pointwise* algorithms that only consider a single document to assign its relevance, *pairwise* algorithms consider the relative ordering of pairs of documents. In the case of LambdaMART, the aim is to produce a comparison function between documents. With a comparison function, we can obtain the ranking by sorting.

For the case of this family of algorithms, the aim is to obtain a function that given two documents x_i, x_j, obtains the probability that for a given query q, x_i is ranked with higher than x_j: P_{ij}. With such a comparison function, sorting of the complete set of features can be performed.

To do so, the trained model is a function f that only takes as input a feature x_i, and outputs a real value $f(x_i)$. To obtain the probability of the pairwise comparison a logistic function is used:

$$P_{ij} = \frac{1}{1 + e^{-\sigma(f(x_i) - f(x_j))}}$$

Where σ denotes the logistic growth rate. The loss function used is the cross-entropy measure. Minimizing through gradient descent is the idea behind predecessors of this algorithm like RankNet and LambdaRank.

In LambdaMART this gradient is not computed but predicted by boosted regression trees. We refer to [4] for a more thorough explanation of how these gradients are reduced to scalars subject to optimization by weak learners.

The method has been successfully applied in diverse ranking applications, and in particular, it performed the best in the analysis given by Bertolino, et. al. in [2] on Test Prioritization.

RankBoost

Rankboost was proposed by Freund, Yoav, et. al. [8]. Similar to MART, it uses *boosting*, combining several *weak* learners into a single model.

Each of these learners predicts a relevance function, and therefore a ranking. For training, a distribution D over $X \times X$ is required, where X is the documents on a query. For this reason, this method is $O(|X|^2)$ in memory, which can restrict the applicability of this algorithm.

The distribution D represents which pairs of documents are more relevant to order correctly with regards to optimizing ranking evaluation metrics. Each learner updates the distribution D, emphasizing the pairs that are more relevant for the algorithm to properly order in that iteration. The final relevance function then becomes a linear combination of each of these learners. We refer the reader to [8] for more details and further reading.

AdaRank

This algorithm was proposed by Xu and Li in [19], it was designed to directly minimize Information Retrieval (IR) performance measures like `MAP` and `NDCG`. It is based on AdaBoost, a binary classifier also based on *boosting*, obtaining a model from the linear combination of several *weak* learners.

On each iteration, it maintains a distribution of weights assigned to each of the training queries. Such distribution is updated, increasing the values of the queries that the weak learner ranks the worst. In this way, subsequent learners can focus on those queries for the next rounds.

The weak learners proposed in [19] are linear combinations of the metrics to minimize the weight distribution.

2.2.5 Metrics for evaluating Test Selection and Prioritization

While the problem of ranking has been widely studied and metrics have been proposed for evaluating it; it is more meaningful for our purposes to evaluate the resulting prioritization with metrics in the context of regression testing.

In this section we expand upon some of the metrics previously proposed in the literature, to evaluate the problems of Test Selection and Prioritization.

Test selection execution time

For evaluating selection algorithms, a natural approach is to measure the time it takes to run the subset. To make this metric test suite independent a ratio is used:

$$t_x = \frac{t_S}{t_C}$$

Where t_S is the time taken to execute the selection and t_C is the time taken to execute the complete test suite. Ideally, the aim is to obtain values close to zero for this metric, as this represents a significant improvement in execution time. In contrast, values close to one represent a similar execution time as the complete test suite.

Inclusiveness

In test selection, we do not only rely on the execution time for evaluation. Consider an arbitrarily small subset selected, it would yield good results, but potentially it could also miss some fault-revealing test cases.

To consider this, inclusiveness is introduced:

$$i = \frac{|S_F|}{|T_F|}$$

Where S_F is the set of fault-revealing test cases from a selection, and T_F is the set of test faults in the complete test suite. For completeness, we define $i = 1$ when there are no test faults in the given change.

A *safe* test selection algorithm [15] always has $i = 1$. As every fault-revealing test is included in the selection.

Selection size

On the other hand, high inclusiveness could also be a sign of over-selecting test cases. For example, selecting the whole test suite trivially has $i = 1$. To have a measure of how big the selection is, *selection size* is defined as:

$$ss = \frac{|S|}{|T|}$$

Where S denotes the set of selected tests and T denotes the complete set of tests. A good selection algorithm strives for having a small selection size, while high inclusiveness.

Time to the first failure

Likewise, another time-related metric of interest for prioritization is the time it takes to reach the first failure:

$$t_{ff} = \frac{t_F}{t_C}$$

Where t_F is the time taken to reach the first failure for the proposed prioritization.

Normalized Average of the Percentage of Faults Detected

For prioritization, only focusing on time to get to the first failure is skewed. As one could have detected the first failure soon while prioritizing the rest of the failing test cases with low priority.

To overcome this, a widely used metric, proposed by Elbaum, Malishevsky, and Rothermel in [7] is the Average of the Percentage of Faults Detected (APFD).

Normalizing this metric to the number of failures detected allows considering cases where no selected test case was failing. This is useful for evaluating prioritizations that had previously some selection criteria applied.

It is defined by:

$$NAPFD = p - \frac{\sum_{i=1}^{m} TF_i}{nm} + \frac{p}{2n}$$

Where p denotes the ratio of detected faults on a selection against the total amount of faults, n denotes the number of test cases, m denotes the total number of faults, and TF_i denotes the ranking of the test case that revealed the i-th failure.

This metric represents the proportion of the test failures detected against each executed test. We aim for this value to be close to 1, representing that the accumulated amount of failures detected is obtained early with the prioritization.

