UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

KESLLEY LIMA DA SILVA

# Predicting Prime Path Coverage Using Regression Analysis at Method Level

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Érika Fernandes Cota

Porto Alegre
October 2021

**CIP — CATALOGING-IN-PUBLICATION**

Silva, Keslley Lima da

Predicting Prime Path Coverage Using Regression Analysis at Method Level / Keslley Lima da Silva. – Porto Alegre: PPGC da UFRGS, 2021.

68 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2021. Advisor: Érika Fernandes Cota.

1. Software testing, Coverage prediction, Code coverage, Regression analysis. I. Cota, Érika Fernandes. II. Título.

*“Without ambition one starts nothing. Without work one finishes nothing. The prize will not be sent to you. You have to win it.”*

— Sir Ralph Waldo Emerson

## ABSTRACT

Test coverage criteria help the tester in analyzing the quality of the test suite, especially in an evolving system where it can be used to guide the prioritization of regression tests and the testing effort of new code. However, coverage analysis of more powerful criteria such as path coverage is still challenging due to the lack of supporting tools. As a consequence, the tester evaluates a test suite quality employing more basic coverage criteria (e.g., node coverage and edge coverage), which are the ones that are supported by tools. In this work, we evaluate the opportunity of using machine learning algorithms to estimate the prime-path coverage of a test suite at the method level. We followed the Knowledge Discovery in Database process and a dataset built from 9 real-world projects to devise three regression models for prime-path prediction. We compare four different machine learning algorithms and conduct a fine-grained feature analysis to investigate the factors that most impact the prediction accuracy. Our experimental results show that a suitable predictive model uses as input data only five source code metrics and one basic test coverage metric. Our evaluation shows that the best model achieves an MAE of 0.016 (1,6%) on the cross-validation (internal validation) and an MAE of 0.06 (6%) on the external validation. Finally, we observed that good prediction models can be generated from common code metrics although the use of a simple test metric such as branch coverage can improve even more the prediction performance of the model.



**Keywords:** Software testing, Coverage prediction, Code coverage, Regression analysis.

## RESUMO

Os critérios de cobertura de teste auxiliam o testador na análise da qualidade do conjunto de testes, em especial em sistemas em evolução onde pode ser utilizado para orientar a priorização dos testes de regressão e o esforço de teste de um novo código. No entanto, a análise da cobertura de critérios mais poderosos, tais como a cobertura de caminhos, continua a ser desafiante devido à falta de ferramentas de apoio. Como consequência, o testador avalia a qualidade de um conjunto de testes utilizando critérios de cobertura mais básicos (por exemplo, cobertura de nós e cobertura de arcos), que são os que são suportados por ferramentas. Neste trabalho, avaliou-se a oportunidade de utilizar algoritmos de aprendizagem de máquina para estimar a cobertura de caminhos primos de um conjunto de testes em nível de método. Seguiu-se o processo de descoberta de conhecimento em base de dados e um conjunto de dados construído a partir de 9 projetos do mundo real para se criarem três modelos de regressão para a previsão do valor de cobertura do critério de caminhos primos a partir de métricas de código. Compararam-se quatro algoritmos diferentes de aprendizagem de máquina e realizou-se uma análise detalhada de características para identificar aquelas que mais afetam o desempenho da predição. Os resultados experimentais mostraram que modelos preditivos de boa acurácia podem ser gerados a partir de um conjunto de métricas de código pequeno e de fácil obtenção. O melhor modelo gerado utiliza como dados de entrada apenas cinco métricas de código fonte e uma métrica básica de cobertura de teste e atinge um MAE de 0,016 (1,6%) na validação cruzada (validação interna) e um MAE de 0,06 (6%) na validação externa. Por fim, observou-se que modelos preditivos adequados podem ser gerados a partir de métricas de código comuns, embora o uso da métrica de cobertura de arcos, quando disponível, possa melhorar ainda mais o desempenho de predição.



**Palavras-chave:** Teste de Software, Predição de Cobertura, Cobertura de Código, Análise de Regressão.

## LIST OF ABBREVIATIONS AND ACRONYMS

| | |
|---|---|
| TR | Test Requirements |
| SUT | Software Under Test |
| NC | Node Coverage |
| EC | Edge Coverage |
| ML | Machine Learning |
| PPC | Prime Path Coverage |
| IDE | Integrated Development Environments |
| CFG | Control Flow Graph |
| TP | Test Path |
| KDD | Knowledge Discovery Databases |
| OOP | Object-Oiented Programming |
| PP | Procedural Programming |
| LOC | Lines of Code |
| STMT | Number of Statements |
| STMTd | Declarative Statements |
| STMTx | Executable Statements |
| CC | McCabe complexity |
| SCC | Strict Cyclomatic Complexity |
| SCM | Source Code Metrics |
| SVR | Support Vector Regression |
| RANN | Regression Artificial Neural Network |
| KNNR | K-nearest Neighbours Regression |
| RFR | Random Forest Regression |
| MAE | Mean Absolute Error |

