Optimizing a Continuous Integration Pipeline of *Business Central* using Test Selection and Prioritization Techniques

Diego Joshua Martínez Pineda

Advisor: Mahsa Varshosaz
Submitted: May 2022

IT UNIVERSITY OF COPENHAGEN

Acknowledgments

`SYN+ACK`[1]

[1]A TCP joke, because less people would get a UDP one.

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 *Business Central* project.

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

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 benefits the development cycles, but overall development costs and energy impact.

Our results show that using coverage information for assigning priorities increases the effectiveness of the prioritization proposed. However, using coverage information as part of the feature vectors reduces its effectiveness and increases the variance of the results for the datasets created.

Based on these results, we argue that with the proposed prioritization algorithms we would be able to reduce the amount 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.

We also give an outline on how this prioritization could be integrated 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, that the software remains being correct. To tackle this complexity problem, many strategies and different angles have been studied and implemented by large-scale software systems.

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

It consists on developers writing automated tests: code that executes scenarios that a user would experience and asserts that desired properties hold true. 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. However, there is a drawback, as the system evolves, more scenarios get added and eventually it can become time-consuming to run the complete set of automated tests.

Another challenge in software engineering consists in having an effective development process when working with large teams. Many strategies have also been studied, and modern 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 prior to 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.

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

We can now see that having a time-consuming set of tests to execute is a relevant drawback for teams that want to adopt CI practices. While on one hand, it is desired to automatically run tests with every small change, a sufficiently large test suite can take a considerable amount of time to execute.

To tackle this problem, academic research and industry has focused on several angles. Algorithms on Test Selection and Prioritization (TSP) are techniques that aim on automatically proposing a relevant sorted subset of tests to execute. These techniques acknowledge that it is not needed to run all the tests, instead, only those which would be more likely to fail first. As the goal of running a test suite is to avoid potential errors to be integrated to the codebase, we can give the developer feedback on errors to be corrected earlier, if found earlier.

For this project, TSP techniques are proposed and evaluated for 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 the product and to propose possible directions the product could take to improve the CI cycle feedback time and effectiveness.

Not only is optimizing CI pipelines relevant to improve agility on software development processes, but it also has a positive impact in 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.

In this project, we will follow the same approach as previous research by Bertolino et.al. in [1], transforming this problem to a *ranking* problem. Afterwards, different *Learning to Rank* algorithms will be applied, and evaluated. However, in contrast to their work, we use test coverage information

for two different parameters of how our ranking datasets are created.

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.

Chapter 2

Background

2.1 Business Central

Business Central(BC) is an Enterprise Resource Planning (ERP) software system targeting Small and Midsized Businesses from Microsoft. It's functionality spans several areas of a company like Finance, Sales, Warehouse Management, among 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. Furthermore, 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 com-

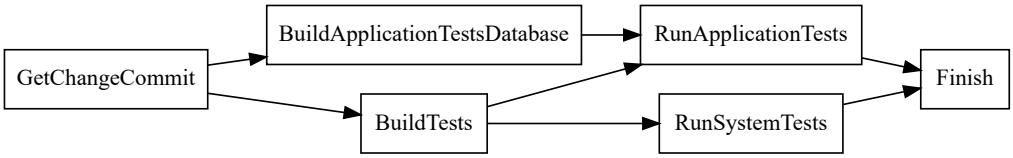

Figure 2.1: Example of tasks that depend on each other for a job execution represented as a DAG.

plete 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 correspond 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 to the application developers a complete control over job's execution. A Pull Request triggering the CI pipeline is not the only way a developer can request *jobs*. They can also require them whenever on 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 by giving an example: Suppose a *job* being executed is defined by a model consisting of the tasks shown in figure 2.1 as the underlying DAG for such 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`. Lets suppose that `BuildApplicationTestsDatabase` takes longer, when the task `BuildTests` is done the task `RunSystemTests` can already begin as it's only dependency has finished. However `RunApplicationTests` will have to wait for it's 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 *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 underlying such execution job.[1]

2.1.2 Test Selection currently on BC *application* tests

In the current CI pipeline of the *application* code of the product there is a selection method based on test coverage information of each test execution. 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 some meaningful way. In this section, we review how are tests defined for *application* code, and what kind of coverage information we have available when executing tests.

Application Tests in AL

We first give an overview of how tests are defined in AL. For a more thorough overview of the AL language and its' role on the *Business Central* system, we refer to appendix A.

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 comprised of 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 tests procedures defined under a test