Chapter 3

Method

To use *Learning to Rank* approaches in our context we require to:

- Collect CI jobs, along with information on the tests executed. Since we are also using coverage information, we also need to collect this information.
- Obtain vectors representing each test execution for each change to the codebase. These vectors represent the *queries* in the context of *Learning to Rank*.
- Label the training dataset by assigning rankings to each. These labels should reflect prioritizing failing tests, the remaining tests could be prioritized with different criteria.
- Train the models with the different parameters that could be considered for each. This training is performed with the training dataset produced after representing each query and assigning priorities to them.
- Use the models to obtain rankings for the validation dataset. With these prioritizations, we can obtain the different evaluation metrics that we require to answer our research questions.

In the remainder of this chapter, we discuss more details of these stages.

3.1 Collecting the dataset of CI pipeline executions

As the initial step of our project, we collect information about the CI pipeline executions from data stored by the DME build system and the history of the repository.

For each CI job, the information extracted was:

- Execution model with the tasks that the build system used as input.
- Job execution properties: duration, result, and date.
- For each of the tasks executed by the job, information on properties: duration, result, and date.
- Other meta information to identify the job in the Version Control System.

- For each of the *application test* tasks, the result of each procedure run for each of the test units, along with information on the duration of its execution.
- Comparison with the last merge from the target branch: path and directories where changes were made, type of changes performed, and the content of modified files.

The aim was to collect enough properties to represent the changes a developer made, along with data to evaluate the prioritized tests.

3.1.1 Coverage information for test runners

The information listed in the previous section was collected from real operation data of the pipeline. As explained in section 2.1.2, there are two different implementations of test runners that the tasks may use.

As explained in section 2.1.2, in the current pipeline, tests run with the *CAL test runner* do have coverage information. However, that is not the case for tests that use the *AL test runner*.

A complete coverage report for all the application tests was not initially available. However, as part of this project, modifications to the *AL test runner* were done to allow for collecting the same kind of information[1]. However, these changes were not integrated into the pipeline. Instead, the changes made to the test runner were run against snapshots of the codebase in a given time.

It has been discussed previously in research how coverage information may be outdated and hard to maintain [2]. This is partly true in our case as well, however, we acknowledge that the information given by coverage can be valuable for our problem.

Busjaeger, et. al. in [5] propose a more robust approach to using coverage information, by defining a coverage score. A feature like coverage score mitigates the lack of accuracy of the coverage information. As additional mitigation to this problem, we introduce windows to compute such coverage scores as it will be shown in section 3.2.

3.1.2 The collection process

As a general overview, over some time, real CI jobs in this pipeline were collected. Sporadically between these jobs, custom jobs were submitted to the build system with the required changes to the *AL test runner* to collect coverage information.

In the next sections, whenever coverage information is required to compute properties of a given CI job, the coverage information used will be the closest earlier collected one.

3.2 Representing tests and codebase changes by a vector

For each of the collected CI jobs, we obtain features to characterize the changes made by the developer, for each of the executed tests in the job.

We consider three categories of properties used to obtain the vector representation of a test's execution:

[1]The changes done were based on previous work by Nikola Kukrika (nikolak@microsoft.com)

- Properties of the change to the AL codebase: Properties related to the changes of the AL files in the codebase.
- Test history properties: Properties related to the historical behavior of the test.
- Coverage properties: Properties related to the coverage information of each test and the change considered.

In the rest of this section, we list the properties considered in each category. In section 6.3 we discuss some of the different features that future work could consider found in the literature.

3.2.1 Properties of the change to the AL codebase

We use the following quantities to represent the changes done to the AL codebase:

- Number of new AL tables.
- Number of new AL objects.
- Number of modified AL objects.
- Number of removed AL objects.
- Number of changed tests.
- Number of added lines to AL objects.
- Number of removed lines from AL objects.
- Number of changed files that are not AL objects.

These quantities represent numerically the typical changes experienced in the AL DSL. These properties are analogous to the *Program size* and *Object-oriented* properties used by Bertolino et. al. in [2], which are not directly applicable to our case.

3.2.2 Test history properties

For each of the tests in a job we add the following historic properties:

- Proportion of times the test has failed within the available data.
- From the past k job executions, the proportion of times the test has failed.
- Average duration of this test in previous executions.

3.2.3 Coverage properties

To understand the effect that coverage information has(**RQ3**), we also consider coverage properties to represent the changes done to the codebase for each test.

Using the most recent coverage information collected previous to the given job, we compute:

- Ratio between lines covered by that test and the average.
- The number of files changed that were covered by that test.
- The number of lines changed that were covered by that test within different *windows*.

The lines changed were not matched exactly but by *windows*. If a change on a line nearby was covered, it was counted for such properties. We added features for different window sizes.

The aim of such is to reduce the impact of having outdated coverage information.

3.3 Prioritizing Test Executions in a CI job

As part of our training dataset for the ranking algorithms, we need to associate a priority value to each of the tests executed in a job. This relevance induces the ranking that we desire the algorithm to learn. This is the process commonly referred to as *dataset labeling* in the context of ML.

It is worth emphasizing that this prioritization is not the one against which the results will be evaluated, as this would be biased. The evaluation will be given only by the TSP metrics presented in section 2.2.5.

A ranking can be defined by a priority function (also called relevance function): when each test case is assigned a priority, this value can be used to produce a ranking where such priority values are in increasing order.

We explain the two different approaches taken to assign priority functions. We created training datasets with both of these priority functions.

3.3.1 Failure and Duration decreasing exponential priority

In [2] Bertolino, et. al. propose a priority function for each test case, based on its duration and outcome. They define a score for the i-th test case R_i by:

$$R_i = F_i + e^{-T_i}$$

Where $F_i = 1$ if the test fails, 0 otherwise, and T_i the duration of its execution.

By design, this score ranks first failing tests and breaks ties via their execution duration. Furthermore, by this being an exponentially decreasing function, changes in duration have a larger effect for tests with small duration, than for test executions with large duration.

3.3.2 Coverage discrete priorities

We propose a discrete priority function using coverage information, since as stated in **RQ3** we aim to understand the effect of using coverage information in different criteria of the *Learning to Rank* techniques.