MSE Mean Square Error

MSLE Mean Squared Logarithmic Error

MedAE Median Absolute Error

## LIST OF FIGURES

## LIST OF TABLES

**CONTENTS**

# 1 INTRODUCTION

Software development is related to the software engineering methodology, which includes various methods and techniques. The software engineering approach in the development process provides high-quality software products at a fair price. One of the software development activities is software testing, estimated at least about half of the entire development cost (HAILPERN; SANTHANAM, 2002). Several challenges are related to the high cost and low efficiency of a testing process, such as redundant testing and insufficient coverage. Despite the high costs, software testing plays a vital role in the software development process for ensuring the software's quality and reliability, especially when catching regression faults (PINTO; SINHA; ORSO, 2012).

Software testing aims to exercise the software's behavior to prevent future bugs by revealing the maximum number of faults using the least amount of resources. For example, when finding defects that cause failures in the developed product, the quality of software products can be improved. Because of its simplicity and practicality, software testing has become one of the main software quality assurance techniques. However, measuring the quality of testing is difficult because its effectiveness depends on the test suite's quality: some suites are better at detecting faults than others. A test suite's power refers to its potential to detect faults: a strong-power suite can detect more bugs than a weak-power one (Zhang et al., 2019). The test coverage level is the degree to which the specified coverage elements have been exercised through a series of tests. Consequently, for an effective testing, the notion of test coverage criteria is important because it provides a means of measuring the extent to which a set of test cases exercises a program. In common practice, test coverage criteria indicate this power by mostly exposing weaknesses of the test suite.

A test coverage criterion is a rule or group of rules that describes the test requirements (TR); each requirement is a specific software element that a test case must satisfy or cover. A criterion is defined for a structure that represents the object under test. Ammann and Offutt (AMMANN; OFFUTT, 2016) compile several criteria defined in the literature into a few ones defined over four generic structures that can be used to represent (or model) a software under test (SUT), i.e., input domain, graphs, logic expressions, and grammars. The most popular test structure is the graph, and graph-based criteria require the tester to cover the graph somehow. The most common graph-based criteria are node coverage (NC) and edge coverage (EC), which can be measured by many testing

tools available for most programming languages. These two criteria can be considered trivial and generate the minimum test requirements for a test suite. In contrast, there are other criteria capable of exploring more intricate uses of the software, such as the criteria that cover paths. Information about test coverage helps the development team to find weaknesses and redundancies in the current test suite. It also provides useful rules and supporting metrics for when to stop testing. Thus, the adoption of appropriate coverage criteria has several advantages for improving software quality and controlling the cost of testing.

## 1.1 Problems in code coverage criteria

Despite the mentioned benefits, there are difficulties related to the measurement of more powerful criteria. For instance, to assess the graph-based coverage of a test suite, one needs to first devise the graph that represents the SUT, then extract the test requirements for a specific criterion, and then trace the execution path on this graph given by each test case. Test requirements covered by the traced execution paths are counted and, at the end of this process, the coverage for that criterion can be calculated.

The manual execution of this process is typically too expensive, whereas its automation has been restricted to basically node and edge (branch) coverage of source-code. Currently, this process is strongly dependent on the technology as it requires knowledge and interaction with the compiling and execution environment of the SUT. Therefore, any change on the programming language development or execution environment impacts the corresponding test coverage evaluation tool and a great deal of effort is required just to keep up with the natural evolution of the programming environment. Moreover, the vast majority of available tools are limited to node and edge criteria which, as mentioned, also provide limited help in assessing the quality of the test suite.

Durelli et al. (DURELLI; DELAMARO; OFFUTT, 2018) compare the effectiveness and cost of three structural graph coverage criteria: edge coverage, edge-pair coverage, and prime path coverage (PPC). As a result, they inferred that PPC is more effective, especially in programs that have complicated control flows. In terms of cost, they concluded that there is not much difference in terms of the number of TRs, but PPC leads to much more infeasible TRs than EC and edge-pair coverage.

In order to mitigate the challenges of technology-dependent tools and help the testers to improve their test suites, recent work has proposed the estimation of the cov-

erage value rather than its exact calculation. For instance, machine learning (ML) algorithms are used to predict the mutation score (STRUG; STRUG, 2012; STRUG; STRUG, 2016; Strug; Strug, 2017; JALBERT; BRADBURY, 2012; Grano; Palomba; Gall, 2019; Zhang et al., 2019) and the edge coverage (GRANO et al., 2019) achieved by test-data generation tools.