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

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

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

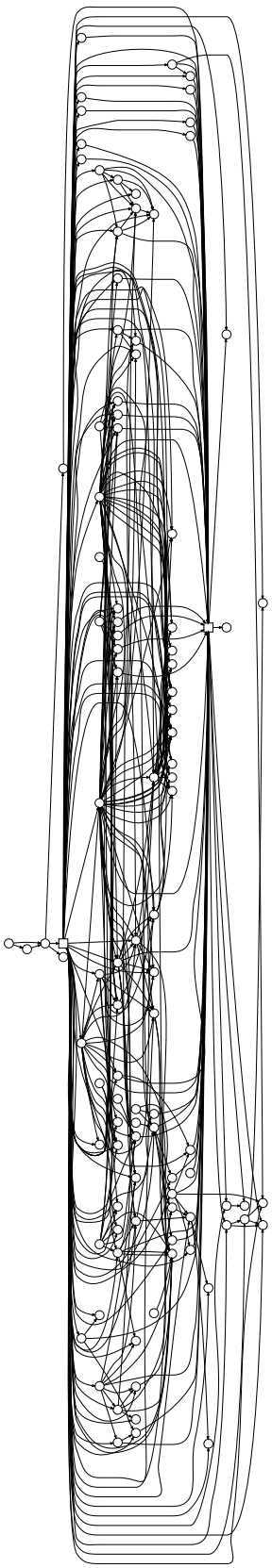

Figure 2.2: Underlying DAG of the metamodel of job executions of the *Application* pipeline for the *Business Central* product.

```
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.

codeunit. In 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 should the test runner 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, result of historical legacy. As means of identification, we identify 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 these test runners have, for our purposes we highlight a significant difference: The *CAL Test Runner* can produce coverage information of tests execution, and the *AL Test Runner* currently in production does not.

Coverage information

AL has implemented a primitive of the language that can be used to record information about which lines of code of which objects were executed. In

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

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 3 different kind of outputs relating to coverage information. We will use information that will allow us to: given a test codeunit executed by the test runner, to have a list of the different *AL objects* and lines within these objects that were ran by the test execution.

For concreteness and further illustrate the information available 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. Such coverage 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 such 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.

We can see in this example that Codeunit 28 had among others lines 111, 112, 115 and 116 executed. In figure 2.5 we can see such lines.

```
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.

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 in obtaining the lines and files which the developer is affecting when integrating it'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 the current technique as a partial selection based on coverage traces.

As we will explain in section 3.2.1, 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

Test Selection and Prioritization techniques have a large body of knowledge, result of extensive focus from both academy 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. Obtain a subset of tests, such that the capacity of fault detection of the test suite is not lost, i. e. if for such change, the test suite execution results on a failure, this subset of tests should fail as well. A stronger condition is given by a *safe* selection algorithm, which requires that every failing test case for a given change to be included in the resulting 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 failing first.

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

2.2.1 Related work

Yoo and Harman in their survey [16] present a detailed overview of techniques in the problems of regression test selection, prioritization and minimization[4]. In this survey, several foundational works on this area are presented, formal definitions, and metrics to evaluate this problem.

It is also worth noting, similar techniques found in the literature to the already existing selection technique in *Business Central*. As explained in section 2.1.2 the CI pipeline of interest of the product collects coverage information for some subset of test tasks. Using the information of which files were changed by the developer, and the coverage information for each test *codeunit*, a selection is proposed.

This intuitive approach aims to select *modification-traversing* tests [12], this was one of the first approaches studied by Rothermel, and Harrold,

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

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[11], others use execution traces (Vokolos and Frankl in [14]) and some others use coverage information. Like the work of Beszédes et. al. [2], where they describe populating a coverage database for the C++ components of the WebKit project, and identify the changed modules from a given revision.

As a starting point for this project, we reviewed the survey by Pan, et. al. [10] as such survey focused specifically on Selection and Prioritization and techniques using Machine Learning (ML). We initially aimed to find approaches using Reinforcement Learning, as online learning could be beneficial for this use case. In this area, we find the case of Speiker et. al. in [13], where a reinforcement learning agent is proposed using only history on failure and duration of previous iterations.

This leads us to a publication which was the main inspiration and outlined our approach: the proposal by Bertolino et. al. in [1], where they do a more thorough comparison of the effectiveness of Reinforcement Learning approaches, and approaches using ranking algorithms.

Another similar approach is given by Busjaeger and Xie in [4], where they evaluate applying learning algorithms in the case of Salesforce. We highlight that for the features representing the changes a developer made, they use coverage information. In particular the idea of coverage score, to reduce the impact of outdated coverage information which is a reality in large scale environments such as *Business Central*.

2.2.2 Ranking algorithms

As explained in section 2.2.1, 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, algorithms proposed to tackle them. In particular, the techniques we will focus on for 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 them for a given information need (query). Ranking algorithms are relevant in different problems. Examples of systems using them are search engines, or recommender systems.

The ranking problem has been extensively studied as it is fundamental for dealing with the information overload that working with computer systems creates.

An approach that has been subject of extensive research in recent years is *Learning to rank*, a set of ML techniques whose goal is to obtain a *ranking model* out of training data. This model is reused when new queries to the system are given, with the goal of ranking new unseen queries in a similar way as 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 the developer wants to commit to the target branch. The set of documents D corresponds to the complete test suite of the product. Our relevance criterion corresponds to sorting the failing tests first (if any) and the rest of the tests can be sorted through a different criteria like duration or test coverage as we will explain in section 3.4.

With this interpretation of the Test Prioritization problem, there is still freedom of interpretation in two aspects: the representation of the query q for a given codebase change, and the relevance criterion to use. These will be subject of our experiments as explained in section 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.

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. Note that these metrics in our context are used for training and not for evaluating the proposed rankings. This is because it is more meaningful to evaluate with metrics specific to the desired features of the Test Prioritization problem. We will further expand on metrics used for evaluation in section 2.2.3.

The relationship of metrics used for training and its effect on Test Prioritization metrics is also subject of our experiments as it will be described in chapter 3.

A common metric used to evaluate ranking algorithms is the Discounted

Cumulative Gain (DCG). `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. As we see, this metric increases when the first values are given a high relevance as expected. In contrast, high priority 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`, to do so it compares against the ideal ranking for that query and computes its corresponding `DCG@k`, lets call it $IDCG@k$:

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

`ERR@k`

`MAP` The Mean Average Precision (MAP) is based on binary classification metrics. Traditionally precision and recall are widely used for binary classification. In the context of Information Retrieval, *precision* refers to how many documents marked as relevant are relevant by our prediction, *recall* refers to how many of the relevant documents were retrieved from all the relevant documents in the query.

The average precision (AveP) represent 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 (see section 3.4), they can be considered binary by giving a cutoff point for the assigned relevance.