Given the i-th test, we define L_i as the number of lines covered by this test.

For a given job executing a subset of tests τ, we define $\mu_{L,\tau}$ to be the mean of all values $\{L_i\}_{i\in\tau}$. For such job, our proposed priority function prioritizes tests in the following order:

- Failing Tests
- New or modified tests in the job
- Tests that have no coverage information
- Tests covering changed lines, where $L_i \geq \mu_{L,\tau}$
- Tests covering changed lines, where $L_i < \mu_{L,\tau}$
- Tests where $L_i \geq \mu_{L,\tau}$
- Tests where $L_i < \mu_{L,\tau}$

This prioritization ranks first failing tests, and then as a conservative approach the modified tests in the change, and tests for which we do not have any coverage information, for example, new tests. Afterwards, we assign a higher priority to tests that traverse the changed lines on their execution, and finally the other unrelated tests. We break ties between them by the density of lines covered by each test since intuitively, a test covering more lines has a higher likelihood of failing.

As described in section 3.1, for each of these jobs we do not have the exact coverage information, but the most recent, previously collected.

3.4 Training ranking models

For each training performed, we varied the training metrics presented in section 2.2.3.

Additionally, for algorithms based on regression trees like MART and LambdaMART, we varied the number of trees for gradient boosting.

As explained in previous sections, we produced different datasets changing the prioritization criteria and the features that describe each change. As means of identification for evaluation, we name them as described in table 3.1.

We considered these different combinations of prioritization criteria and features, in accordance with the research question **RQ3**, as we aim to understand the effect that coverage information has when applying *Learning to Rank* techniques. Therefore we create datasets with and without coverage information to compare their effectiveness.

Dataset name	Prioritization	Features of each test execution
EP-NCI	Decreasing exponential as in [2]	• Properties related to AL changes • Test history properties
EP-CI	Decreasing exponential as in [2]	• Properties related to AL changes • Test history properties • Coverage properties
CP-NCI	Coverage based as described in section 3.3.2	• Properties related to AL changes • Test history properties
CP-CI	Coverage based as described in section 3.3.2	• Properties related to AL changes • Test history properties • Coverage properties

Table 3.1: Naming of the different datasets created.

Chapter 4

Results and Evaluation

4.1 Experiment setup

The collection process was executed for over a week. We retrieved information from 172 real CI jobs from the pipeline under study. We submitted two coverage-collecting jobs to the build system and collected the coverage traces.

The scale of the collected information limited the number of jobs we were able to collect and process.[1]

Each of the datasets considered: `EP-NCI`, `EP-CI`, `CP-NCI`, and `CP-CI` consisted of 172 queries corresponding to the collected CI jobs, from which 20% of the failing test jobs were used as the validation dataset. Each of these jobs contains around 18000 tests.

It is worth comparing the size of this dataset with the provided datasets from the literature. Spieker et. al. in [17] use the *Paint Control* dataset consisting of 180 cycles with an average of 65 tests each. In [2], Bertolino, et. al. consider datasets of 522 cycles with an average of 22 tests each, which they created from the Apache Commons project's commit history.

The training of the different *Learning to Rank* algorithms was performed with the different datasets and varying criteria. For implementation, we used Ranklib 2.17. The training was performed with the High-Performance Cluster from the IT University of Copenhagen. The minimum resources required for each of the training jobs were 115GB of RAM, which resulted in a training time with an upper limit of 2 hours for nodes with processors with over 32 cores.

Using the trained models to produce a prioritization was lightweight to perform on a regular computer. This negligible process would be the overhead that the CI pipeline would have to execute to obtain the proposed selections of these approaches.

4.2 Overview of the results presented

First, we use the NAPFD metric as defined in section 2.2.5 to compare the effectiveness of the rankings for the Test Prioritization problem.

For each ranking algorithm, we present how the different criteria used influence the NAPFD behavior. To emphasize, the varying criteria for the experiments were:

[1]The information on CI job executions amounted to 6GB of data and the coverage data to 44GB.

- Priority function used to label the training dataset: either the exponential prioritization proposed by Bertolino, et. al. in [2], or the discrete coverage prioritization we proposed.
- Features characterizing each test run: we included features related to AL file changes, test execution history, and varied either considering coverage properties or not.
- Training metric used for training the ranking algorithms: we use the different ranking evaluation metrics explained in section 2.2.5.
- For MART and LambdaMART the number of regression trees used: as explained in section 2.2.4, these methods combine several regression trees as weak learners. We use 5, 10, 20, and 30 trees in our experiments.

For each algorithm, we present box plots of the distribution of this metric for some of these configurations.

Additionally, for each of these ranking algorithms and configurations, we induce selection algorithms by taking the tests for which the model predicted the highest priorities. To determine the number of tests that we take from a prioritization we use the following criteria:

- A strictly safe selection `S-SEL`: we take tests from the prioritization until for all the evaluated jobs, every test failure is included.
- An above 80% average selection `80-SEL`: we take tests from the prioritization until the average of their *inclusiveness* is at least 80%.
- An above 50% average selection `50-SEL`: we take tests from the prioritization until the average of their *inclusiveness* is at least 50%.

These selections were considered to support the research question **RQ2**, to obtain definite quantities of the number of tests that could be omitted from execution in this pipeline.

For a complete set of values shown in the distribution plots and more evaluation metrics, see the appendix of this project and the evaluation data provided in the accompanying repository.

4.3 Results for Coordinate Ascent

Using different training metrics did not have a large impact on the behavior of this algorithm. However, the highest average and lowest variance of the NAPFD of different datasets ranked was obtained with the `NDCG@30` training metric.

Additionally, results show that using coverage information as part of the features characterizing a change results in lower values of NAPFD. However, for the coverage priority function, the NAPFD values were consistently higher.