Given this context, we consider PPC as a valuable criterion because 1) it subsumes most graph-based criteria, including data-flow ones (AMMANN; OFFUTT, 2016); 2) provides more accurate information than basic criteria (e.g., EC and NC); and 3) allows the tester to evaluate the reach and omissions of the test suite with more conviction. Still, nowadays, there are no practical tools to support the tester to evaluate the test suite based on this criterion. We believe there are two main reasons for that. First, to build and maintain a test coverage tool using PPC criterion is challenging due to the required technological dependency of Integrated Development Environments (IDEs) and third-party tools. Second, since few works show practical evidence about the advantages of using PPC criterion, testers believe weaker criteria (the ones supported by available tools) are enough and the culture for its usage is not created. Thus, in this work we investigate the use of regression analysis algorithms to predict the PPC value and aim at provide a technology-independent tool for testers to evaluate the potential benefits of this coverage criterium.

## 1.2 Proposed Research and Contributions

This work tackles the two main issues that seems to preclude the usage of PPC as a coverage criterium in industry: the lack of coverage analysis tools that consider PPC and the lack of evidence of the cost-effectiveness of this criterium in industrial settings. To this end, we propose the use of ML algorithms to predict the PPC value of a test suite to tackle the problem of technology dependency that limits the construction of PPC coverage analysis tools. Then, we provide suitable predictors that can be used as a tool to evaluate the benefits of PPC in industrial settings. The proposed predictors use as input a set of code metrics and, if available, the basic edge coverage metric. The metrics used as input are extracted at method level from the source code, since coverage analysis is usually applied together with unit testing.

The contributions of this work are detailed below.

- It analyses the relationship between source code metrics and PPC of a test suite at the method level;
- it proposes a methodology to support the tester to evaluate the power of a test suite by using a prediction model and a more strong criterion;
- It validates the proposed methodology by performing experiments to evaluate the predictive model in a realistic scenario;
- It shows that PPC can be estimated using source-code metrics and basic test coverage metrics and provides the devised predictive model;
- It provides a public dataset that involves PPC criterion in open source projects, making it available online.
- From the tester's point of view, we deliver as a solution different predictors, which are technology independent, and that can help evaluating the cost-effectiveness of PPC.

## 1.3 Outline

This dissertation is structured as follows: Chapter 2 reviews the main concepts necessary to understand the approach of this work. In Chapter 3, we present our approach (the methodology and implementation) that aims to generate a predictive model from source code metrics to estimate the PPC value of a test suite at method level. Chapter 4 describes the methodology to validate the feasibility of using our model. After that, Chapter 5 describes the experimental results, showing the model's performance in different scenarios, and also discusses some threats to validity. Chapter 6 discusses related work. Finally, we provide, in Chapter 7, the conclusions of our investigations.

# 2 BACKGROUND

## 2.1 Code Coverage

Software testing is a broad term that embraces a wide variety of different activities, from testing a small piece of code by the developer (unit testing) to the customer validation of a system (acceptance testing). Despite this variety of activities, software testing can be described as the validation and verification process for software to meet business, technical and functional requirements. To this end, a wide range of software testing techniques and strategies are used. The most common techniques are the black box and the white box.

In black-box testing, the tester concentrates on what the software does, i.e., they will not focus on how (internal logic structure) the software does it because the source-code structures are unknown and not considered by the tester (MALL, 2018). On the other hand, the white box, also known as structural testing, is a technique where test cases are designed on source-code information (LIU; TAN, 2009). In the white box technique, the tester, usually the code developer, knows the source code structure and writes test cases to verify small parts of the code (unit testing). As this technique is related to the internal structure, it essentially focuses on the program's control flow or data flow.

Code coverage criteria are often employed as goals to be achieved in white box testing. With this purpose, the set of test cases (i.e., test suite) is improved and increased until the required coverage level has been reached according to the chosen criteria. The literature proposes different testing coverage criteria, but graph-based ones are undoubtedly the most popular.

Graph-based coverage criteria are divided into two types: control flow and data flow. These criteria are applied to directed graphs that represent a SUT and resulting test requirements are sub-paths of length 0 (node coverage), 1 (edge coverage), 2 (edge-pair coverage), or more (path coverage) in this graph. When applied to source-code, the directed graph is called Control Flow Graph (CFG), where a node represents an indivisible piece of code (basic block), and an edge represents a possible control flow between two nodes. In software testing, a path in the CFG that starts at the initial node and ends at a final node is called a test path and represents a specific test case (AMMANN; OFFUTT, 2016).