These are the metrics which were subject of change in the training of the different ranking algorithms. We will now briefly explain the different ranking algorithms explored.

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-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 [9], and it has successfully been applied in 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.

Among other things that differ from the traditional optimization, in [9] they propose other constrains to the scoring functions and transformations to reduce the parameter space being optimized to find values on a simplex.

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:

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

For free parameters to optimize Ω, the feature vector $f(D, Q)$ and a constant Z. The free parameters are positive values such that they sum up to 1. In the ranking library used for the implementation in this project, Ranklib, Z is set to zero.

Notice that this scoring function is not the function being optimized, but the ranking evaluation metric used which will be varied on our experiments.

MART

Regression algorithms using gradient boosting were first proposed by Friedman in [7], Multiple Additive Regression Trees (MART) is a gradient boosting technique further developed by Friedman and Meulman in [8].

Again, we give a very shallow explanation, as a general introduction. We refer to the reader to the articles cited above and Machine Learning literature for a more complete 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 h_i (called weak learners):

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

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 for such model that minimizes the loss function.

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

For *pointwise* ranking algorithms, regression algorithms such as MART can be used to rank by regressing a relevance function for each of the documents to rank, minimizing some loss function. In our case, the loss functions are the different training metrics presented in section 2.2.2.

LambdaMART

LambdaMART takes it's name from its constituents: LambdaRank and MART. In the previous section we explained how MART works for general regression and for ranking.

In contrast to *pointwise* algorithms that used a relevance function to produce a ranking, *pairwise* algorithms like LambdaMART aim to produce a comparison function between documents.

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.

In order 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:

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

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 [3] for a more thorough explanation of how this gradients are reduced to scalars subject to learning models.

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 [1] on Test Prioritization.

RankBoost

First proposed by Freund, Yoav, et. al. in [6]. As MART, it uses *boosting*, i. e. 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 are the documents on a query. For this reason, this method is $O(|X|^2)$, which can restrict the applicability of this algorithm.

Each learner updates the distribution D, emphasizing the pairs that are more relevant for the algorithm to properly order. The final relevance function then becomes a linear combination of each of these learners. We refer to the cited article for more details and further reading.

AdaRank

First proposed in [15] by Xu and Li, 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 next rounds.

The weak learners they propose are linear combinations of the metric to minimize and the weight distribution.

2.2.3 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 the time taken to execute the complete test suite.

Inclusiveness

In test selection, we do not only rely on 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 take this into consideration, 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 [11] always has $i = 1$. As every fault revealing test is included in the selection.

Selection size

On the other hand, a 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 for how big the selection is, *selection size* was proposed:

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

A good selection algorithm strives for having a small selection size, while high inclusiveness.

Time to 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, first proposed by Elbaum, Malishevsky, and Rothermel in [5] is the Average of the Percentage of Faults Detected (APFD).

Normalizing this metric with respect to the amount of failures detected allows to take into consideration 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:

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

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

It represents the proportion of the tests 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

3.1 Outline

In the existing research, datasets have been constructed focusing on test run properties and results. For instance, Spieker in **??** worked on ??. And Bertolino et. al. in **??** evaluate their algorithms by constructing a dataset from the Apache Commons projects commit history.

In section 3.2, we describe how we obtained the information from the *Business Central* build system, along with the required changes to collect coverage information.

In section 3.3, we describe how we obtain features for the collected changes and test runs. These features are used to numerically represent a test run in a CI job, for them to be input of the ranking algorithms.

Afterwards, in section 3.4, we describe how we label the training dataset. That is, the different criteria used to assign priorities to the given tests.

Finally, in section 3.5, we describe the training of the different ranking models, with different datasets.

3.2 Collecting the dataset of CI executions

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

For each CI job, the information extracted was:

- Execution model with the tasks that the DME 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 metainformation to identify the job in the VCS.
- For each of the *application test* tasks, the result of each procedure run for each of the test units considered, along with information on duration of it's execution.
- Comparison with 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 with respect to different properties of the codebase for each test. Along with data to evaluate the prioritized tests.

3.2.1 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 ran with the *CAL test runner* do have coverage information. However that is not the case for tests that use the *AL test runner*.

A full 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 to 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 **??**. This is partly true in our case as well, however, we acknowledge that the information given by coverage can be valuable for our problem.

In **??** a more robust approach to use coverage information is proposed, by defining a coverage score. A feature like coverage score, mitigates for the lack of accuracy of the coverage information. As an additional mitigation

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

to this problem we introduce windows to compute such coverage scores as it will be shown in 3.3.

3.2.2 The collection process

As a general overview, over roughly a week period, real CI jobs in this pipeline were collected. And sporadically between these jobs, custom jobs were executed with the required changes to the *AL test runner* to collect coverage information.

For the results presented in this work, two of these coverage collecting jobs were executed and retrieved, and 172 CI jobs were collected.

Whenever coverage information is required to compute the features of a given CI job, the coverage information used will be the closest earlier collected one.

The scale of the collected information limited the amount of jobs we were able to collect.[2]

3.3 Characterizing Test Executions for a Codebase Change

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.

In this section we explain the different kinds of information obtained to characterize them.

3.3.1 File changes

We use the following quantities to represent the changes done for each object:

- Amount of new AL tables.
- Amount of new AL objects.
- Amount of modified AL objects.
- Amount of removed AL objects.

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

- Amount of changed tests.
- Amount of non AL file changes.
- Amount of added lines to AL objects.
- Amount of removed lines from AL objects.

3.3.2 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, proportion of times the test has failed.
- Average duration of this test within the available data.