The best results for this algorithm, across the different metrics, were obtained by datasets that do not use coverage information in the features, but that use the coverage prioritization we proposed. In the worst case for these datasets, the NAPFD of the proposed test ranking can be as low as 67%, but as high as 99%.

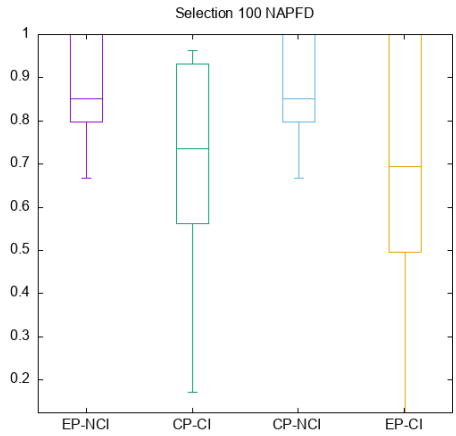

Figure 4.1: Distribution across the different datasets of NAPFD values for the Coordinate Ascent algorithm using the `DCG@10` training metric.

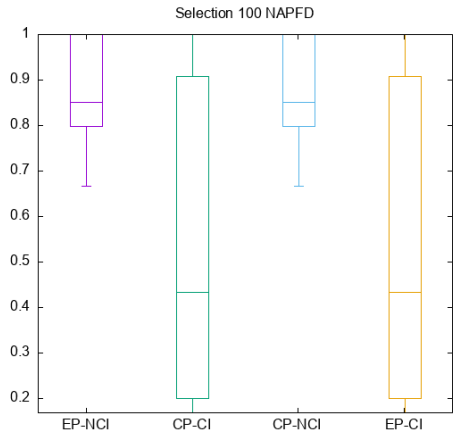

Figure 4.2: Distribution across the different datasets of NAPFD values for the Coordinate Ascent algorithm using the `MAP` training metric.

Figures 4.3 to 4.3 show the box plot of the distribution of NAPFD values for the jobs in the validation dataset for the different training metrics.

Additionally, for `NDCG@30`, we present the distribution of values across each different dataset of the *time to the first failure* metric in figure 4.3.

We observe that only in the case of the dataset with coverage information and coverage prioritization, the proposed prioritization achieves a value as high as 88% of the total execution time with an average of 14%. For the remaining datasets, the first failure is detected at most in the first 2% of the total execution time.

Regarding the induced selections, for the training metric `NDCG@30` and `CP-CI` dataset the following results were obtained:

- `S-SEL`: A safe selection was achieved with a selection size of 40%, corresponding to 10% of the execution time.
- `80-SEL`: A selection with average inclusiveness over 80% was achieved with a selection size of 40%, corresponding to 10% of the execution time.
- `50-SEL`: A selection with average inclusiveness over 50% was achieved with a selection size of 10%, corresponding to 3% of the execution time.

4.4 Results for LambdaMART

For LambdaMART, using the `MAP` metric, consistently ranked with a NAPFD value of 66% for any of the different trained datasets, as it can be seen in figure 4.4. With the `ERR@10` metric, the training did not converge, resulting in an invalid model.

Apart from these two metrics, the other training metrics behaved similarly across the different datasets. The best NAPFD value was obtained with the `DCG@10` metric.

Regarding the number of trees used for gradient boosting, the best performing values were obtained with 20 regression trees.

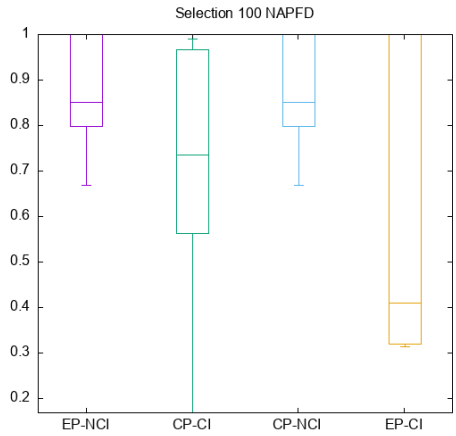

Figure 4.3: Distribution across the different datasets of NAPFD values for the Coordinate Ascent algorithm using the `NDCG@10` training metric.

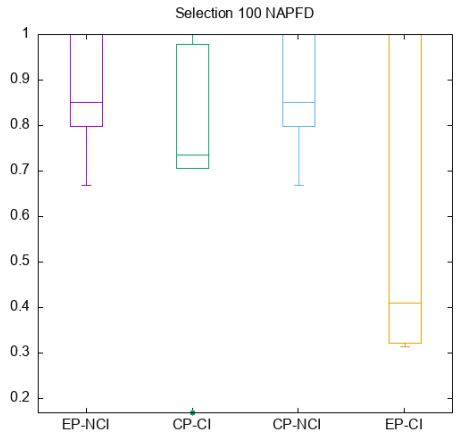

Figure 4.4: Distribution across the different datasets of NAPFD values for the Coordinate Ascent algorithm using the `NDCG@20` training metric.

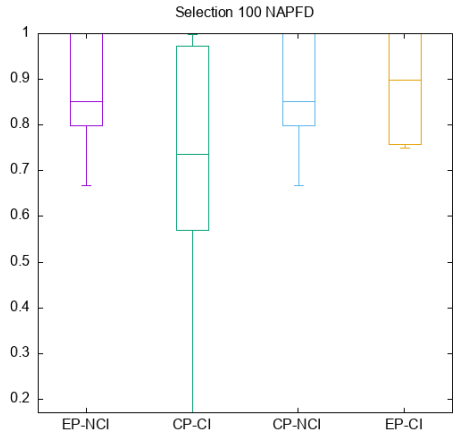

Figure 4.5: Distribution across the different datasets of NAPFD values for the Coordinate Ascent algorithm using the `NDCG@30` training metric.