This work focuses on prime path coverage, which is a control flow coverage cri-

terion. The PPC criterion is based on the definition of a simple path. A simple path is a path in which no node appears more than once (no internal loops), and only starting and ending nodes can appear more than once (AMMANN; OFFUTT, 2016). A prime path is a simple path that is not a sub-path of any other simple path (Li; Praphamontripong; Offutt, 2009). Then, the PPC criterion defines as test requirements the set of all prime paths of the graph.

A test path (TP) is a path that starts at an initial node and ends at a final node, representing the execution of test cases. A test path covers a test requirement when the later (a prime path) is a sub-path of the former. PPC has two distinct cases that involve the treatment of loops with round trips: Simple Round Trip Coverage criterion and Complete Round Trip Coverage criterion. A round trip path is a prime path of nonzero length that starts and ends at the same node (AMMANN; OFFUTT, 2016). The first criterion requires at least one round trip path for each reachable node, whereas the second requires all round trip paths for each reachable node.

Because of the treatment of loops in PPC, the test requirements can be covered in three forms (AMMANN; OFFUTT, 2016): 1) directly — i.e., each node and edge in the test path must be visited precisely in the order that they appear in the TR; 2) with side-trips, in which every edge in TR is also in a test path in the same order; and 3) with detours, where every node in TR is also in a test path in the same order. For this work, we consider only the directly coverage form.

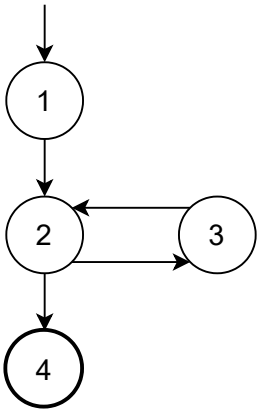
To better understand the PPC criterion, let us consider the graph shown in Figure 2.1 that illustrates the flow of a simple source code unit (e.g., method). For a more helpful example, we use the following notation (AMMANN; OFFUTT, 2016):

*****: denotes that the simple path is a loop and should not be further extended because any extension would make that path no longer simple.

**!**: denotes that the simple path cannot be further extended. The two possible deductions for this are that either it was reached a final node or it would be the same case as the situation above.

To extract all prime paths from a graph, we start by identifying all simple paths in that CFG (Figure 2.1).To this end, we need to find all sub-paths of length 0, 1 and 2. Table 2.1 shows the simple paths that make up the test requirements needed to satisfy PPC for the given CFG. Therefore, the set of TR is [1, 2, 3], [1, 2, 4], [2, 3, 2], [3, 2, 3] and [3, 2, 4]. As mentioned in Section 1.1, the coverage value for a criterion can be calculated by

Figure 2.1: An example of a simple CFG.




Source: The author

Table 2.1: The set of sub-paths

| | |
|---|---|
| **Length 0** | {[1], [2], [3], [4]} |
| **Length 1** | {[1, 2], [2, 3], [2, 4]!, [3, 2]} |
| **Length 2** | {[1, 2, 3]!, [1, 2, 4]!, [2, 3, 2]*,[3, 2, 3]*, [3, 2, 4]!} |

Source: The author

comparing the set of test paths (i.e., the execution path of test cases on the CFG) with the set of test requirements. For instance, based on the scenario described in Table 2.2, the PPC value of test suite 1 is 80% because it covers four out the five prime paths. Test suite 2, on the other hand, achieves PPC coverage value of 100% because it covers all prime paths in the TR set.

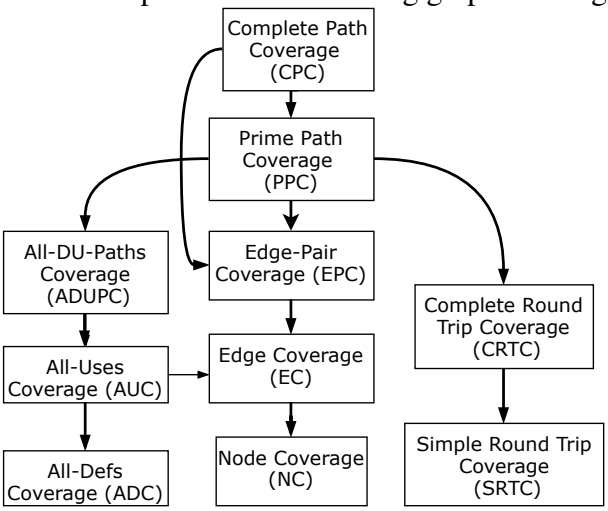
Table 2.2: The set of TR and possible corresponding test suites

| **TR of PPC** | **Suite 1** | **Suite 2** |
|---|---|---|
| [1, 2, 3], [1, 2, 4], [2, 3, 2], | [1,2,4] | [1,2,4] |
| [3, 2, 3], [3, 2, 4] | [1,2,3,2,4] | [1,2,3,2,3,2,4] |

Source: The author

The test requirements generated by the PPC criterion represent less trivial uses of a code-unit and are especially interesting in high complexity SUTs. According to (AMMANN; OFFUTT, 2016), in practical situations, PPC criterion subsumes node and edge coverage criteria, as well as data-flow criteria. It is subsumed only by the complete path coverage criterion, which is the strongest one in graph coverage and defines all possible paths of the graph as test requirements being, therefore, impractical in most cases. An important consideration is that the PPC does not subsume edge-pair coverage. This is due to the case when a node has a self-loop. The ten structural criteria defined in Ammann & Offutt's book (AMMANN; OFFUTT, 2016) are shown in Figure 2.2, with edges expressing subsumption relationships.