3.3.3 Coverage properties

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

- The proportion of lines covered by that test in relation to the average.
- The amount of files changed that were covered by that test.
- The amount 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 windows sizes.

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

3.4 Prioritizing Test Executions in a CI job

As part of our training dataset for the ranking algorithms, we need to associate a ranking to each of the tests executed in a job. This ranking is what 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 the results will be evaluated, as this would be biased. The evaluation will be given only by the TSP metrics presented in section 2.2.3.

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 outline two different approaches taken to assign priority functions.

3.4.1 Failure and Duration decreasing exponential priority

In [1] they 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, and 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. We also notice something else, by this being an exponentially decreasing function, changes in duration have a larger effect for tests with small duration, than test executions with large duration.

3.4.2 Coverage discrete priorities

We propose a discrete priority function using coverage information.

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

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 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 $L_i < \mu_{L,\tau}$

It is worth reminding the reader, that from such given job we do not have the exact coverage information but the most recent, previously collected. As we described in section 3.2.

We created training datasets with both of these priority functions.

3.5 Training ranking models

We performed training of the different described ranking algorithms for each of the training datasets created. For implementation, we used Ranklib 2.17. The training was performed with the High Performance Cluster from the IT University of Copenhagen.

For each training performed, we varied some of the available hyperparameters and also varied with the training metrics presented in 2.2.2.

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

We varied the results to get the best performing configuration for each approach as it will be presented in chapter 4.

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.

Dataset name	**Prioritization**	**Features of each test**
EP-NCI	Decreasing exponential as in [1]	Every feature described in section 3.3 except coverage related
EP-CI	Decreasing exponential as in [1]	Every feature described in section 3.3
CP-NCI	Coverage based as described in section 3.4.2	Every feature described in section 3.3 except coverage related
CP-CI	Coverage based as described in section 3.4.2	Every feature described in section 3.3

Table 3.1: Naming of the different datasets trained.

Each of these datasets 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 queries contained around 18000 tests.

Chapter 4

Results and Evaluation

First we use the NAPFD metric as defined in section 2.2.3 to compare the effectiveness of the rankings for the Test Prioritization problem.

In this section, we present for each ranking algorithm used, how the different criteria used influences the NAPFD behavior. To emphasize, the varying criteria on the experiments were:

- Priority function used on datasets.
- Usage of coverage information
- Training metric used for ranking algorithms.
- For MART and LambdaMART, the number of trees used.

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, we induce selection algorithms by taking the first proposed results. We take the first proposed results based on the following criteria:

- An strictly safe selection `S-SEL`: we take the first results such that for all the evaluated jobs, every test failure is included.
- An above 80% average selection `80-SEL`: the first results such that the average of its *inclusiveness* is at least 80%.
- An above 50% average selection `50-SEL`: likewise, we include tests until the average inclusiveness is at least 50%.

We obtain these selections by iterating in increments of 10%.

For a complete set of values shown in the distribution plots and more evaluation metrics, see appendix C.

4.1 Coordinate Ascent

Using different training metrics did not have a large impact in 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 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%.

Figures 4.1 to 4.1 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 first failure* in figure 4.1.

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.

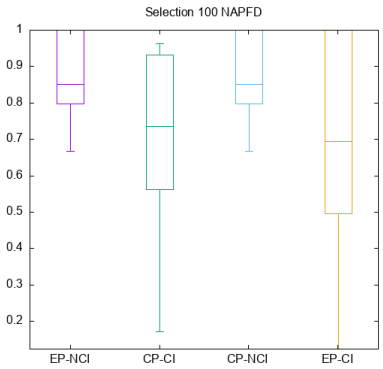

Figure 4.1: Distribution across the different datasets of NAPFD values for the Coordinate Ascent algorithm using the DCG@10 training metric.

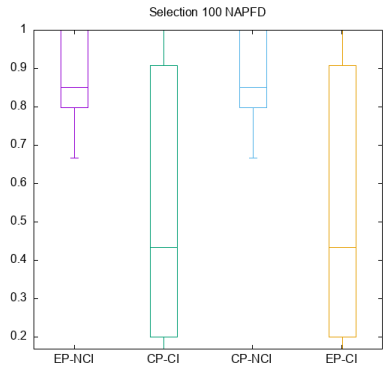

Figure 4.2: Distribution across the different datasets of NAPFD values for the Coordinate Ascent algorithm using the MAP training metric.

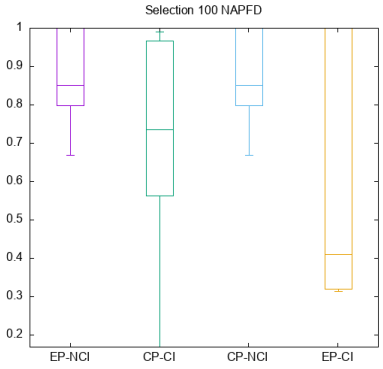

Figure 4.3: Distribution across the different datasets of NAPFD values for the Coordinate Ascent algorithm using the NDCG@10 training metric.

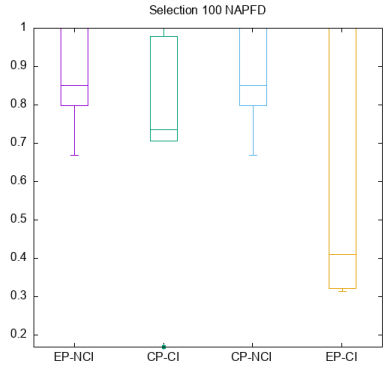

Figure 4.4: Distribution across the different datasets of NAPFD values for the Coordinate Ascent algorithm using the NDCG@20 training metric.

