Deliberate Code Coverage

by

R.M. de Britto Heemskerk

Student number:	4961269
Thesis committee:	S. Proksch, TU Delft, Thesis advisor
	C. Brandt, TU Delft, Daily Supervisor
	G. Migut, TU Delft, Graduation committee member

TU Delft

Acknowledgements

This thesis has been coming for a long time. With various delays and complications along the way, I want to put special emphasis on the various people who got me through this entire process.
I want to start of by thanking my thesis committee. First of which is Carolin Brandt, who has been there the entire way. Thank you very much for the patience and the opportunity to work together. Next is Sebastian Proksch, who helped a lot by offering a different perspective from both Carolin and I. And finally, Gosia Migut, without which we could not finish the process. I also want to give special thanks to student counsellor Michel Rodrigues who helped me get back on track when things became more difficult.
But other than the professors I also have friends and family I would like to thank for their support during this process. To start I would like to thank my parents for their continuous support and belief in me. Furthermore, I would like to thank my best friend Sander for being able to call on him whenever to have a good time. Special thanks goes out to Alex as well, who helped me by body-doubling and in this way actually being able to finish work every once in a while. Furthermore, they invited me to a community of like-minded people which was desperately needed this past year. I'd also like to thank Lieve and Max, for making me feel appreciated as a friend.
Finally, despite not being named individually, I would like to thank all other friends I have. The company which my friends bring will always be one of my major drivers in life, and they have given me the energy to finish this thesis.

Summary

Testing is a major part of software development. Within testing often coverage requirements are used as a tool for quality assurance. But what code should be covered to reach the requirement is not clear. To address this, we suggest using historic data to make these decisions more deliberate. In other words, we want to use machine learning to predict coverage.
Building upon previous research, we investigate how different approaches affect the performance of decision tree models. We did this using data from the Mozilla Firefox codebase. We focused in particular on the C/C++ code within there. Naively splitting training and test set and representing coverage per lines leads to best performance. Analysis showed that grouping coverage data based on basic blocks slightly lessened the predictive performance of the model. Meanwhile, splitting the data across the training set and test set based on their files appears to take away all predictive performance.
This study provides a new dataset for use in developer coverage prediction. It also introduces a new way of representing coverage data for developer coverage prediction, being basic-block coverage. And finally, gives insights on the effects of different coverage representations on decision trees.

Contents

1 Introduction

It is hard to exaggerate the importance of testing in software development. Without sufficient testing many pieces of Software would be delivered with more bugs, and there is less of a guarantee in quality. Estimates shows that the cost of bad software $\$2.41$ trillion in the U.S. alone, where about 50% is wasted on bug fixes [1]. However, how much testing is enough? In an optimal world all code would be 100% tested. However, due to several types of resource constraints, i.e. time, money or a combination of the two, fully testing code is infeasible. As such, some code will not be tested. To limit the amount of code is untested usually benchmarks of 80% coverage are set. This still leaves 20% of code which is not covered, and what code is not covered is arbitrary. At the end of the day, how can a software developer argue why a piece of code was left untested if it someone asks about it during code review? In the best case, the software developer would be able to argue why pieces of code should be covered or not, by attributes of the code or its context.

In terms of state of the art to steer coverage we would be looking in a few different domains. The first of which is defect prediction. Defect prediction is a technique to find defect-prone areas of code using historical defect data before the testing phase. Kaya et al. [2] This is often done using a dataset with various metrics regarding pieces of the code. By figuring out where the defects in the code are, programmers could decide to prioritize their efforts on code with a higher risk of defects.
The other domain we will refer to as Developer Coverage Prediction (DCP), as coverage prediction already describes a different problem. DCP involves predicting which code would be covered based on historic information of the code base. To my knowledge this has only been attempted by two papers, these being Ivankovic et al. [3] and Brandt and Ramírez [4]. Ivankovic et al. [3] find similar pieces of code by doing pattern matching on bigrams and trigrams of commonly covered code. Brandt and Ramírez [4] on the other hand trained machine learning models to predict coverage on Java code.

Despite reasonable results, defect prediction still has issues regarding actionability [5]. An area of the code can be marked as defect prone, but this does not necessarily mean that it makes sense to cover it with tests. There are other aspects which play a role in testing as well. For example, there may be a higher chance of defects in a piece of code, but if it is there for exception handling or debug purposes it may not need to be covered by tests. Furthermore, the code might be far out of the way of the main program execution, meaning there is less of a risk of bugs being triggered there. We conjecture that these aspects might be noticeable by taking in developer intent or expertise.
As such the vision is for us to create a tool which without code execution, by characteristics of the code, can tell us which parts of the code are more important to be tested, and feasible. This would allow developers to prioritize testing efforts in order of importance, and properly budget it. Considering an optimal world, this could be done with sufficient certainty that if something goes wrong in untested code, there is sufficient argumentation as to why this code was not tested.

This however, assumes we already have such a tool. The challenge lies in creating such a tool. One potential way to do this, and the way we will explore in this thesis is by training a model on data. The basic idea is, that if high quality data is chosen, we could use historic data to predict future cover-

age. High quality data in this case is represented by data where testing efforts are less arbitrary than usual.
To fully evaluate such a model is beyond the scope of this thesis though. The focus therefore will be to serve as a stepping stone, by comparing different approaches and see which performs better, in terms of both accuracy and potential overfitting.

This thesis sets itself apart by stepping away from line-based coverage and using basic block coverage instead. This is because we feel like line based coverage does not represent coverage accurately. This is due to the fact that if one line in a basic block is covered, all lines of the basic block will be covered.
Since our other concern is overfitting, we also look into splitting up directories as a whole between test and training set. We will refer to this as **File-splitting**. This is opposed to treating all lines equally. This way we try to limit contamination of the test set. Both of these approaches deal with **data representation**. For the basic block we are dealing with the actual representation of the data. Meanwhile, File-splitting deals with which data gets represented in the train and test sets.
These goals give rise to the following research questions:

RQ1. How does representing coverage as block-based instead of line-based influence the performance of the coverage prediction model?

RQ2. How does splitting on files influence the performance the model?

RQ3. Which metrics are most influential in terms of predicting coverage?

To see how the representation of coverage influences the model we use the following approach. We first train a model with the same parameters on both representations. Then, for both approaches we will retrieve information on precision, accuracy, recall and f1 score using 10-fold cross-validation. Next, by looking at the learning curve and validation curve we get extra information regarding the behavior of the model. To compare the internal workings of the models, we will also compare the most influential metrics. Finally, we will look at the difference between the training accuracy and the test accuracy, this gives some indication of the amount of overfitting done by the model. Combining all these aspects we will conclude which approach is better overall.
To compare splitting on directories to the naive approach, we use a similar method. We train two new models based on the directory split, and do the same comparisons as for the first research question. This time however, we compare full training set with the directory split. Following this we draw conclusions on which is the best approach.
To find which metrics are most influential we compare them based on permutation importance. Which determines the mean decrease in performance from leaving a certain feature out. The most influential features will then be inspected by using a Kernel density plot to draw further conclusions.

As results we have found the following.
Comparing the block-based approach to the line-based approach showed a minor decrease in performance. We hypothesize that the line-based approach memorizes more from lines within basic blocks, which lead to a higher performance.
When splitting the data based on files the overall accuracy decreases drastically for the file split. Our hypothesis is that the model does not generalize well between files. Potentially because different teams work on different files. For the metrics we find that the most important ones are the number of arguments and the Halstead purity ratio. More generally, metrics which give an indication of complexity of the underlying code appear to do well.

The main contributions of this thesis are as follows:

- A dataset prepared for the task of developer coverage prediction.
- A new method of representing coverage for Developer Coverage Prediction, being Block-based coverage.
- An insight on the two representation approaches set: Line based-coverage and Block-based coverage
- An insight on the two training set preparations: Naive and File Split

2 Literature

In the literature section we will discuss some related topics to the thesis. Here however, I will explain their relevance to the topic.

First we have Coverage. As implied by the title ”Deliberate Code Coverage”, coverage itself plays a major role. As such it is important to properly define what is meant by coverage and its importance in the testing landscape.

Secondly we have Defect Prediction. Defect prediction is a closely related domain, which tries to discover faulty code without code execution. This could be considered as an alternative to our model, where developers focus their testing effort on pieces of code which are considered to have a high risk of faults. However, the results of defect prediction are often limited. Defect prediction also does not consider developer behavior/ effort it takes to actually test the code.