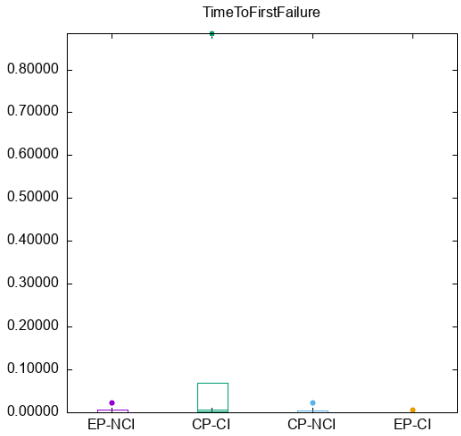

Figure 4.6: Distribution across the different datasets of times to first failure for the Coordinate Ascent algorithm using the `NDCG@30` training metric.

For this algorithm, using the coverage prioritization proposed yielded better NAPFD values. When using the exponential prioritization, using coverage information as features increased the NAPFD average and reduced its variance for the majority of the experiments.

In figure 4.4 we can see the comparison of distributions of *time to the first failure*, for the metric `DCG@10` and 20 regression trees. For this configuration, the `CP-NCI` dataset, which performed better across configurations, yields an average NAPFD value of 86%.

The resulting induced selections are:

- `S-SEL`: A safe selection was achieved with a selection size of 40%, corresponding to 9% of the execution time.

- `80-SEL`: A selection with average inclusiveness over 80% was achieved with a selection size of 40%, corresponding to 9% of the execution time.

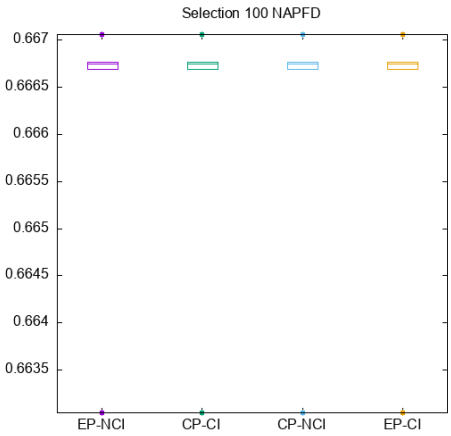

Figure 4.7: Distribution across the different datasets of NAPFD values for the LambdaMART algorithm using the `MAP` training metric and 30 trees.

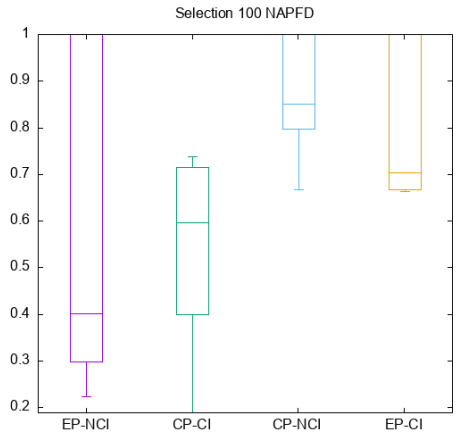

Figure 4.8: Distribution across the different datasets of NAPFD values for the LambdaMART algorithm using the `NDCG@10` training metric and 30 trees.

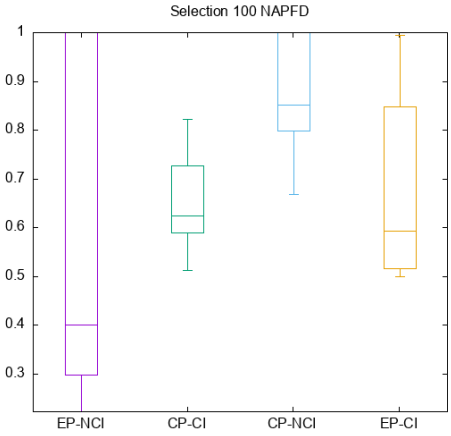

Figure 4.9: Distribution across the different datasets of NAPFD values for the LambdaMART algorithm using the `NDCG@10` training metric and 20 trees.

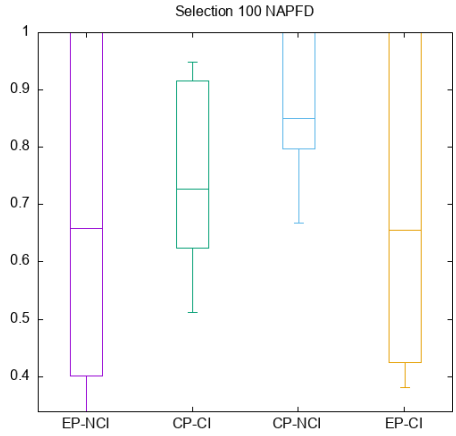

Figure 4.10: Distribution across the different datasets of NAPFD values for the LambdaMART algorithm using the `DCG@10` training metric and 30 trees.

- `50-SEL`: A selection with average inclusiveness over 50% was achieved with a selection size of 10%, corresponding to 2% of the execution time.

4.5 Results for MART

The results show no significant impact on the training metric used for this algorithm. The best configuration uses 30 regression trees.

As with the other algorithms, the dataset that provided the best results is the one using coverage information for prioritizing, but not using coverage information on the feature vectors.

For one of the best performing configurations, we can see in figure 4.5 a distribution of the *time to the first failure* values. The best configurations achieved an average NAPFD value of 67%.

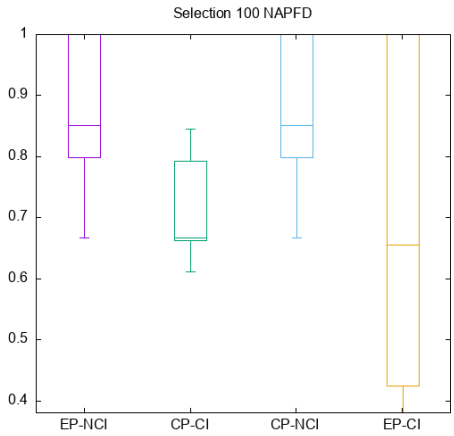

Figure 4.11: Distribution across the different datasets of NAPFD values for the LambdaMART algorithm using the `DCG@10` training metric and 20 trees.