Figure 2.2: Subsumption relations among graph coverage criteria




Source: Adapted from (AMMANN; OFFUTT, 2016)

An infeasible path in software testing context can be defined as the path that no set of possible input values can verify. Since all graph coverage criteria only use the properties of a CFG to determine a set of TRs, it is not possible to guarantee that all TRs in that set are feasible. Infeasible paths in a set of TR influence the coverage value since it is logically impossible to cover an infeasible path. Therefore, to deal with this, it is necessary to perform an analysis to remove all existing infeasible paths in a set of TRs. This analysis is beyond the scope of this work and we assume that the set of TRs of a given method does not contain infeasible paths.

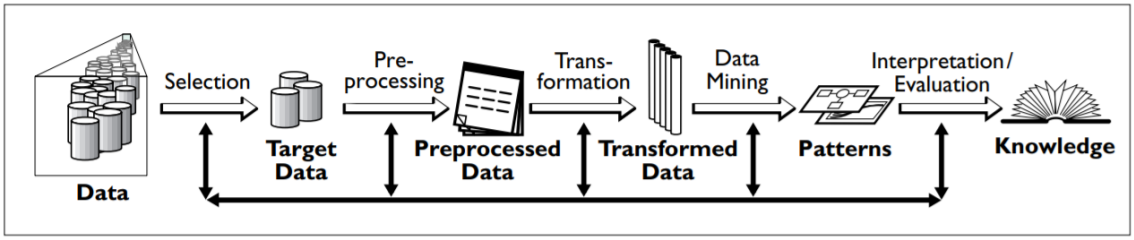
## 2.2 Knowledge discovery in databases

Fast advances in data collection and storage tools have allowed organizations to accumulate vast amounts of data daily. However, extracting useful information has proved to be notably challenging. In this context, the data mining field becomes very important for different industries and businesses because of its capacity to use a vast volume of data that had previously no use and make analysis and predicting trends and patterns (CHEN; HAN; YU, 1996). With this field's emergence, different process models were proposed, guiding and carrying the data mining tasks and their applications. The most popular data mining process models are CRISP-DM, SEMMA, and Knowledge Discovery Databases (KDD) process. In this work, we follow the KDD process model because it is a model that has its stages very well defined, providing better support throughout our data mining activities, which aims to create a predictive model to estimate the PPC value at the method

level.

The KDD process model is proposed as a set of techniques to support data exploration and analysis, aiming to identify valid, novel, potentially useful, and finally understandable patterns in data (FAYYAD; PIATETSKY-SHAPIRO; SMYTH, 1996). The KDD process overview is outlined in Figure 2.3. This process model can involve significant iteration and contain loops between any two phases (TAN; STEINBACH; KUMAR, 2016). The model consists of nine different steps, which are presented below.

Figure 2.3: Overview of the KDD process model




Source: (FAYYAD; PIATETSKY-SHAPIRO; SMYTH, 1996)

**1- Goal-Setting and Application Understanding:** this is the first stage of the process and requires a prior understanding of the subject matter to be applied. In this step, the goals are determined from the end-user viewpoint and used to develop and understand the application domain and its prior knowledge. It develops the premises for understanding what should be done with the different decisions in the further steps, such as transformation, cleaning, and mining algorithms of choice.

**2- Data Selection and Integration:** this is the second stage of the KDD process that targets the dataset and subset of data samples or variables/metrics. Once defined the objectives (Step 1), the data needs to be selected into meaningful sets based on the following parameters: importance, availability, accessibility and quality. These parameters are critical for data mining activities because they make the base for it — i.e., the evidence base for building the models — and will influence which models are formed. All the mentioned parameters were considered when determining the projects and metrics that make up our dataset.

**3- Data Cleaning and Preprocessing:** generally, the collected data is not clean, containing missing values, errors, and noisy data. As a result, the reliability and efficiency of the set of data is improved at this stage, emphasizing cleaning and preprocessing the target data to produce a noise-free and consistent dataset. Different strategies and algorithms can be applied for handling missing data fields, noises, and outliers during this step.