Finally we have Coverage prediction. This is actually the domain this thesis deals with directly. This domain is not particularly well-researched, but there have been some previous studies in this domain, but these either use different methods to predict coverage, or do not consider the full basic block when predicting coverage.

2.1. Coverage

Since our paper deals with code coverage explicitly, it is important to properly define what we mean by it. A piece of code is considered covered if there is a test case which executes that piece of code. There are various scopes, from statement coverage to path coverage. [6]

For this paper we will mostly be dealing with statement coverage, however, since in the code we will be dealing with most statements take up one line, line coverage can be used synonymously here. Simply put, a statement/ line is considered covered if a test executes the statement/ line. If the line contains a branching instruction, if either branch is exercised, it is considered covered by the test.

While commonly adopted, it is ”an open research question whether code coverage alone reduces defects” [7]. Not all defects can be discovered even when covering all code, because the relevant areas might not have been implemented yet [8]. So only looking at the code coverage ratio is not sufficient. Despite this, while high code coverage does not give a guarantee of quality, low coverage does guarantee that large pieces of code go untested [7]. From the developer side code coverage is generally perceived as a positive inclusion, and used quite often [9]. It also generally shortens the review process by making it easier to see which parts of the code have been covered [3].

While having reached the conclusion separately, our model follows an idea present in the Google code coverage best practices blog post [7]. The idea is that more important than what is tested, is a human judgment of what is not covered. By making this tool, we are hopefully aiding in making this process more efficient, and gaining deeper insight into developer behaviors regarding coverage.

2.2. Defect Prediction

Defect prediction is a related topic which has seen quite some research over the years, and many have used ML models to tackle this problem. Pachouly et al. [5] defines it as ”a mechanism that indicates

possible defects in the newly written code or modified existing code without testing the code."
Translated to the ML domain this means that a model is trained to use aspects of the code to predict where potential defects lie in the code. The same paper, found 146 different papers in this domain in 2022. And it makes sense that this topic is seeing attention, as knowing what part of the code is potentially buggy could serve as a guiding hand for testing. However, the same paper mentions that prediction outcomes are often limited, with few to no actionable items.
Naseem et al. [10] studies how different tree-based ML models perform on the problem of defect prediction. It does so by training them on different benchmark datasets commonly used for this task, and shows that they have good potential by judging them on various metrics. In particular random forest performs best on 5 out of the 10 datasets. In fact, Random Forest achieves the highest overall average accuracy with 90.22%. However, it should be noted there is a pretty major class imbalance in most datasets. With at most 20.5% belonging to the defective class.
While the goal of defect detection is different than the proposed goal of our research, the underlying mechanics are similar. As such some of the metrics present in these papers can be reused. In particular the Halstead and Mccabe metrics seem to be commonly used. Since overall developers would like to cover high-risk code, it makes sense to investigate if metrics which can be used to detect defects can be used to predict coverage.
The main difference between our model and defect prediction lies in what our they aim to predict. Where defect prediction tries to predict faulty code, our model would predict which pieces of code are most likely to be left uncovered by developers. This adds an element of developer experience and an implicit element of effort required to test certain pieces of code.

2.3. Predicting coverage

As a final topic we have coverage prediction, the main connected domain of this thesis. Seemingly the suggestion to do some kind of prediction of coverage the way we suggest is pretty new. The main connected paper, other than the paper we are building off of, is the paper by Ivankovic et al. [3]. This paper develops a new method for dealing with coverage.
They compare the covered code to other pieces of uncovered code, finding similar code based on textual similarity using n-grams. This is then used to give a suggest making a test case for the uncovered piece of code. They reported positively on their results stating that it "improves code quality and prevents defects from being introduced into the code."
Our suggested use case for our model looks quite similar, but instead would use metrics and an ML model to detect similarity of covered code.

3 Data Collection

One of the most important aspects when dealing with ML is having a proper dataset, both for the sake of reproducibility and creating a proper model. In this chapter we will discuss the data we chose to use. Since we are using data from Mozilla, we will give a quick explanation of Mozilla themselves and the code base. Furthermore this chapter will dive into what is done to ensure the quality of the data. From what type of files are being used, to the review practices at Mozilla.

3.1. Mozilla

Before discussing what the underlying data entails, it is good to first take a look at the organization the data comes from. Mozilla is an organization that has had a substantial impact on the browser market. In this section we will discuss its history and their code management. In section 3.1.1 we will discuss the history of Mozilla, and their significance to the browser. Section 3.1.2 discusses the code at Mozilla, putting emphasis on how they deal with code quality.

3.1.1. History

Mozilla is a community open source project founded in 1998. It consists of two separate entities, the Mozilla Foundation, and the Mozilla Corporation. The Mozilla foundation is the non-profit part of Mozilla, whereas the Mozilla Corporation is the taxable subsidiary. Their mission statement is to ensure that the internet remains a force for good.
Mozilla is most famous for it's web browser, Firefox, which was built in the early 2000s. It changed the browser ecosystem, and lessened the monopoly internet explorer had at the time. At the height of their success, they were the second most popular browser, with approximately $32,21\%$ market share. [11] However, over the years Chrome has grown to be the biggest. As of June 2025 they are the 4th biggest desktop browser with $5,86\%$ market share of the desktop browser market. [11] However , among non-chromium based browsers, Firefox stands as the largest.
Other than Firefox, Mozilla has other products as well, mainly consisting of internet services like Email and a VPN. But also a privacy management application, and a Phone and email masking service. However, for this thesis we focus on the Firefox codebase.

3.1.2. Code at Mozilla

Mozilla has been around for a significant period of time, for a software community. As such they have a significant amount of legacy code. For this thesis, legacy code is defined as important yet hard to manage code which was introduced early in the project's life cycle. Often this code is also characterized by being less tested.
While unclear what they used to analyze their code in the past, currently they use various tools and processes to ensure their code meets their standards.
The first among these is their code review process. All patches are submitted to the tool they use for code review, Phabricator. Here all patches are tagged based on their testing status, and reviewed. The tag system shows if a particular commit has been tested or not. If it is, it gets tagged with testing-approved, otherwise it gets tagged with one of the testing-exception tags. In case of testing-exception,

the developer comments the reason for not having tests, or where the tests are located. Commits only go through if checked by module owners or one of their dedicated peers. Important to note is that while coverage is tracked, there are no explicit coverage requirements at Mozilla.
Furthermore, they keep track of all of their bugs using Bugzilla. This allows developers to have a good overview of all the currently known issues in the code. Most patches also directly address specific bugs, which makes it easier to see if the patch actually does what it is supposed to.
As a final tool, developers working on Mozilla have acess to SearchFox, an indexing tool for the mozilla code base. This makes it easier to find particular files, and shows the coverage data of the latest revision.

3.2. Files

As this thesis builds on research done in collaboration with Mozilla, we are limiting ourselves to files in the mozilla-central codebase. We are using a revision [1] of June 2024, with the corresponding coverage report. Within these files we are limiting ourselves to C++/C files, because we construct the basic blocks manually. This makes sure we have fewer different structures to deal with from the different programming languages.
These files are often split across main files and header files. Where the header files use the .h extension. In terms of filtering files, we first remove the files which give have a difference in lines of more than 2 between the coverage file and the original file. This is because in the cases where the difference is less, this is mostly caused by closing braces at the end of the file not being counted. But in the other cases, there is no guarantee that the covered lines actually match the original lines. And finally we filter out the files which give errors when calculating the AST, unfortunately this includes all .h files, leaving us with 4357 files.
This filtering does mean we are missing out on a not insignificant part of the Mozilla code base. This means there may be certain coding practices or particular logic which we don't handle with the model. Furthermore, the .h files actually contain logic which often is present in the main C++ file, potentially changing control flow. The fact that we are missing this sometimes crucial information should be considered when evaluating the results.

3.2.1. Domain

The number of files gives an indication of the total project size. However, it gives no indication what the domain is of the code within.

As we are using the Mozilla Firefox code base, the files are all files which are relevant to the browser properly functioning. A summary on all the directories can be found from the firefox documentation.[2] There are 42 total directories, however, not all directories are as relevant, as our files are not equally distributed across all directories. This can be seen represented in the below table 3.1. The most relevant files for this thesis, as they have the most c++ files, are as follows:

`dom`: Interface Description Language (IDL) definitions of the interfaces defined by the Document Object Model (DOM) specification

`gfx`: Interfaces for graphics toolkits. Provides methods for drawing images, text, and basic shapes.