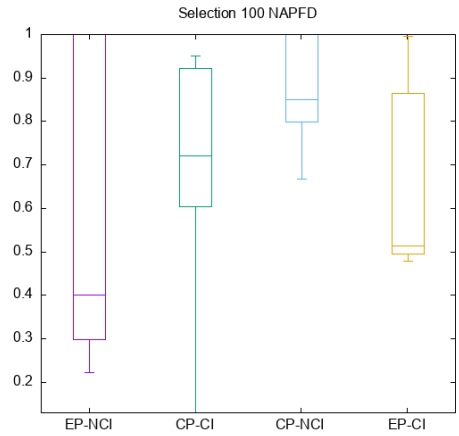

Figure 4.12: Distribution across the different datasets of NAPFD values for the LambdaMART algorithm using the `NDCG@20` training metric and 30 trees.

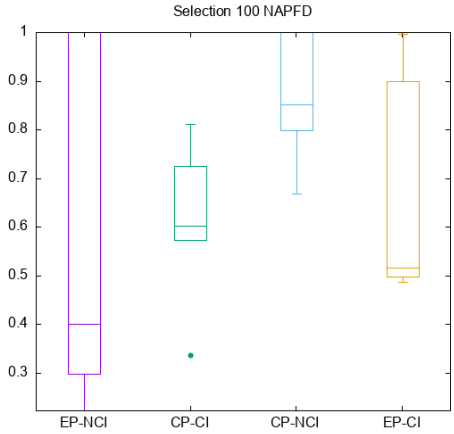

Figure 4.13: Distribution across the different datasets of NAPFD values for the LambdaMART algorithm using the `NDCG@20` training metric and 20 trees.

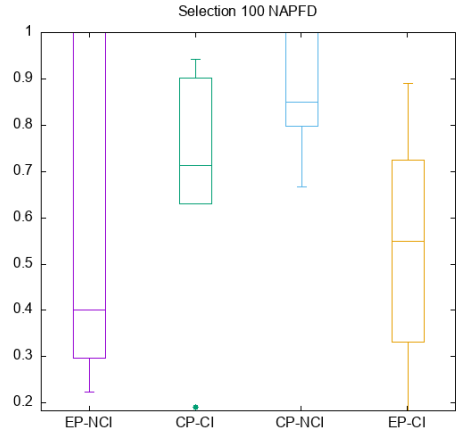

Figure 4.14: Distribution across the different datasets of NAPFD values for the LambdaMART algorithm using the `NDCG@30` training metric and 30 trees.

For such configuration, the resulting induced selection has similar values as the other algorithms: The resulting induced selections are:

- `S-SEL`: A safe selection was achieved with a selection size of 40%, corresponding to 9% of the execution time.
- `80-SEL`: A selection with average inclusiveness over 80% was achieved with a selection size of 40%, corresponding to 9% of the execution time.
- `50-SEL`: A selection with average inclusiveness over 50% was achieved with a selection size of 10%, corresponding to 2% of the execution time.

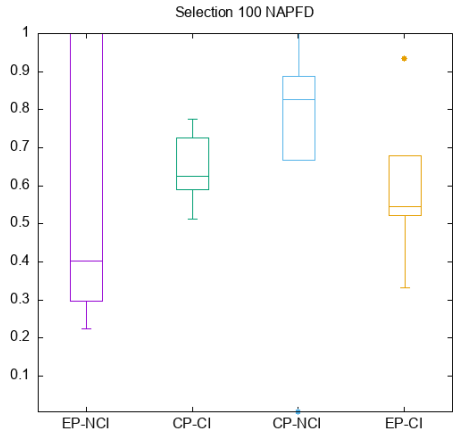

Figure 4.15: Distribution across the different datasets of NAPFD values for the LambdaMART algorithm using the `NDCG@30` training metric and 20 trees.

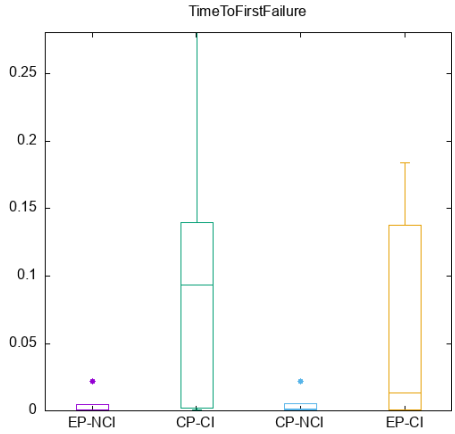

Figure 4.16: Distribution across the different datasets of *time to the first failure* values for the LambdaMART algorithm using the `DCG@10` training metric and 20 trees.

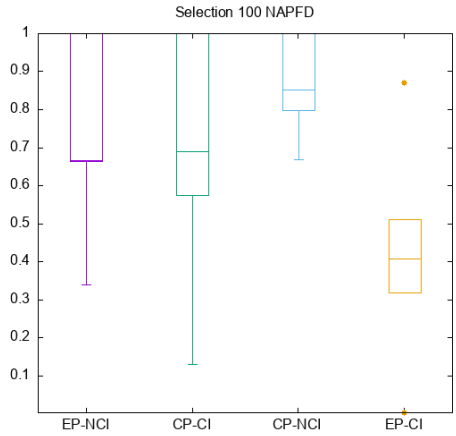

Figure 4.17: Distribution across the different datasets of NAPFD values for the MART algorithm using the `DCG@10` training metric and 30 trees.