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

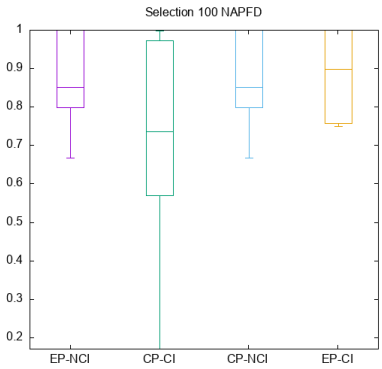

Figure 4.5: Distribution across the different datasets of NAPFD values for the Coordinate Ascent algorithm using the `NDCG@30` training metric.

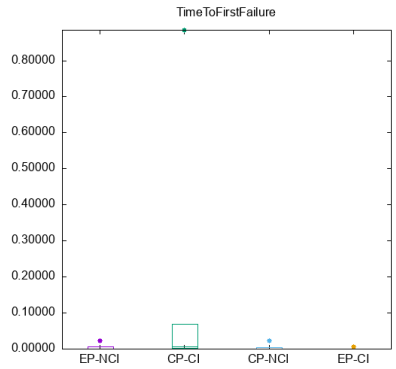

Figure 4.6: Distribution across the different datasets of times to first failure for the Coordinate Ascent algorithm using the `NDCG@30` training metric.

4.2 LambdaMART

For LambdaMART, using the `MAP` metric, resulted on consistently ranking with an NAPFD value of 66% for any of the different trained datasets, as it can be seen in figure 4.2. With the `ERR@10` metric, the training did not converge, resulting on 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 trees.

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 4.2 we can see the comparison of distributions of *time to first failure*, for the metric `DCG@10` and 20 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.

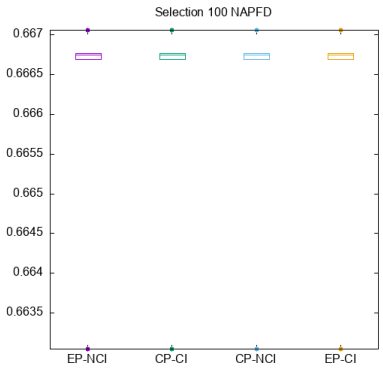

Figure 4.7: Distribution across the different datasets of NAPFD values for the LambdaMART algorithm using the `MAP` training metric and 30 trees.

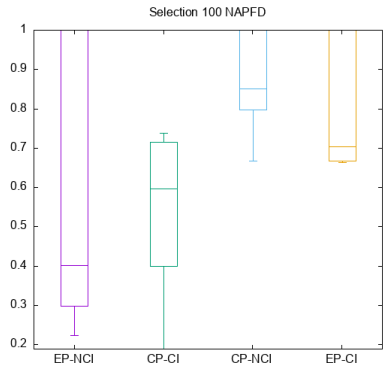

Figure 4.8: Distribution across the different datasets of NAPFD values for the LambdaMART algorithm using the `NDCG@10` training metric and 30 trees.

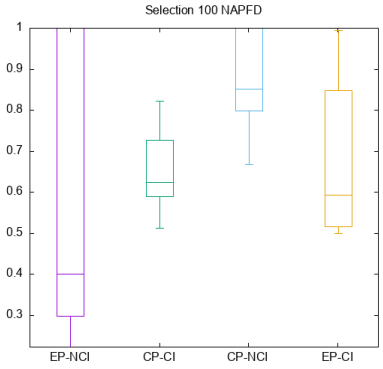

Figure 4.9: Distribution across the different datasets of NAPFD values for the LambdaMART algorithm using the `NDCG@10` training metric and 20 trees.

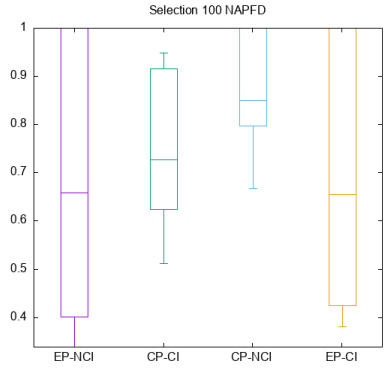

Figure 4.10: Distribution across the different datasets of NAPFD values for the LambdaMART algorithm using the `DCG@10` training metric and 30 trees.

- `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.

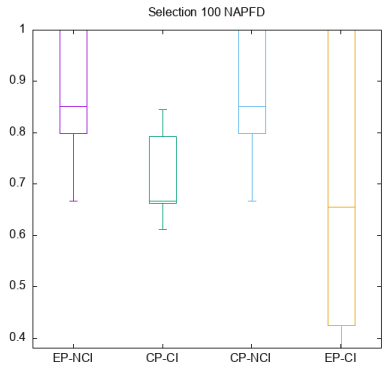

Figure 4.11: Distribution across the different datasets of NAPFD values for the LambdaMART algorithm using the DCG@10 training metric and 20 trees.

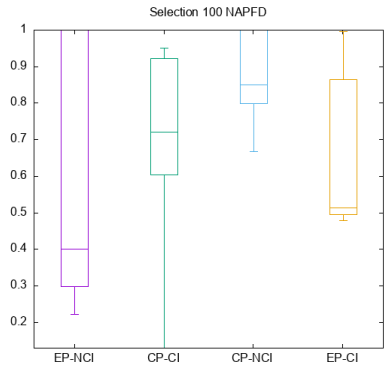

Figure 4.12: Distribution across the different datasets of NAPFD values for the LambdaMART algorithm using the NDCG@20 training metric and 30 trees.