`js`: Code for their JavaScript engine called SpiderMonkey. Also includes support code to call JavaScript code from C++ and vice versa.

`media`: Contains sources of used media libraries, for example, libpng.

`intl`: Internationalization and localization support. Makes it so code can be used worldwide. Includes code for correct string representation, and conversion of encodings.

`netwerk`: The networking library, handles transfers from and to servers. Also takes care of URI handling.

`security`: Contains Network Security Services (NSS) and Personal Security Manager (PSM), to support cryptographic functions

`toolkit`: Contains front-end components shared between applications.

[1]691601156e52d3f8b6f035ff6e02eb1575d3a385
[2]`https://firefox-source-docs.mozilla.org/contributing/directory_structure.html`

Looking at Table 3.1, we see that the highest number of C++ files are in the dom directory.The high concentration of c++ files in dom means dom will likely mostly shape the model we will train. Of the directories with a high number of C++ files (files > 350), third_party is least covered, with intl second least. It makes sense that the third_party directory would be less covered, considering it is not their own code. Gfx, also has a lower coverage which shows that graphics algorithms are potentially harder to test.

Directory	Coverage%	LOC	Percentage C/C++	Total Files	C++ Files
dom	82,17	626192	0,5	4486	**2243**
gfx	46,88	573859	0,471	3218	**1516**
media	47,53	240080	0,766	983	**753**
js	82,98	396072	0,475	1337	**635**
third_party	29,87	499990	0,122	4021	**491**
security	44,53	194572	0,807	543	**438**
toolkit	62,35	466172	0,243	1790	**435**
intl	37,55	144354	0,547	766	**419**
netwerk	71,86	168554	0,528	721	**381**
layout	91,14	174982	0,47	713	335
xpcom	85,88	66286	0,53	534	283
widget	57,12	82288	0,518	434	225
accessible	78,35	31589	0,515	268	138
ipc	50,75	39180	0,396	318	126
modules	72,69	20452	0,772	145	112
nsrpub	41,33	19838	0,989	93	92
mfbt	84,64	26564	0,444	189	84
image	85,84	27640	0,523	149	78
mozglue	71,08	19536	0,507	134	68
editor	76,99	50981	0,558	104	58
tools	81,49	19135	0,44	116	51
parser	82,16	26189	0,475	99	47
storage	87,25	7310	0,583	62	36
browser	83,76	171594	0,041	689	28
docshell	79,82	20270	0,473	55	26
uriloader	78,89	8950	0,491	53	26
extensions	52,61	15865	0,407	59	24
memory	76,74	6247	0,559	34	19
devtools	74	251335	0,011	1548	17
caps	87,34	4330	0,485	33	16
hal	70,44	998	0,889	18	16
testing	24	3462	0,733	15	11
xpfe	82,46	3853	0,5	12	6
build	9,09	99	0,8	5	4
chrome	82,38	891	0,444	9	4
startupcache	90,8	750	0,571	7	4
view	86,33	1295	0,5	4	2
config	100	1	1	1	1
other-licenses	53,77	1577	0,1	10	1
remote	94,34	26436	0	162	0
services	84,1	20838	0	79	0
servo	88,49	40289	0	241	0

Table 3.1: Table showing file information of the various directories

4 Machine Learning

In this chapter we will discuss everything involved in the machine learning process for this thesis, including how we evaluate the resulting models. As such Section 4.1 discusses the selection of metrics and how the data is represented for the machine learning. Section 4.2 discusses the training and selection of the model. Finally in Section 4.3 we discuss how we evaluate the results.

4.1. Metrics

In this section we will look at the selection of metrics and representation of data. The selection of metrics can be found in Section 4.1.1. While how the data is represented is in Section 4.1.2

4.1.1. Selection of Metrics

To give the model sufficient information for training it is important to pass along the important features of the blocks. We try to do this in such a way that the model does not just learn where the block is in the code, and give a prediction based on that. The goal instead is to see if the model uses the input features to generalize when code is or is not covered. As a basis for potential metrics we looked at the features which were chosen in Tufano et al. [12]. That paper however trains a model on java code, and not all of those features translate as easily to C++. In the end we decided to focus on metrics which Mozilla already gathers themselves. Some other features were added based on easy implementation, combined with a hypothesis that they could be informative for predicting coverage. This gave us the following list of features for the line-based approach.
The asterisk (*) indicates that the metric is taken from the static code analysis tool used by Mozilla, and as such was taken at the level of the deepest closure. This is because the static code analysis considers these metrics at different levels. So we take the metric at the deepest level it is considered for that line of code. This is to avoid the scenario where either the metric would be the same for the entire file, or the metric would be null for certain lines.

- Line Length
- Starting line of first line of method
- Nesting depth
- Halstead metrics*
- Logical lines of code*
- Source lines of code*
- Maintainability index*
- Cognitive complexity*
- Cyclomatic complexity*

- Number of arguments of method*
- Boolean attributes

These are the metrics we use for the block-based approach:

- Lines of code in block
- Starting line of first line of block
- Nesting depth
- Halstead metrics*
- Maintainability index*
- Number of calls in block

The boolean attributes indicate if the line is in a match block, in a catch clause or in a try statement. Taken from the documentation of the static code analysis, the maintainability index is suite that allows to evaluate the maintainability of a software.

Halstead metrics These metrics are some of the metrics used by Mozilla on their code. Since they were easily accessible to us, and these are used to analyze their code in practice already, we decided to use some of them for the training of the model. Halstead Metrics are also often used in defect prediction.[10]

- n_1 (Number of distinct operators)
- n_2 (Number of distinct operands)
- N_1 (Total number of operators)
- N_2 (Total number of operands)
- Program Vocabulary $(n = n_1 + n_2)$
- Program length $(N = N_1 + N_2)$
- Level $\left(L = \frac{n_1}{n_2}\right)$
- Estimated program length $(\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2)$
- Purity Ratio $\left(PR = \frac{\hat{N}}{N}\right)$
- Volume $(N \log_2 n)$
- Difficulty $\left(D = \frac{n_1}{2} \times \frac{N_2}{n_2}\right)$
- Effort $(E = D \times V)$
- Time $\left(T = \frac{E}{18}\right)$
- Bugs $\left(\frac{E^{\frac{2}{3}}}{3000}\right)$

To try and mitigate the risk of having the model learn the coverage data of functions instead of a more general rule, we tried binning the data. The idea behind this is that by having the data not be as granular, the resulting decision tree branches cannot pinpoint specific floating point values.

4.1.2. Representation of data

In this section we will discuss the how we represent the training/ test data. First we discuss how we deal with the Coverage data. Next we discuss how we add in the corresponding metrics.

Coverage Representation In this thesis we consider two ways to represent the coverage data. This can be seen as changing the minimum group size considered. In the first case, we take the coverage data as-is, which is line coverage. This means that for every line of code we have information if it is covered or not. Sometimes as a third scenario, the code is left outside of coverage. Usually this is meant for comments, but sometimes code can be manually removed from consideration for the coverage.
In the second case, we group lines together based on if they are part of the same basic block. For this we first use the definition of a basic block:

”A basic block is a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed).” [13] The reason we consider this approach is we think lines without context do not properly reflect the behavior of coverage. Because in principle, if one line of a basic block is covered, all lines in that block are covered, and vice-versa with not being covered.

To properly split the code into basic blocks, we use the following simple algorithm. We set leaders for each basic block, evaluating the code sequentially, where the lines from the leader up to the next leader are part of the same block. The leaders are set according to a few rules (adapted from [14]):

- The first line of code (of a program) is a leader.
- The target of any jump is a leader.
- The next line following a jump initiator (conditional or not) is a leader.

To see how this works in practice an example is given in Appendix A. The effect of this is two-fold. First of all, this allows us to bring in context for the piece of code. Second of all, it limits the extent to which samples with nearly the same information gets introduced. This is important since if they were included the problem turns more into memorization than generalization.
This representation does affect some of the metrics, since plain line-length does not make sense anymore. Instead this has been replaced with block length, which is the number of lines which is part of a block. For either representation, we combine the coverage data with the metrics we introduced in the previous section. For each line or block, we store the coverage information with the respective metrics in tuples, which is then stored as rows in the dataset.

4.2. Machine Learning Selection and Preparation

In this section we will discuss the selection and the preparation of the machine learning models. The selection can be found in Section 4.2.2. Meanwhile, the preparation of the model can be found in Section 4.2.1

4.2.1. Model Selection

The main models we considered were the models available through sk-learn library. These are Decision Tree (DT), Random Forest (RF), Stochastic Gradient Descent Classifier and Kernel approximation. These each come with certain strengths and weaknesses. For this thesis we ended up choosing only to use Decision trees, for reasons we will elaborate on in this chapter.