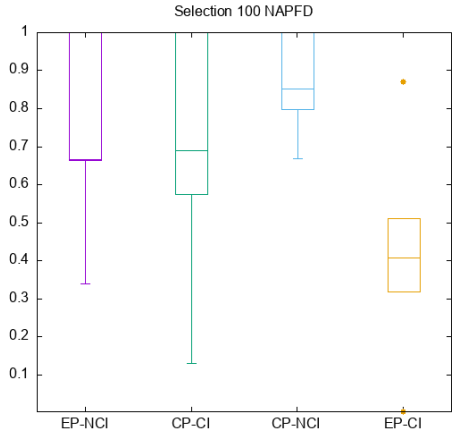

Figure 4.18: Distribution across the different datasets of NAPFD values for the MART algorithm using the `MAP` training metric and 30 trees.

4.6 Other algorithms considered

We performed training of AdaRank, with 500 rounds. As with other approaches, we considered different training metrics. However, none of the resulting models converged for our dataset.

Training of RankBoost was also attempted. However, as explained in section 2.2.4, it requires at each stage to keep a distribution D of memory complexity $O(n^2)$ for n the number of tests on each CI job. With the dataset used, such memory complexity exceeded the memory capacity of even the HPC nodes where the training was performed.

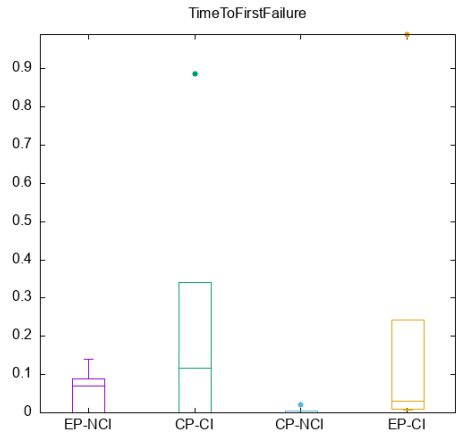

Figure 4.19: Distribution across the different datasets of *time to the first failure* values for the MART algorithm using the `DCG@10` training metric and 30 trees.

Chapter 5

Conclusion

In this project, we created different test prioritization datasets with information collected from the CI pipeline of *Business Central*. These datasets included different features to represent changes to a codebase and criteria to prioritize tests. With these datasets, different ranking algorithms were trained. For the validation dataset, the proposed prioritizations were evaluated.

The infrastructure to collect CI history information and related coverage information, as well as the extraction mechanism of features can be found in the accompanying repository of this project. Such tooling can be used to increase the strength and confidence of the evaluation here presented by adding more samples and extending the time from which the dataset is collected.

We also provide the ranking datasets used for this project in the accompanying repository. In contrast to existing datasets for the prioritization problem, we include coverage information properties.

We proceed to discuss the different research questions stated in chapter 1.

5.1 RQ1: Which of the ranking techniques yields the best prioritization results?

From the evaluated algorithms, we found that the best performing and most consistent approach was Coordinate Ascent with the training metric `NDCG@30`.

For this algorithm, across the different datasets considered, the average NAPFD value obtained was 82%. For the `CP-NCI` dataset, the average NAPFD value obtained was 86%.

5.2 RQ2: How much could we reduce the number of tests and their execution time?

For the best configurations of Coordinate Ascent described for **RQ1**, the induced *safe* selection was 40% the size of the complete test suite. This means that for this dataset, the CI pipeline would execute 60% fewer tests.

The corresponding execution time for the proposed selected tests is 10% of the execution time of the complete test suite.

5.3 RQ3: What is the effect of using coverage information for *Learning to Rank* techniques?

In contrast to existing *Learning to Rank* approaches to the problem, we use coverage information. We found that the best-performing dataset was `CP-NCI`, which uses coverage for prioritizing the training dataset, but does not use it as part of the features that identify each test.

`CP-NCI` consistently achieved better results in every algorithm and configuration. This suggests that for this pipeline there exists a correlation between a good prioritization in regards to test prioritization metrics and a prioritization that executes lines covered by the changes first.

However, adding coverage information as part of each test increases the dimensionality. Increasing the dimensions of the input space requires more training samples to obtain reliable results. This could explain why `CP-CI` performed worse than `CP-NCI`.

5.4 Threats to validity

Although the results are promising, it is important to consider the threats to the validity of the evaluation. The most relevant threat is a biased dataset.

As explained in section 4.1, the dataset was collected over a week. Also, as previously explained, the size of the collected dataset and test suite restricted the training of the ranking algorithms.

Among some of the reasons this could have biased the evaluation, is that a week is a short time to be representative of the overall test execution dynamics.

For example, a developer working on a feature may re-run the pipeline and continuously fail the same set of related tests. Given that history features are used, the learning algorithms could use this as an indicator of a high priority test. Which may not be true after the work is done. In an online implementation of these techniques, this problem could be mitigated by periodic retraining of the model.

Chapter 6

Future Work

6.1 Usage of a prioritization technique in the context of the existing system.

This work focused on an offline evaluation of the techniques. For an online implementation, other technical challenges and practical considerations remain.

So far in our discussion, we did not tie how proposing a prioritization with these techniques relates to how they are executed by the DME system explained in section 2.1.1.

In this section, we outline the required changes, and possible strategies to use such rankings.

Recall that the DME system, executes tasks from a given job, by traversing each of the required dependencies defined by the model of the job. A single task may execute multiple test codeunits or test solutions.

Given a ranking proposed by these techniques, we can use the induced selection algorithm to reduce the number of test codeunits that each of these tasks executes.

Furthermore, if the selection results in *job tasks* with no test codeunits to execute, we can remove this task, along with the dependencies that were only required by this task. By doing so recursively we can remove entire paths of the job's execution.

For clarity, see figure 6.1, where a DAG representing the dependencies of a job's execution is given. Filled with black are the test tasks that after the selection algorithm, had no tests to execute. Filled with gray are the tasks that were only required for such tests, and therefore could be removed.