**4- Data Transformation:** in some data mining projects, the data, even after cleaning, is not ready for mining tasks. Therefore, at this point, suitable data for data mining is prepared and developed. It focuses on transforming data from one form to another to perform data mining algorithms efficiently. For this purpose, many data reduction approaches can be implemented on the target data, such as feature selection, record sampling, and extraction. Complementary to that, data transformation techniques can be used, for example, functional transformation and discretization of numerical attributes.

**5- Choosing the data mining task:** this is the fifth stage of the KDD process in which it is defined which kind of data mining task to use. As a multidisciplinary field, the data mining phase uses many areas; one of these is machine learning, which refers to the automated detection of significant patterns in data (KIRK, 2014). A primary notion in ML is the difference between supervised and unsupervised learning. There is no distinction between training and test data in unsupervised learning, whereas, in supervised learning, there are examples for the learner (training process). In this work we want to evaluate whether a supervised learning process can generate a good predictive model for PPC criterion.

Data mining tasks are divided into predictive tasks and descriptive tasks. The goal of the predictive task is to use supervised learning to predict a value (dependent variable) based on the values of other attributes (independent variables). In descriptive tasks, the goal is to use unsupervised learning to derive data patterns, such as clusters, anomalies, and relationships. There are two types of predictive tasks: classification, which is used to predict categorical labels, such as "yes" or "no"; and regression, which is widely used for numeric prediction.

**6- Selecting the Data Mining algorithm:** this is the sixth step of the KDD process in which one or more fitting data mining algorithms are selected for building the model. In this work, the goal is to build a regression model. There is more than one way to apply a regression analysis. Consequently, there are many regression algorithms proposed in the literature, such as random forest (BREIMAN, 2001), logistic regression (HOSMER; LEMESHOW; STURDIVANT, 2013), support vector machine (CHANG; LIN, 2011), among others. Each algorithm has parameters and strategies for learning, such as K-fold cross-validation (STONE, 1974) or another division for training and testing.

**7- Utilizing the Data Mining algorithm:** this is the seventh step of the KDD process in which selected algorithms are implemented. In this stage, some data mining projects may need to employ and modify the algorithm parameters many times until a

satisfying outcome is obtained.

**8- Pattern Evaluation:** in this step, the goal is to evaluate and interpret the mined patterns with respect to the goals defined in the first step. This step focuses on the comprehensibility and utility of the produced model. This step may involve pattern extraction and visualization, for example, representing in discrete forms such as pie charts, graphs, and histograms.

**9- Knowledge Discovery and Use:** this is the ninth and final step of the KDD process in which the discovered knowledge is used for different purposes. In this work, the final result of the process, the predictive model, will support the tester, estimating the PPC value for any test suite at the method level.

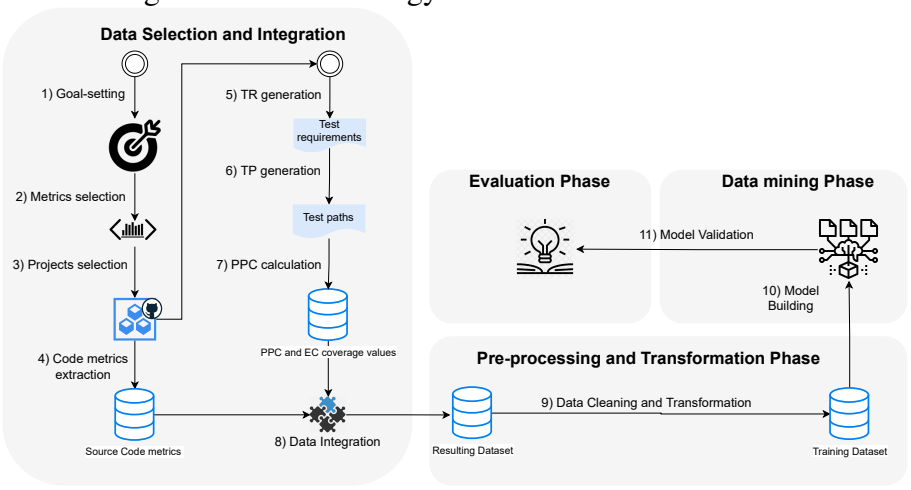
# 3 PROPOSED APPROACH

In this chapter, we present an ML-based solution that follows the KDD process detailed in the Section 2.2. Our approach aims to link information derived from source code metrics and basic test code coverage criterion to predict the PPC value of a test suite at the method level. A first version of this approach is presented at (SILVA; COTA, 2020) where the PPC value is predicted from a single metric. Based on the obtained results, we have improved that approach by implementing a detailed data selection analysis. Section 3.1 describes the proposed methodology and Section 3.2 details the activities and challenges for the implementation of the first part of the proposed approach, i.e., the dataset construction. The implementation of the data mining and evaluation steps of the proposed methodology are detailed in Chapter 5.