Decision Tree A DT consists of many decision nodes ending in leaf nodes. At every decision node, branches are made which split based on one of the input features. This continues until a leaf node is reached which predicts which class it should belong to.

One of the major advantages of decision trees is interpretability. Every input will travel down a specific path in the tree, and each node can be internally examined to understand which path it took. As such, more general conclusions can be drawn from it. However, decision trees are quite prone to overfitting. If Decision trees are not given any stopping condition, the DT can continue branching indefinitely essentially exactly encapsulating the training data. This leads to too granular predictions, which causes overfitting.

Random Forest Random Forest essentially consists out of multiple decision trees, which each contribute a certain weight to the prediction.

In terms of advantages, Random Forest usually is quite accurate. This comes with some drawbacks though. Random Forest take significantly longer to train and take longer to give predictions. Similarly to decision trees, it is quite prone to overfitting as well, in many cases worse. It is also less interpretable than a decision tree.

Other Machine learning models Other candidates for potential models included Stochastic gradient descent, and kernel approximation. The advantage of both of these is that they are quick to train and to give predictions. However, Stochastic gradient descent can only fit a linear model, and depends heavily on which kernel is used.
Kernel approximation usually fits a linear model, however by making use of a transformation it can account for non-linearity. However, more complex boundaries are still hard to describe. As such for this thesis, looking at the data and a few trial runs, showed that either of these models did not perform sufficiently on this dataset. This is likely due to a complex decision boundary, if there even is an appropriate one.

Chosen Model Having considered the options available, we chose to use decision trees. This was done because of the interpretability of decision trees, and its ability to find patterns in data that is not linearly separable. Furthermore, decision trees deal with high quantities of data better than random forest does.

4.2.2. Preparation

In this section we will discuss the steps we took to prepare for the model training. This includes discussing issues with the dataset and pre-processing. Next we will discuss the hyperparameter tuning for the models. And finally, we will discuss the training and test split for the models.

Big dataset Our dataset is large, seeing as we are dealing with over 1 million samples. As a result certain models become infeasible to use based on long training times. It would be possible to take a subset of the full dataset as a representative for training. However, we fear that we lose accuracy, or can not as easily describe such a varied dataset.

Pre-processing Part of the pre-processing steps have been considered in Chapter 3 and Section 4.1. Namely the acquisition of the dataset and encoding of the dataset.
To properly use the data, there are still a few things that should be done. The first is proper scaling of data. For certain models it is important that all data is scaled similarly. As such we ensure to normalize the data. We do this by using the $MinMaxScaler$ from sklearn. This ensures the minimum value corresponds to 0, and the maximum to 1, and the rest is scaled accordingly.

Hyperparameter tuning Before training the true models we first have to find the optimal parameters for our model.

For DT we deal with the following hyperparameters:

- Class weight {Balanced,None}
- Regression Criterion {gini,entropy, log_loss}
- Minimum samples leaf (int)
- Maximum tree depth (int)

To do this we employ the grid search strategy, which tries all combinations of hyperparameters which we set. We ran grid search with the following options for the hyper parameters:

- Class weight: {**Balanced**, None}
- Regression Criterion: {gini, **entropy**, log_loss}

- Minimum samples leaf: [**1**, 3, 5, 10]
- Maximum tree depth: [1, 3, 5, 10, **None**]

Bolded are the best performing hyperparameters.

Training The next step is to split the data into a training set and test set. Our approach to doing this was informed by the literature on defect prediction [5, 10, 15] and the paper by Brandt and Ramírez [4]. Essentially, many of the defect prediction papers use either the NASA PROMISE datasets, or the NASA metrics data programs datasets. The entries in these datasets are all tuples of metrics and defect information belonging to distinct functions. Our case is different in that our coverage is line or block based. Switching to grouping our data to be at function level loses us the flexibility of selecting specific branches to cover. Meanwhile, Brandt and Ramírez [4] used line-based data similar to our case. While attention is put into vetting the data, entries seem to have been naively split between the training and test set in both cases.
Having seen this we split in two ways, naively and splitting the data based on files. In the naive approach we don't group the lines or blocks at all before splitting. This means that all lines can randomly end up in either the training set or test set. We use a 70-30 split for training and test set.
In the other approach, we randomly assign 70% of all files to the training set, and 30% of all files to the test set. This is done to try and limit the memorization of lines between the training and test set.

4.3. Evaluation Methods

In this section we will discuss how the way we evaluate the models to answer the research questions. In Section 4.3.1 we discuss what makes up a good model on a base level. Next, Section 4.3.2 discusses the experiment setup for the first two research questions, which deal with comparing performance of the coverage prediction models based on data representation. And finally, Section 4.3.3 deals with the experiment setup of research question 3 which compares the influence of the code metrics on the coverage predictions.

4.3.1. What makes a good model

To evaluate which model is better, we first need to properly define what makes a model better than another.
For nearly any machine learning model, we care about the accuracy of the model. As such we want to see which model has the best fit given the training data. For this we can look at different metrics regarding classification. Usually those are given inherent from verifying the model on the test set. These are Accuracy, Recall, Precision and F_1-score. These are defined according to the following formulas:

$$\text{Accuracy} = \frac{TN + TP}{TN + TP + FN + FP}$$
$$\text{Recall} = \frac{TP}{TP + FN}$$
$$\text{Precision} = \frac{TP}{TP + FP}$$
$$F_1\text{-score} = 2 * \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Where TP and TN stand for True positive and True negative respectively, and FP and FN for False positive and False Negative respectively. For each of these metrics, the higher the result the better.
Furthermore, as we want to understand the underlying behavior of the models, we will compare them by means of validation curve and learning curve. These show how the model behaves when hyperparameters are changed, and how the model behaves when increasing the training data. Finally to see if the models are comparable, we look at the influential metrics.

4.3.2. RQ 1 & RQ 2: Comparing models based on data representation

For the first two research questions, we will use the same comparison methods, swapping out the relevant model to compare. For the first research question we compare the naive lines approach with the naive basic block approach, while the second research question covers the comparison between the naive lines approach and the File-splitting lines approach. To do a proper comparison between the models we use multiple techniques, which each was done using 10-fold cross-validation.

We first compare the performance metrics (accuracy, precision, recall, f1-score) of the two models. The results of this will be shown in a bar chart. Based on the average score, as well as the variance we can see how the model usually performs when training on the representation of data.
Next we compare validation curves to see how the test and training accuracy changes based on the max depth of the tree. By looking at the difference between the test and training accuracy curves, we can estimate how much the model overfits. This allows us to compare if the representation of data changes the underlying learning of the model, potentially leading to a higher or lower overfit.
For the same reason we compare the learning curves, which shows how the model performs as more data is made available for training. Once again, by looking at the difference between the test and training accuracy curves we can see if the overfit changes based on the representation of the data.
As a final comparison we may look at the most important metrics. This is to conclude if the models use similar internal logic. The inner workings of how we determine metric importance will be explained in Section 4.3.3.

4.3.3. RQ3: Metric evaluation

To determine how well a metric predicts a feature a number of different methods can be applied. In our case we will be using a correlation matrix.
Correlation matrices give an indication of linear correlation between attributes. This can show us if certain metrics are redundant, and how well any individual metric linearly predicts code coverage.
The other method we will use to evaluate the metrics are their importance to the model. As we will see in the results section, there does not seem to be a linear relationship with covered and non-covered code for any individual metric. As such, it makes sense to evaluate the metrics based on how much of a difference they make for the model. This is can be done in a few ways, but we will focus on two, the mean decrease in impurity (MDI) also known as gini importance and permutation importance.
MDI is calculated by first taking the decrease in impurity in a node. The impurity in a node (D) is defined by the following formula:

$$Gini(D) = 1 - \sum_{i=1}^{k} p_i^2$$

Where k is the number of classes, and p_i is the probability of belonging to class i. The decrease in impurity after splitting on a feature is then averaged over each node where the feature is used to split, multiplied by the probability of reaching this node. Doing this for each feature gives an importance score. Important to note, however, is that this way of ranking features does favor features with high cardinality. As such features which have not many distinct values or a lot of repetition in the data will be less favored in this approach. Seeing as there is a clear bias for this approach, we will not be using this method. Instead we will be using permutation importance.
Permutation importance on the other hand calculates how well the model would perform with essentially the removal of that attribute. Taken from the User guide of sklearn, the algorithm works as follows:

- Inputs: fitted predictive model m, tabular dataset (training or validation) D.
- Compute the reference score s of the model m on data D (for instance the accuracy for a classifier or the R^2 for a regressor).
- For each feature j (column of D) :
  - For each repetition k in $1, \dots, K$:
    - Randomly shuffle column of j dataset D to generate a corrupted version of the data named $\tilde{D}_{k,j}$.