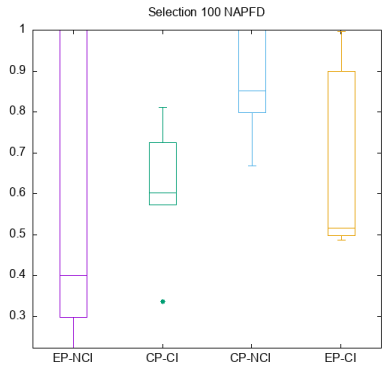

Figure 4.13: Distribution across the different datasets of NAPFD values for the LambdaMART algorithm using the NDCG@20 training metric and 20 trees.

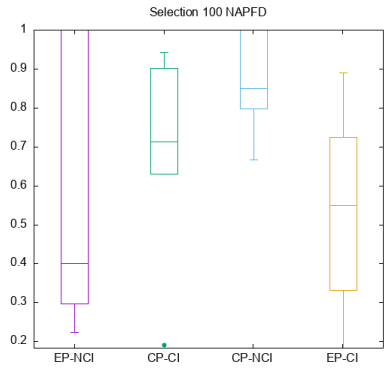

Figure 4.14: Distribution across the different datasets of NAPFD values for the LambdaMART algorithm using the NDCG@30 training metric and 30 trees.

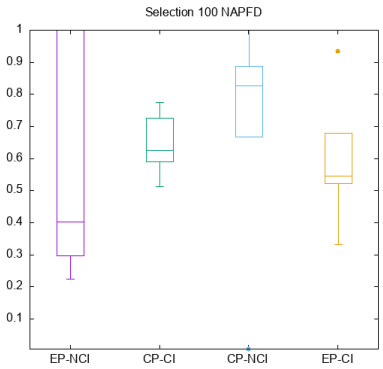

Figure 4.15: Distribution across the different datasets of NAPFD values for the LambdaMART algorithm using the `NDCG@30` training metric and 20 trees.

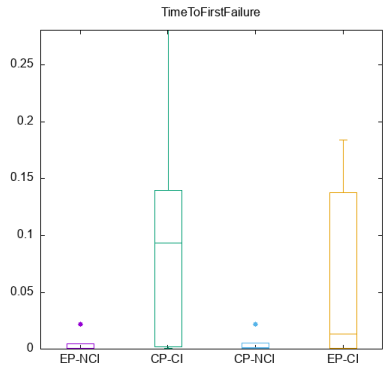

Figure 4.16: Distribution across the different datasets of *time to first failure* values for the LambdaMART algorithm using the `DCG@10` training metric and 20 trees.

4.3 MART

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

As in 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 4.3 a distribution of the *time to first failure* values. The best configurations achieved an average NAPFD value of 67%.

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.

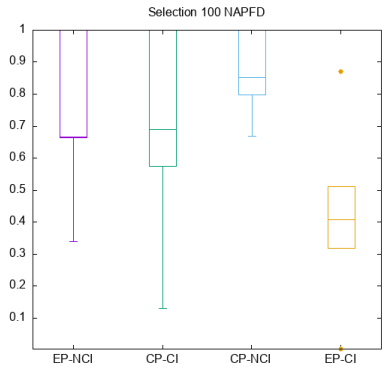

Figure 4.17: Distribution across the different datasets of NAPFD values for the MART algorithm using the DCG@10 training metric and 30 trees.

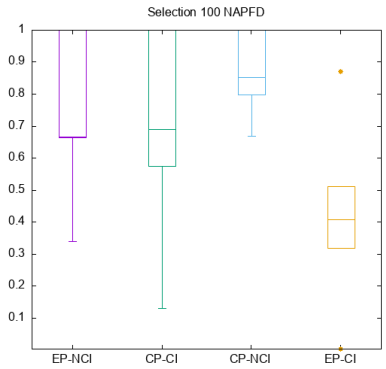

Figure 4.18: Distribution across the different datasets of NAPFD values for the MART algorithm using the MAP training metric and 30 trees.

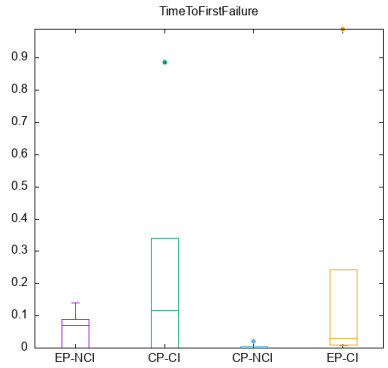

Figure 4.19: Distribution across the different datasets of *time to first failure* values for the MART algorithm using the DCG@10 training metric and 30 trees.

4.4 Other algorithms

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

Training of Rankboost was also attempted. However, as explained in section 2.2.2, 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. Further work on tuning this algorithm include appropriately reducing the amount of tests considered in each job execution for the training dataset.

Chapter 5

Conclusion

5.1 Contributions

5.1.1 Evaluation of the ranking algorithms.

Results for the best configurations of each algorithm induce promising selection sizes. The algorithm with most consistent behavior was Coordinate Ascent optimizing the `NDCG@30` metric.

Inducing a safe selection on our dataset yielded a selection size of 40%. This implies that the amount of tests executed could be reduced by more than half. The corresponding execution time for such selection was in average 10% the total execution time from executing all the tests.

While results were promising, it is relevant to note the threats to validity in this project described in section 5.2.

5.1.2 Using coverage information in *Learning to Rank* approaches to the Test Prioritization problem

In literature, coverage information in *Learning to Rank* approaches to test prioritization is not often used. With the distinction of Busjaeger in [4], studying an industrial scenario. There is no available dataset that includes such information.