## 3.1 Methodology

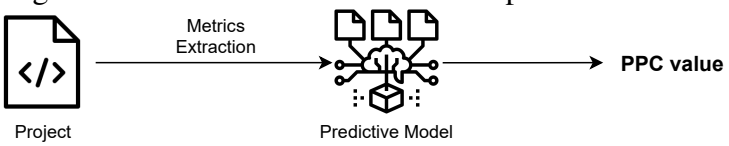
The KDD process is a generic method for mining a large volume of data with a certain goal in mind. In this section we describe how each step of the KDD is applied to our goal of devising a predictive model for PPC estimation. Figure 3.1 and Figure 3.2 provide an overview of the steps required to, respectively, generate and use the PPC predictor model. The general steps of the KDD process (data selection, integration, preprocessing, transformation, mining, and model evaluation) can be identified in Figure 3.1. However, due to the nature of the problem in hand, the actual data is not originally available to be used. Thus, as we detail next, Step 2 of the KDD process needs to be adapted to first generate the data to be analyzed. On the other hand, since the data generation process is a more controlled one, Steps 3 and 4 of data cleaning, preprocessing, and transformation can be simplified.

Figure 3.1: Methodology for the model construction.




Source: The author

Figure 3.2: Overview of the use of our predictive model.




Source: The author

### 3.1.1 Goal setting

As mentioned, the goal of this mining process is to devise a predictor model for PPC to be used as depicted in Figure 3.2. Thus, we want to identify the best regression model for a dataset composed of source code and basic test metrics as independent variables and PPC as the dependent variable.

### 3.1.2 Data selection

Following the KDD process, after a well-defined solution purpose, the selection phase encompasses the definition of the source code and test metrics that will form the training set. This step needs to be carefully considered to be aligned with the defined goals. To apply one of the graph coverage criteria, the first step is to define a graph abstraction of the source code (i.e., CFG). A CFG of a class is a set of CFGs of its methods. Since our goal is to evaluate code coverage in unit test, we decided to predict method-level ppc value.

A mapping study conducted by Nunuez-Varela et al. in (VARELA et al., 2017) present almost 300 source code metrics and the most applied code metrics tools collected from 226 studies published between the years 2010 and 2015. Based on that study, we defined two primary filters to determine the initial set of metrics:

1. **Metrics with the highest number of occurrences:** first, we consider only the two most relevant programming paradigms: object-oriented programming (OOP) and procedural programming (PP). A total of 194 different metrics were found for the OOP paradigm and 16 metrics for the PP paradigm. Despite many existing metrics for these two paradigms (210 metrics), the minority are method-level metrics. Since we intend to choose only metrics that describe method-level features, all non-method-level metrics (e.g., class, package) are not considered for a detailed analysis.
2. **Software metric tools with the highest occurrences:** source-code tools influence our feature selection process because our goal is that the chosen set of metrics is ideally extractable by one or more widely used, well-known, and free to use tools. For this purpose, we consider for investigation the tools presented in Table 3.1, ordered by the highest number of occurrences in the studies. For each tool, we analyze if they are operational and which metrics they can extract.

Table 3.1: Software metric tools with highest occurrences

| **Tool** | **Paradigm** | **Version** | **Working tool** | **Open-source** |
|---|---|---|---|---|
| Understand | OOP/PP | 5.1 | Yes | No |
| CKJM | OOP | 1.9 | Yes | Yes |
| Eclipse Metrics | OOP | 1.3.10 | Yes | Yes |
| JHawk | OOP | 6.1.7 | Yes | No |

Source: The Author

A set of 46 method-level source code metrics has been identified using the two above mentioned filters. All of those metrics can be extracted using one or more of the tools listed in Table 3.1. We performed a comprehensive analysis from this set to arrive at the final set of metrics that make up our dataset. The qualitative analysis of this set was based on the understanding of how each metric can represent characteristics of the application method and test method that influence a higher or lower PPC value. For example, the NPATH metric represents complexity of a code-unit since it indicates the number of possible paths. Similar to this metric, PPC is also related to the paths covered

Table 3.2: The set of independent variables organized by feature set.

| Metric | Set |
|---|---|
| Lines of code (LOC) | *A* & *T* |
| Number of statements (STMT) | *A* & *T* |
| Declarative statements (STMTd) | *A* & *T* |
| Executable statements (STMTx) | *A* & *T* |
| NPATH | *A* & *T* |
| NPATHLog | *A* & *T* |
| McCabe complexity (CC) | *A* & *T* |
| Strict cyclomatic complexity (SCC) | *A* & *T* |
| Fan-out | *A* & *T* |
| Fan-in | *A* |
| Knots | *A* |
| MaxNesting | *A* |
| Edge Coverage (EC) | *S* |