    ◇ Compute score $s_{k,j}$ of model m on corrupted data $\tilde{D}_{k,j}$.
  – Compute importance i_j for feature f_j defined as:

$$i_j = s - \frac{1}{K}\sum_{k=1}^{K} s_{k,j}$$

This method should have less bias for high cardinality features.

After having calculated the importances for all metrics, we will display it in a ranked bar chart. This allows us to easily see which metric has the greatest influence on the model.

5 Results

In this chapter we will discuss the results of the experiments. The first research question, on the performance of Basic Block coverage representation, is discussed in Section 5.1. The second research question, on the performance of the File-splitting method, is discussed in Section 5.2. The third and final research question, on metric importance, is discussed in Section 5.3.

5.1. Lines vs Basic block

In this section we compare the performance of the naive line-based approach with the naive basic block approach. This is done to answer the question:

> RQ1: How does representing coverage as block-based instead of line-based influence the performance of the coverage prediction model?

To answer this, two separate decision tree models were trained on the same dataset. However, with the line-based approach the rows represent of separate lines of code and their attributes, while with the basic block approach, each row represents a basic block in code with its relevant attributes. For each of the experiments we use 10-fold cross validation, and the same set of hyperparameters to ensure we can compare the results fairly.

As a first experiment to properly compare the performance of the line-based approach with the basic block approach, we use 10-fold cross-validation. As a quick recap, the line-based approach simply uses the base line coverage to For this approach we split both data sets in 10 different parts. Following this, we train a decision tree on 9 of the folds and leave 1 of them as a test set. We repeat this until each fold was used as a test set. For each of these runs, we calculate the performance metrics. This allows us to compare the average performance of the decision tree model on the two approaches. These results are portrayed in Figure 5.1:
The results of this cross-validation are supposed to indicate which method performs better. In this bar chart it can be seen that the naive lines approach outperforms the naive blocks approach on all metrics. For most metrics the naive lines approach reaches about 95%, while for most metrics the naive blocks approach is about 90%. This shows that in terms of performance metrics, the line coverage leads to better results.

As a second experiment, we compare the extent to which the models overfit. To do this we compare the validation curve for max tree depth between the two models. For this a model is trained for different max depths for each approach. Each of these trained models is evaluated on the test set, and the accuracy is recorded. To get a proper average for these, 10-fold cross-validation is applied for each of the max depths. Following this a curve was fit to the results. These curves can be seen in figure 5.2 and figure 5.3.

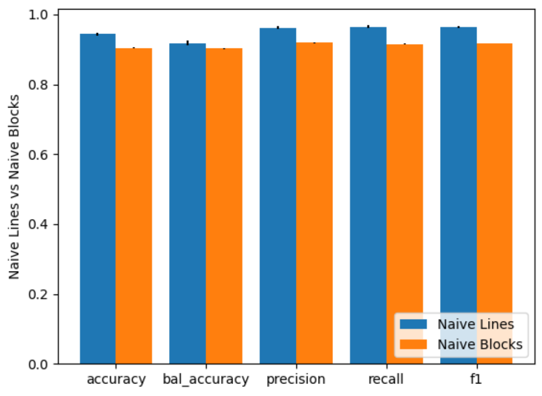
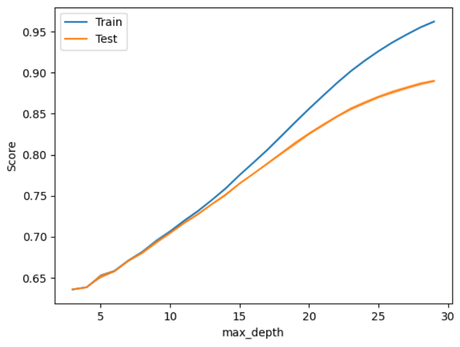


Figure 5.1: Bar Chart comparing Naive Line Coverage against Naive Basic Block Coverage



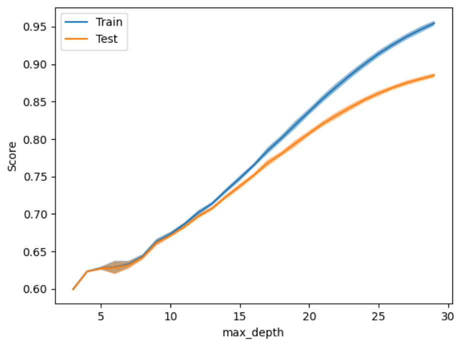


Figure 5.2: Validation curve plotting max depth against accuracy for the naive lines approach

Figure 5.3: Validation curve plotting max depth against accuracy for the naive blocks approach

Looking at these validation curves, we see very similar behavior. Training and test accuracy appear to diverge at max depth around 11. At max depth 30, we see both have a similar difference between test and training accuracy. This implies that both training sets lead to a similar overfit, for the same value of max depth. This shows that grouping the lines based on basic blocks did not substantially lower the overfit.

The third experiment involves comparing the Learning Curve between the two approaches. The learning curve is generated by fitting a curve to the results of training the models on differing numbers of samples. This shows how the model behaves for increasing number of samples, which gives an indication on if sufficient data has been provided. Once again 10-fold cross-validation is applied to get averages for the curve.

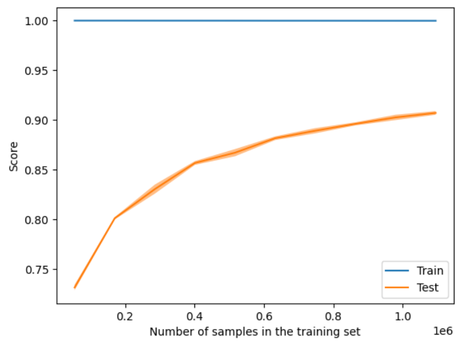


Figure 5.4: Learning curve for the naive line coverage approach

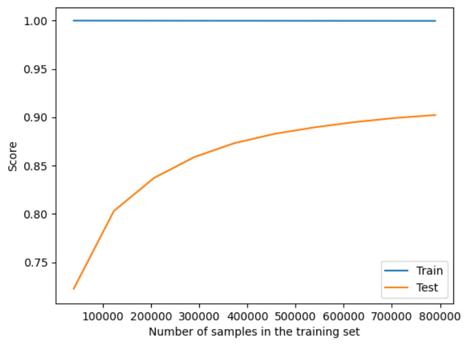


Figure 5.5: Learning curve for the naive basic block coverage approach

Figure 5.4 and Figure 5.5 show the learning curves for the naive line coverage model and the naive basic block coverage model respectively. These essentially show the same behavior, where it appears that the model would benefit from having even more data. However, the curve seems to be flattening out towards the end. While it appears that more data would increase the performance, the data used to train the model is already considered on the high side for decision trees.

Finally, we compare the feature importance between the line-based approach and block-based approach. This can give an indication of the internal workings of the model. To generate the following graphs, the mean F-measure decrease from removing a feature is measured. This decrease is measured for each of the features, and then collected to generate a bar chart showing all feature importances.