The goal of comparing different configurations varying the information provided was to understand the effect of using coverage information.

The best performing dataset across algorithms and configurations consistently was the one that used the proposed discrete relevance function

given in 3.4.2 that uses coverage information. But that does not use coverage information in the features of each test case.

For this pipeline, this seems to imply that coverage information is useful to determine the prioritization, but the dimension complexity of adding more properties related to coverage is not.

5.1.3 Infrastructure to produce *Learning to Rank* datasets for the *Business Central* pipeline.

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 thesis project.

Increasing the strength and confidence on the evaluation can be achieved by enhancing the dataset. This also allows for a more data-driven approach to do analysis on the performance on the CI pipeline.

5.2 Threats to validity

The most relevant threat to validity of our results is a biased dataset.

As explained in section 3.2, the dataset was collected over a week period. Also, as previously explained, the size of the collected dataset and test suite restricted how much data we could process.

Among some of the reasons that 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 tests. Given that history features are used, the learning algorithms could use this as indicator of a high priority test. Which may not be true after the work is done.

Another example, is the occasional test failure that went in to the main codebase, for instance by a failing test selection criteria. This results in subsequent CI jobs failing in such tests, and effectively blocking the pipeline and give invalid data points. On the period of time the dataset was collected this was not observed.

A possible mitigation is to continuously retrain the algorithm, for which exploring reinforcement learning approaches could be interesting.

Future work on a more robust evaluation would entail a large timespan collection of data, with an appropriate reduction of the dataset to be manageable by the learning algorithms and feature extraction infrastructure.

5.3 Future work

5.3.1 Usage of a prioritization technique in context of the existing sytem.

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 amount of test codeunits that each of these tasks execute.

Furthermore, if the selection results on *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. In general terms, this corresponds to the reachable vertices of these removed vertices that are not reachable by any other non removed vertex on the opposite graph.

For clarity, see figure 5.1, where a DAG representing the dependencies of a job 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.

This would be an effective use of the selection proposed, however engineering is required to allow for such dynamic changes of a job's execution. In particular, test codeunits are not *first-class citizens* on the data model proposed by the DME, as it is instead responsibility of the task's implementation.

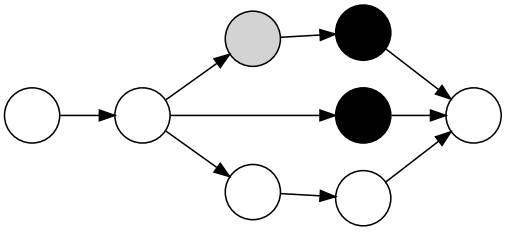

Figure 5.1: Tasks and predecessors removed from the job model.

For using the prioritization, other strategies can be leveraged. Since test tasks can be performed in parallel by the DME, 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. That is, 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 to allow dynamically sorting of the test codeunits by the test runners is missing to implement.

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 ran. Note that this would also require for the DME system to have knowledge of the tests being run by a task.

5.3.2 Tackling CI optimization from other angles

The product studied 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 these CI pipelines, namely Test Selection and Prioritization.

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

Another approach could be to have learning models to predict 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 a binary classifier, for which extensive research exists.

5.3.3 Further evaluation of prioritization techniques

Reducing the amount of tests considered for each job execution for our training dataset, would allow for a larger timespan to be considered by our evaluation.

This would increase the confidence and validity on the results presented in this project.

Appendix A

The AL Language

One of the main design aspects of *Business Central* (BC) is that all business logic is written in a custom Application Language (AL), hiding all implementation details dealing with the technology.

Extensions can be provided for the *Business Central* runtime to add business functionality and to fit customer's requirements as much as desired. Such extensions are not only written by Microsoft, as users can customize their experience via extensions as required.

This effectively creates an ecosystem of AL software, for diverse use cases of BC as an ERP.

Adding a language abstraction force abstraction layers to what is possible within the language and effectively separates business logic from technology implementation specifics.

This has allowed the product to evolve throughout the years and experience several changes that with a different design would be harder to achieve, for example: changing from database engine, or migrating to a cloud environment. Additionally, the language continues to evolve and be actively maintained by the team, to keep up with the requirements a modern language infrastructure requires.

Interestingly, this approach has been successfully used by other products as well. For example, another ERP: SAP, has the language ABAP. A custom Domain Specific Language (DSL) resembling COBOL to extend the functionality of the system.

In this section we give a brief tour of AL, *Business Central*'s custom DSL for modifying its runtime. For a complete reference, we refer to Microsoft's documentation [?].

```
var
    myInt: Integer;
    isValid: Boolean;
```

Figure A.1: Variable declaration in AL.

```
Amount := Total * myInt;
```

Figure A.2: Assignment and operations in AL.

```
if x = y then begin
    x := a;
    y := b;
end else
    y := b;
```

Figure A.3: Branching in AL.

```
procedure MyProcedure(Arg1: Integer;
           Arg2: Boolean): Integer
begin
    if Arg2 then
        exit(-Arg1)
    exit(Arg1)
end
```

Figure A.4: Declaring a procedure in AL.

A.1 How does AL look?

In terms of syntax, AL resembles Pascal. It is an imperative, and procedural language. As a quick overview, we show some of the basic building blocks of the language in figures A.1 to A.1.

As you see, variables are *typed*. The `Record` variable type corresponds to records in a table on the underlying database of the system. As we will explain later, a user can also define the tables to use in this language.

A.2 AL objects and BC's runtime

Objects in AL are not the general objects as understood in the Object Oriented Programming paradigm. Instead, they correspond to different units of BC's functionality.