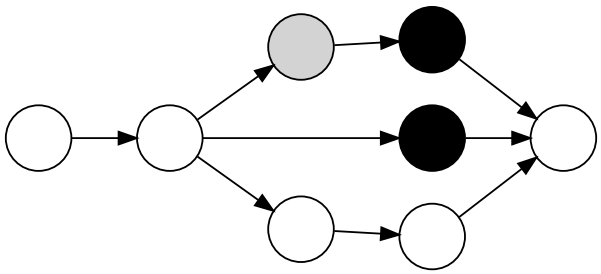

Figure 6.1: Tasks and predecessors removed from the job model.

This would be an effective use of the selection proposed, however, engineering is required to allow for such dynamic changes in a job's execution.

Since test tasks can be performed in parallel by the DME system, a proposed prioritization of all the tests in the job can not be executed as given. Instead, the prioritization can be used locally on each task. For a given test task and the overall prioritization, one can get a local prioritization for the tests belonging to such task. However, the engineering that allows dynamically sorting the test codeunits in a single task is also missing.

Finally, the current build system allows for assigning priorities to the tasks to run, which is taken into consideration when deciding which task to execute next from the set of tasks with completed dependencies. This could be dynamically assigned based on the prioritization of the tests being run.

The main engineering challenge to allow for the usage of these proposals is to make tests *first-class citizens* of the data model proposed by the DME system. Currently, the DME system has no knowledge of the tests being run by a task, as it is instead the responsibility of the task's implementation.

6.2 Tackling CI optimization from other angles

Business Central has several CI pipelines that constitute each of the different development cycles, from different areas that the product has. We studied a single optimization strategy for one of these CI pipelines, namely Test Selection and Prioritization.

However, different strategies exist in the literature, for instance, the usage of test suite minimization to remove superfluous tests from a codebase.

Another approach could be to have learning models to predict the task failure of each of the tasks a job is comprised from. A similar feature vector as obtained for this project representing the change could be part of the training data used to train binary classifiers.

The approach taken on this work, to obtain a meaningful representation of the changes done to a codebase can be further leveraged for future analysis of this CI pipeline.

6.3 Further evaluation of test prioritization techniques

Reducing the number of tests considered for each job execution for our training dataset would allow for a larger timespan to be considered in our evaluation. As each job would become more manageable for the learning algorithms and feature extraction infrastructure.

For instance, in this project, we were not able to perform training of RankBoost, as its memory complexity is squared in the number of tests.

This would also increase the confidence and validity of the results presented in this project. Having a larger dataset allows for multiple validation datasets, to increase the confidence of the proposed selection sizes in this work.

Other approaches that avoid retraining like Reinforcement Learning could be leveraged and be the subject of future work with the produced dataset, although comparison in existing literature favors *Learning to Rank* approaches [2].

In regards to *Learning to Rank* approaches, more properties could be obtained to give a vector representation of a codebase change and tests. For instance, in [2], Bertolino, et.

al. propose more code metrics that were not in scope for this project. Examples include cyclomatic complexity, number of public or private methods, number of new functions, and others. Other interesting features were explored by Busjaeger et. al. in [5], where besides coverage score, they use *text similarity* metrics between the names of the test procedures and paths of the files or their content.

Adding more useful features to this dataset would allow for more robust criteria for the learning algorithms.

Bibliography

[1] Luciano Baresi and Mauro Pezzè. An introduction to software testing. *Electronic Notes in Theoretical Computer Science*, 148:89–111, 02 2006.

[2] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration. *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1–12, 2020.

[3] Árpád Beszédes, Tamás Gergely, Lajos Schrettner, Judit Jász, Laszlo Lango, and Tibor Gyimóthy. Code coverage-based regression test selection and prioritization in webkit. *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 46–55, 2012.

[4] Christopher Burges. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11, 01 2010.

[5] Benjamin Busjaeger and Tao Xie. Learning for test prioritization: an industrial case study. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[6] Olivier Chapelle, Donald Metlzer, Ya Zhang, and Pierre Grinspan. Expected reciprocal rank for graded relevance. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, page 621–630, New York, NY, USA, 2009. Association for Computing Machinery.

[7] S. Elbaum, Alexey Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *Software Engineering, IEEE Transactions on*, 28:159–182, 03 2002.

[8] Yoav Freund, Raj Iyer, Robert E. Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *J. Mach. Learn. Res.*, 4(null):933–969, dec 2003.

[9] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2001.

[10] Jerome H. Friedman and Jacqueline J. Meulman. Multiple additive regression trees with application in epidemiology. *Statistics in Medicine*, 22, 2003.

[11] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, oct 2002.

[12] Donald Metzler and W. Bruce Croft. Linear feature-based models for information retrieval. *Information Retrieval*, 10:257–274, 2006.

[13] Rong Pan, Mojtaba Bagherzadeh, Taher Ahmed Ghaleb, and Lionel Claude Briand. Test case selection and prioritization using machine learning: A systematic literature review. *ArXiv*, abs/2106.13891, 2021.

[14] John Rooksby, Mark Rouncefield, and Ian Sommerville. Testing in the wild: The social and organisational dimensions of real world practice. *Comput. Supported Coop. Work*, 18(5–6):559–580, dec 2009.

[15] G. Rothermel and M.J. Harrold. A safe, efficient algorithm for regression test selection. In *1993 Conference on Software Maintenance*, pages 358–367, 1993.

[16] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[17] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. Reinforcement learning for automatic test case prioritization and selection in continuous integration. *CoRR*, abs/1811.04122, 2018.

[18] Filippos I. Vokolos and Phyllis G. Frankl. Pythia: a regression test selection tool based on textual differencing. 1997.

[19] Jun Xu and Hang Li. Adarank: a boosting algorithm for information retrieval. pages 391–398, 07 2007.

[20] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Softw. Test. Verification Reliab.*, 22:67–120, 2012.

[21] Mu Zhu. Recall, precision and average precision. 09 2004.