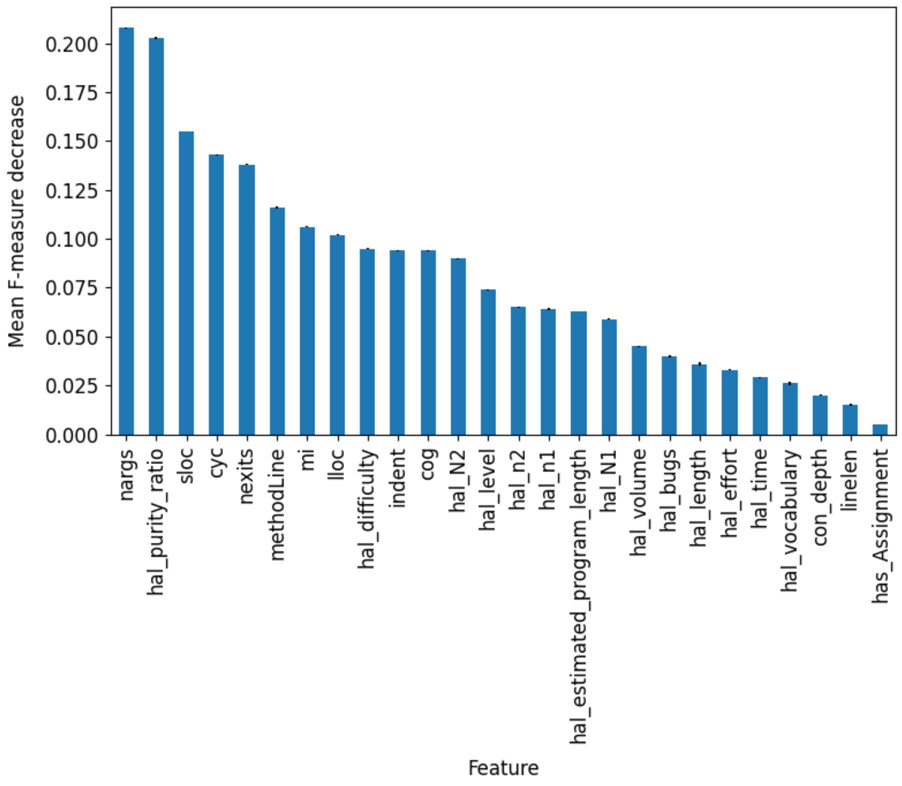


Figure 5.6: Feature importance for naive lines approach

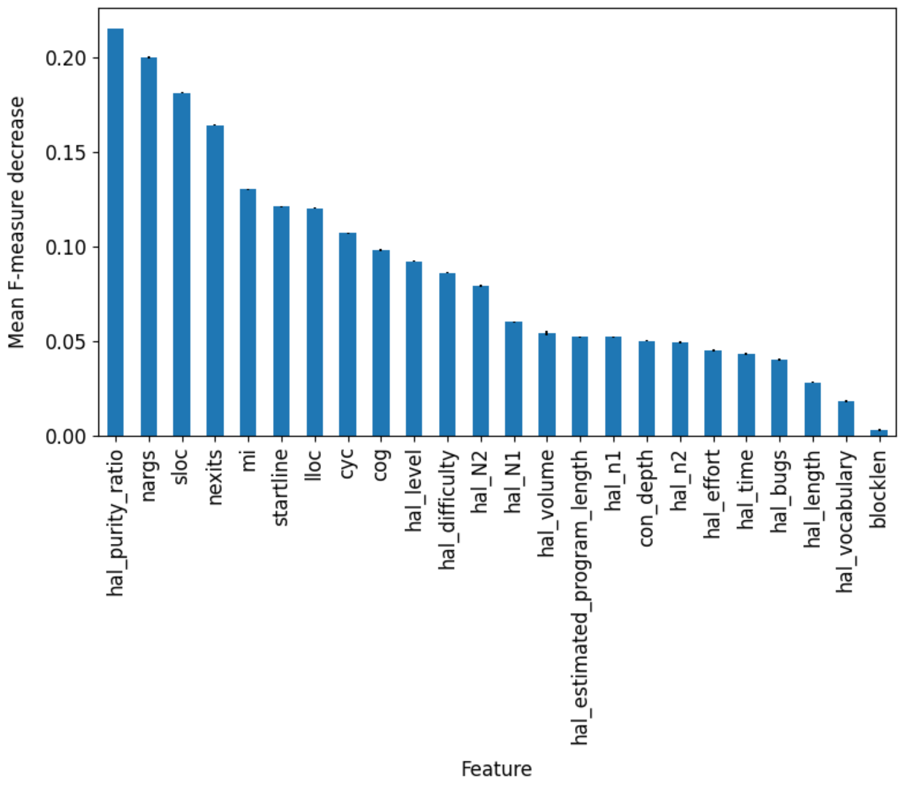


Figure 5.7: Feature importance for naive blocks approach

Figures 5.6 and 5.7 show the feature importance of the various metrics used in the training of the model. In figure 5.12 we see the highest performing metrics being the number of arguments, and the halstead purity ratio each having a mean F-measure decrease of around 0.2. Then the other metrics have a decreasing influence with line length and the line containing an assignment ranking lowest. Meanwhile in figure 5.13, we see a similar story. Once again the number of arguments and Halstead purity ratio rank highest, once again with mean F-measure decrease around 0.2. However, this time halstead purity ratio ranks higher. Similarly as line length for naive lines, block length ranks lowest for naive blocks. Other than number of arguments, and halstead purity ratio, the same metrics appear in the high scoring metrics for both approaches, though cyclomatic complexity scores significantly lower for the basic block approach.
The similarity between these two charts show that essentially the same metrics are important to both approaches. This loosely indicates a similar underlying logic being used to predict the coverage information.

5.2. Naive vs File split

In this section we compare the performance of the naive line-based approach with the file-splitting line-based approach. This is done to answer the question:

RQ2: How does splitting on files influence the performance the model?

To answer this two separate decision tree models were trained on the same dataset. However, with the naive line-based approach, all rows of the dataset are randomly split within training set and test set. Meanwhile in the file-splitting approach, the rows are first grouped based on files, and then split across the training and test set. For each of the experiments we use 10-fold cross validation, and the same set of hyperparameters to ensure we can compare the results fairly.

To compare the naive lines approach with the file-splitting lines approach, we use the same method as the previous section. Once again we use 10-fold cross-validation. For the naive approach this is straightforward, however, for the file-splitting approach a bit more work is required. For this approach we need to supply custom folds which leave lines from different files separate from each other. To do this we first split the files across the folds, and then supply the lines with the corresponding metrics. This does mean that there is a sizable difference between the sizes of the folds. Performing the cross-fold validation on the two approaches gave us the following bar chart:

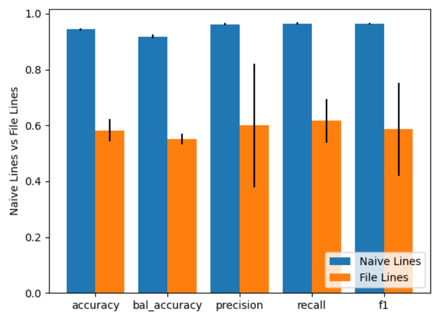


Figure 5.8: Bar Chart showing classification metrics of the naive splitting method vs the file splitting method

Looking at Figure 5.8, we can see that the naive method appears to outperform the file splitting method on all metrics. All the metrics on the naive approach score around the high 90%s, while the File splitting method performs around the low 60%s. My inclination regarding this is that at the very least, coverage does not generalize well outside of the files in which the code is found. Also notable is that the naive deviation for the file splitting approach is significantly higher for precision and f1. However, if the model needs data from every file to generalize well, the use case becomes a lot more limited.

Now we need to compare the validation curve between the naive line-based approach and the file-splitting line-based approach. Since we already have the validation curve for the naive approach from the previous section, we only have to generate it for the file-splitting approach. To do this we get the training and test accuracies for varying max depths using the file-splitting folds again. The results of this can be seen in Figure 5.9 and Figure 5.10.

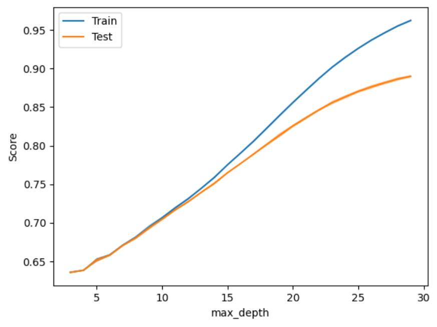


Figure 5.9: Validation Curve for naive approach

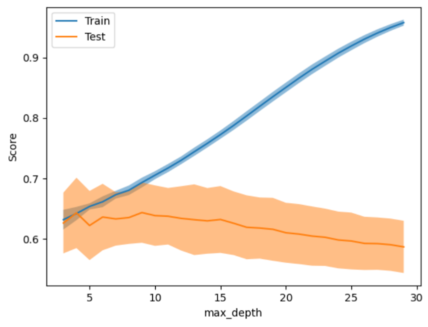


Figure 5.10: Validation Curve for the file splitting approach

The validation curve for the naive approach in Figure 5.9 shows that the training accuracy and the test accuracy diverge from max depth approximately 11. This is an indication that the model overfits for higher values of max depth. It is well-known that decision trees often overfit if you do not bound the tree depth. As such this result is not surprising. Figure 5.10 shows that with increasing max depth the model does not increase. In fact, there appears to be a downward trend based on max depth increasing. This appears once again to indicate that the model can not generalize between files.