Every code element in AL belongs to some object. To see the syntax of how to declare them, see figure A.5. It requires an `ObjectID` a positive integer, an `ObjectName` a string identifier, and an `ObjectType`.

We will show some of these `ObjectType`s, and their effect in the runtime.

```
<ObjectType> <ObjectID> <ObjectName>
{
    // Definition of the object
}
```

Figure A.5: Syntax to define an AL object.

```
page 379 "Bank Acc. Reconciliation"
{
    Caption = 'Bank Acc. Reconciliation';
    PageType = ListPlus;
    PromotedActionCategories = 'New,Process,Report,Bank,Matching,Posting';
    SaveValues = false;
    SourceTable = "Bank Acc. Reconciliation";
    SourceTableView = WHERE("Statement Type" = CONST("Bank Reconciliation"));

    layout
    {
        // ...
    }
}
```

Figure A.6: Example of an AL page.

Figure A.7: The interface the user interacts with corresponding to the AL page from figure A.6.

A.2.1 The Page type

Objects of `Page` type, correspond to interactive interfaces the user experience within the product.

In figure A.6 you can see the AL code used to define the page a user can interact with shown in figure A.7

A.2.2 The Table type

Objects of `Table` type, correspond to persistent storage in the system. Defining this object corresponds to creating a table in the underlying SQLServer database.

Having defined a table, a developer can now use `Record` type variables

```
table 273 "Bank Acc. Reconciliation"
{
    Caption = 'Bank Acc. Reconciliation';
    DataCaptionFields = "Bank Account No.", "Statement No.";
    LookupPageID = "Bank Acc. Reconciliation List";
    Permissions = TableData "Bank Account" = rm,
                  TableData "Data Exch." = rimd;

    fields
    {
        field(1; "Bank Account No."; Code[20])
        {
            // ...
        }
        // ...
    }
    // ...
}
```

Figure A.8: Example of an AL Table.

```
    // ...
    CurrPage.Update(false);
    if not BankAccReconciliation.IsEmpty() then begin
        BankAccReconciliation.Validate("Statement Ending Balance", 0.0);
        BankAccReconciliation.Modify();
    end;
    // ...
```

Figure A.9: Example of using a record `BankAccReconciliation` of the type defined by the table in figure A.8.

of such table, to manipulate and use this table as required, effectively acting as a data layer abstraction.

In figure A.8 you can see the definition of a table, and in A.9 how data can be manipulated by usage of a `Record` variable.

A.2.3 The `Codeunit` type

Objects of `Codeunit` type, correspond to logical units of functionality, much like *modules* in other languages. They are a set of procedures that can

be called from anywhere else within the AL codebase with certain access modifiers.

In figure ?? you can see a procedure defined in a codeunit and in figure ?? an example of it being used from a different AL object.

A.3 AL Tests

Of particular relevance for this project, is how tests are defined in this language. Tests in AL are defined on AL objects of type `Codeunit` with appropriate annotations.

Depending on the scope of a test, these tests can be integration tests, or unit tests. See an example of a test in figure A.10.

The test infrastructure required for running these scenarios with different settings is also maintained by the team and written in AL itself. We go to a bit more detail, as required in section 2.1.2.

We have just scratched the surface with this brief introduction, as it is meant to give a general idea of the type of programs and tests that this thesis project centers around.

```
codeunit 134141 "ERM Bank Reconciliation"
{
    Permissions = TableData "Bank Account Ledger Entry" = ri,
                  TableData "Bank Account Statement" = rimd;
    Subtype = Test;
    TestPermissions = NonRestrictive;
    // ...
    [Test]
    [Scope('OnPrem')]
    procedure BankAccReconciliationBalanceToReconcile()
    var
        BankAccReconciliation: Record "Bank Acc. Reconciliation";
        GenJournalLine: Record "Gen. Journal Line";
        BankAccReconciliationPage: TestPage "Bank Acc. Reconciliation";
        BalanceToReconcile: Decimal;
        i: Integer;
    begin
        // [SCENARIO 363054] "Balance to Reconcile" does not include amounts from
        Initialize();

        // [GIVEN] Posted Bank Reconciliation A with Amount X
        CreateAndPostGenJournalLine(GenJournalLine, CreateBankAccount);
        CreateSuggestedBankReconc(BankAccReconciliation, GenJournalLine."Bal. Acco
        LibraryERM.PostBankAccReconciliation(BankAccReconciliation);

        // [GIVEN] Bank Reconciliation B with Amount Y
        for i := 1 to LibraryRandom.RandInt(5) do begin
            CreateAndPostGenJournalLine(GenJournalLine, GenJournalLine."Bal. Accou
            BalanceToReconcile += GenJournalLine.Amount;
        end;
        Clear(BankAccReconciliation);
        CreateSuggestedBankReconc(BankAccReconciliation, GenJournalLine."Bal. Acco

        // [WHEN] Bank Reconciliation B page is opened
        LibraryLowerPermissions.AddAccountReceivables;
        BankAccReconciliationPage.OpenView;
        BankAccReconciliationPage.GotoRecord(BankAccReconciliation);

        // [THEN] "Balance To Reconcile" = Y.
        Assert.AreEqual(
          -BalanceToReconcile,
          BankAccReconciliationPage.ApplyBankLedgerEntries.BalanceToReconcile.AsDE
          StrSubstNo(
            WrongAmountErr, BankAccReconciliationPage.ApplyBankLedgerEntries.Balan
            -BalanceToReconcile));
    end;
    // ...
}
```

Figure A.10: Example of a test codeunit and test procedure.

Appendix B

Evaluation results

Bibliography

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

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

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

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

[10] 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.

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

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

[13] 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.

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

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

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