In fact a similar story can be seen in the learning curve of the file lines approach. This is shown in Figure 5.11, which was created using the same method as the other learning curves, except using the file-splitting folds.

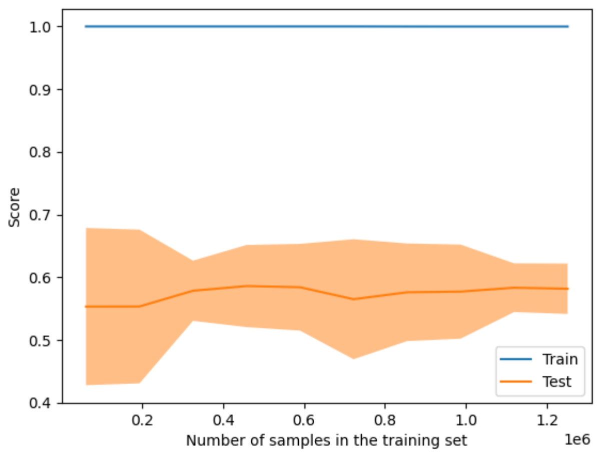


Figure 5.11: Learning Curve for the File Splitting approach

What can be seen is that with an increase in training samples the model barely increases in performance. Also, the range of values seems quite large, which also seems to indicate that most correlation found is likely to be more coincidental. Overall it seems to be important to include lines from within files to be able to make any kind of useful prediction. Our hypothesis is that this is due to the fact that different files are managed by different teams. As such, it is likely that the coding practices differ between teams, and as a result, it is not possible to generalize between these behaviors.

Seeing as the prediction capacity of the file-splitting approach was this minimal, we considered no real reason to look at the feature importance. The feature importance is interesting if it accurately predicts the results, because then we can conclude which metrics are most valuable.

5.3. Feature importance

In this section we compare the importance of the metrics used during the training of the decision trees. This is done to answer the question:

RQ3: Which metrics are most influential in terms of predicting coverage?

To answer this we reused the trained decision trees from the previous sections. Once again, 10-fold cross validation was used to generate these results.

The following figures were taken from Section 5.1. These were created by taking the mean F-Measure decrease from leaving certain metrics out of consideration, and plotting them in descending order. These give us the following feature importance charts.

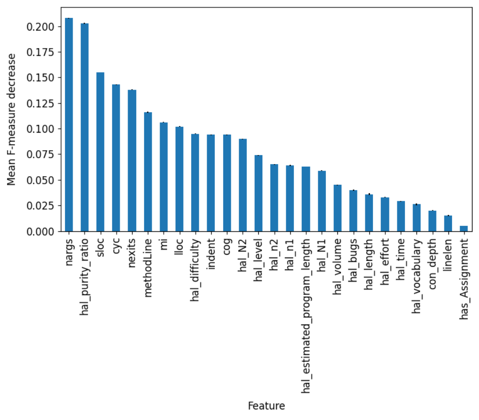


Figure 5.12: Feature importance chart for naive lines

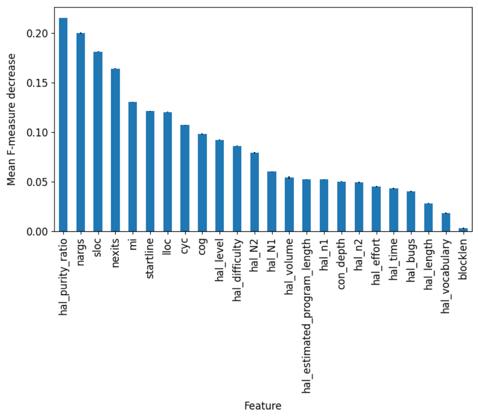


Figure 5.13: Feature importance chart for naive blocks

The best performing metrics here are number of arguments, and the halstead purity ratio. These have a mean decrease in F-measure of around 0.2 for both models. If we were to try and draw conclusions from this, it may make sense that high argument functions are less likely to be covered.
To investigate this we can look at the Kernel Density Estimation plot (KDEplot) for the number of arguments. This plot shows the distribution of differing values of metrics. The KDEplot for the number of arguments can be seen in Figure 5.14.

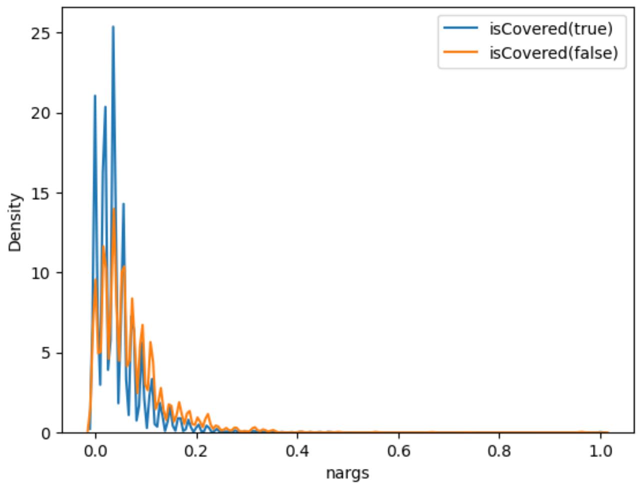


Figure 5.14: Kdeplot for number of arguments

Looking at the plot our suspicions are confirmed. This shows that for lower number of arguments, lines are more likely to be covered. Meanwhile, for larger number of arguments, lines are less likely to be covered. While this is not sufficient to make predictions on separately, we do see some correlation here which could be used as a part of the prediction process.

6 Discussion

In this chapter we will summarize the results from Chapter 5, discussing their further implications. Furthermore, we will discuss the threats to validity and future work. The first research question, on the performance of Basic Block coverage representation, is discussed in Section 6.1. The second research question, on the performance of the File-splitting method, is discussed in Section 6.2. The third and final research question, on metric importance, is discussed in Section 6.3. Section 6.4 discusses the threats to validity. Finally, Section 6.5 discusses future work.

As a means to an end, we set out to answer the following research questions in this thesis:

1. How does representing coverage as block-based instead of line-based influence the performance of the coverage prediction model?
2. How does splitting on files influence the performance the model?
3. Which metrics are most influential in terms of predicting coverage?

In this section we will interpret the results of the previous section. In doing so we will answer these questions.

6.1. Lines vs Basic Block (RQ1)

To summarize the results for the comparison between the Lines approach and the basic block approach, we saw a decrease in performance for the basic block approach. Meanwhile, the feature importance shows that similar metrics play a major role in the prediction. Our hypothesis is that the decrease in performance likely is not caused by basic blocks being more difficult to predict. The basic blocks essentially had access to the same high-performing metrics, yet the performance still was worse. We hypothesize that the line-based approach is memorizing more from lines within basic blocks.
At the same time looking at the difference between the training accuracy and the test accuracy, there is a similar difference. As such the basic block approach also does not lead to a decrease in the overfit.

To answer the research question, the basic block approach with the same metrics does lead to a decrease in performance. Since the overfit remains unaffected as well, we see no convincing reason in our analyses to use the basic block approach over the line-based approach.

6.2. Naive vs File split (RQ2)

In terms of results for this research question we notice a major difference in performance going from the naive approach to the file-based approach. This decrease in performance is in some cases worse than assigning all samples to covered. We also notice a distinct downward trend in terms of performance based on increasing max depth. This decrease in performance indicates that the model overfits more for higher max depth. Which is common for decision trees. This appears to indicate that with the metrics used, the model does not generalize well between files. This could also call into question the results of the naive approach, indicating that the naive approach gets access to training samples it should not.

Overall we notice a severe decrease in performance. This is nearly to the point where there is no increase in performance over naively assigning all samples to true. This has further reaching implications where data from other files can not be used to predict coverage on the current file.

6.3. Feature importance (RQ3)

For the most important metrics we saw that the number of arguments and the halstead purity ratio are most informative. Looking at the plot kdeplot of the number of arguments, we find that for higher number of arguments, there is a lower chance for the line to be covered. This makes some intuitive sense, as the number of arguments does increase the complexity of making an easy test case. Especially if we assume that each argument can change the execution path. The purity ratio gives an indication of how long a piece of code is compared to the estimated length. A lower purity ratio indicates that the program is longer than expected. Once again giving an indication for complexity.

To answer the question which metrics are most influential, we see that metrics which give indications on the complexity of the corresponding code are most influential regarding the prediction of code coverage.

6.4. Threats to validity

To be rigorous in our results we need to be able to admit where we might fall short in terms of our results. As such we identified three main aspects that could be a threat to validity.
The first is regarding the generalizability of our conclusions. Despite using a large dataset, we were limited to only data from Mozilla, and C++ files. As such, certain conclusions might not extend to other projects or programming languages. For instance, it is possible that in other organizations with a more strict/ uniform testing approach, that the model would be able to generalize between files.
The second point regards if we have sufficient support for our conclusion. This is in particular regarding the file-splitting method, where we see that the model does not generalize between files. It is possible that the scale at which we looked might have been to big. Because of the varying level of coverage between directories, it might be possible that the model can generalize between files within directories. As we have not researched this in this paper, the conclusion might have to be weakened.
Finally in terms of feature importance we essentially have one main test. Certain tests for feature importance have flaws, for example, MDI favors high cardinality features. While permutation importance has less of a bias for this, it is merely an indicator of how important the features are for that particular model. As a result, it may be that certain features are actually more important for the prediction of coverage than our model indicates.

6.5. Future work

One of the bigger disappointments in this thesis was that the generalizability of coverage prediction does not extend across files. While the exact reasons for this are unclear, it may be due to the fact that the metrics do not capture enough of the context of the surrounding code. As such re-attempting this problem using a more context-aware approach might be a worthwhile endeavor. Potentially an LLM might be well-suited for this. The capacity of LLMs to capture context and semantic information might prove useful. So as one aspect for future work, it may be interesting to investigate how LLMs deal with this problem.

Another way to potentially get more context in is to take in aspects of the control flow graph. By capturing the way the program works, we might be able to see how integral functions are to the overall program. This could be extended into some kind of critical path analysis. The hypothesis here is that functions closer to the critical path are more likely to be tested, as they are more likely to be called.

7 Conclusion

In this thesis we discussed the usage of machine learning on the problem of code coverage prediction. This was for the end goal of creating a tool which could help in the prioritization of test effort. To find suitable methods we compared the behavior of the approach adjusted from the paper of Brandt and Ramírez [4] with approaches which changed the preparation of the data. This approach involved training a machine learning model to coverage data. To do this, we gathered a dataset of C/C++ files from Mozilla Firefox, filtering the unusable files out. The first adjusted method was to group lines into basic blocks before applying machine learning. For this approach we noted that this leads to a decrease in performance, likely due to less memorization occurring. The second approach was to keep lines from a file in the training set, exclusive to the training set, and similarly for the test set. This led to an even more sever decrease in performance, leading to a conclusion that with the set of metrics used the trends do not generalize between files. Finally we discussed the importance of the metrics used for the model. For this we found that metrics which give an indication of complexity are most informative. With key among these are the number of arguments of used in functions in the piece of code. And also the purity ratio, which gives a ratio between the estimated length and true length of a piece of code. In terms of contributions, this thesis has provided a dataset which can be used for coverage prediction. Furthermore, block-based coverage, a new method of representing coverage for the developer coverage prediction problem has been introduced. Next to that, insights have been provided into the usage of basic blocks, and the generalization of coverage trends between files.

References

[1] Sebastian Avila. *Council post: The hidden cost of bad software practices: Why talent and engineering standards matter*. Mar. 2025. url: https://www.forbes.com/councils/forbestechcouncil/2025/03/28/the-hidden-cost-of-bad-software-practices-why-talent-and-engineering-standards-matter/.

[2] Aydin Kaya et al. "Chapter 6 - Model analytics for defect prediction based on design-level metrics and sampling techniques". In: *Model Management and Analytics for Large Scale Systems*. Ed. by Bedir Tekinerdogan et al. Academic Press, 2020, pp. 125–139. isbn: 978-0-12-816649-9. doi: https://doi.org/10.1016/B978-0-12-816649-9.00015-6. url: https://www.sciencedirect.com/science/article/pii/B9780128166499000156.

[3] Marko Ivankovic et al. "Productive Coverage: Improving the Actionability of Code Coverage". In: *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '24. Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 58–68. isbn: 9798400705014. doi: 10.1145/3639477.3639733. url: https://doi.org/10.1145/3639477.3639733.

[4] Carolin Brandt and Aurora Ramírez. "Towards Refined Code Coverage: A New Predictive Problem in Software Testing". In: *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2025, pp. 613–617. doi: 10.1109/ICST62969.2025.10989028.

[5] Jalaj Pachouly et al. "A systematic literature review on software defect prediction using artificial intelligence: Datasets, Data Validation Methods, Approaches, and Tools". In: *Engineering Applications of Artificial Intelligence* 111 (2022), p. 104773. issn: 0952-1976. doi: https://doi.org/10.1016/j.engappai.2022.104773. url: https://www.sciencedirect.com/science/article/pii/S0952197622000616.

[6] Maurício Aniche. *Effective Software Testing: A developer's guide*. Simon and Schuster, 2022.

[7] Aug. 2020. url: https://testing.googleblog.com/2020/08/code-coverage-best-practices.html.

[8] Hadi Hemmati. "How Effective Are Code Coverage Criteria?" In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. 2015, pp. 151–156. doi: 10.1109/QRS.2015.30.

[9] Marko Ivanković et al. "Code coverage at Google". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 955–963. isbn: 9781450355728. doi: 10.1145/3338906.3340459. url: https://doi.org/10.1145/3338906.3340459.

[10] Rashid Naseem et al. "Investigating Tree Family Machine Learning Techniques for a Predictive System to Unveil Software Defects". In: *Complexity* 2020.1 (2020), p. 6688075. doi: https://doi.org/10.1155/2020/6688075. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1155/2020/6688075. url: https://onlinelibrary.wiley.com/doi/abs/10.1155/2020/6688075.

[11] url: https://gs.statcounter.com/browser-market-share/.

[12] Michele Tufano et al. *Predicting Code Coverage without Execution*. 2023. arXiv: 2307.13383 [cs.SE]. url: https://arxiv.org/abs/2307.13383.

[13] Frances E. Allen. "Control Flow Analysis". In: *ACM SIGPLAN Notices* 5.7 (July 1970), pp. 1–19. doi: 10.1145/390013.808479.

[14] *Basic Blocks in Compiler Design - GeeksforGeeks — geeksforgeeks.org*. https://www.geeksforgeeks.org/basic-blocks-in-compiler-design/. [Accessed 09-05-2025].

[15] Zaheed Mahmood et al. “Reproducibility and replicability of software defect prediction studies”. In: *Information and Software Technology* 99 (2018), pp. 148–163. issn: 0950-5849. doi: `https://doi.org/10.1016/j.infsof.2018.02.003`. url: `https://www.sciencedirect.com/science/article/pii/S0950584917304202`.

A

Basic Block algorithm

```
1  void MaiAtkObject::FireStateChangeEvent(uint64_t aState, bool aEnabled) {
2    auto state = aState;
3    int32_t stateIndex = -1;
4    while (state > 0) {
5      ++stateIndex;
6      state >>= 1;
7    }
8
9    MOZ_ASSERT(
10       stateIndex >= 0 && stateIndex < static_cast<int32_t>(gAtkStateMapLen),
11       "No ATK state for internal state was found");
12   if (stateIndex < 0 || stateIndex >= static_cast<int32_t>(gAtkStateMapLen)) {
13     return;
14   }
15
16   if (gAtkStateMap[stateIndex].atkState != kNone) {
17     MOZ_ASSERT(gAtkStateMap[stateIndex].stateMapEntryType != kNoStateChange,
18                "State changes should not fired for this state");
19
20     if (gAtkStateMap[stateIndex].stateMapEntryType == kMapOpposite) {
21       aEnabled = !aEnabled;
22     }
23
24     // Fire state change for first state if there is one to map
25     atk_object_notify_state_change(&parent, gAtkStateMap[stateIndex].atkState,
26                                    aEnabled);
27   }
```

Using the steps described in section 4.1.2:

- The first line of code is a leader.
- The target of any jump is a leader.
- Any line after a jump (conditional or not) is a leader

We start off by setting line 1 as a leader. Then since the first line of a method can be jumped to, and as such is the target of a jump, we find that line 2 must be a leader as well. A while block can be decomposed into a jump if, body and then a jump back to start again. As such, we find that line 4, line 5, and line 9 must be leaders. Because of the if statement in line 12, there is an implicit jump statement there, as such line 13, and line 16 must be leaders. Similarly, line 17, 21 and 25 must be leaders.