Source: The Author

in a unit of code. On the other hand, metrics that evaluate the number of comments in a unit of code do not influence the value of PPC. As a result of this analysis, we selected only 12 source code metrics and one coverage metric (EC value) as our independent variables, as shown in Table 3.2. Besides the common name of the metric, the last column in Table 3.2 indicates to which code unit each metric refers: *A* for application methods, *T* for test methods, *S* for the test suite of a single method (all test methods related to a single application method). Nine source-code metrics describe factors about application methods and test methods (lines marked with *A* & *T* in the table). Three metrics describe factors related only to application methods (lines marked with *A*). One metric describes the EC value obtained by the test set associated to an application method (lines identified as *S*).

Concerning the selected features, source lines of code (LOC) are the number of lines that contain source code in a source code unit (e.g., class or method), ignoring the comments and blank lines. The metric number of statements (STMT) denotes the number of statements in a method. This metric is an alternative way to LOC metric to measure the system size. It can be used to obtain the statement coverage value (i.e., node coverage) of a source code unit by counting the percentage of statements executed by a unit test. Declarative and executable statements are the two types of statements. The important difference between these two metrics is that declarative statements have no immediate effect during execution. Statements that define constant values, data structures, and procedures are all examples of declarative statements.

The cyclomatic complexity (CC) is a measurement of McCabe's code complexity,

which is often considered a magic number to measure a program's complexity. It is a quantitative measure of independent paths in the source code of an application; therefore, the minimum number of paths should be tested (MCCABE, 1976). The more complicated a piece of code is, the more time and effort are required to create and maintain it, and the higher the likelihood of bugs than code with a lower complexity score. There are some variants of CC. One of them is called Strict Cyclomatic Complexity (SCC), in which logical ANDs and ORs in conditional expressions also add 1 to the complexity for each of their occurrences. To give an example, the statement *if ( a && b || c)* would have a CC of 1 but an SCC of 3.

The NPATH metric computes the number of possible acyclic execution paths through a method. This metric was proposed at (NEJMEH, 1988) based on the cyclomatic complexity. Even though it is connected to cyclomatic complexity, the two metrics are not the same. Cyclomatic complexity only counts the number of decision points, while NPATH counts the number of distinct complete paths from the beginning to the end of the method. A practical case that exemplifies the difference between these two metrics, and at the same time the importance of using both, is that with NPATH, two decision points appearing sequentially have their complexity multiplied, whereas the CC is only added. Ten if-else statements in a row, for example, will result in an NPath of 1024. Cyclomatic complexity, on the other hand, will be counted as 20. Methods with an NPATH complexity of more than 200 are generally regarded as excessively complex. Another metric similar to NPATH is NPATHlog, which is calculated based on the base 10 logarithm (Log10x), truncated to an integer value.

The number of modules or methods that call another module or method is known as Fan-in. One method call will affect the fan-in of many other methods due to polymorphism. Complementary to this metric, Fan-out is the number of modules or methods called by a given module or method. These metrics are applied at both the module and method levels. Fan-in and Fan-out aim to realize how challenging it will be to replace a module or method in a program and how changes to a method or module can impact other methods or modules.

The majority of software complexity measures concentrate on the textual or control flow dimensions of a program. The knot metric is based on both of these characteristics, measuring the complexity and unstructuredness of a method's control flow (WOODWARD; HENNELL; HEDLEY, 1979). This metric is the number of intersections among the control flow paths through a method. A knot is a crossing of control structures caused

by an explicit jump out of a control structure. A knot can be created by misuse of overlapping jump structures, such as *break*, *goto*, *return* and *continue*. The knot metric is a useful and efficient measure for detecting, minimizing, and managing software complexity properties.

The last selected source code metric is also related to the complexity of a method.The maximum block nesting (MaxNesting) computes the depth of the most profound statement in a single method. Deep nesting indicates a sophisticated design with excessive control flow (in the case of if/else nesting), computational complexity (in the case of nested for loops), or a mix of the two.

Finally, as the test coverage metric, we selected the Edge Coverage value. EC criterion requires that for each branch in the program (e.g., if statements, loops), each branch has been executed at least once during testing.

We also perform a test smells metrics analysis. The best open-source tool we found was Jnose. JNose Test (VIRGÍNIO et al., 2020) is a tool developed to automatically detect test smells in a code at class level. Our goal is to use source code metrics at the method level, thus, we did not use any test smell metric in this work.

### 3.1.3 Projects selection

The next step of data selection is the choice of the source of the data. In our case, selected metrics must be extracted from real-world software projects that meet the following criteria: 1) open source projects, 2) projects with an appropriate amount of unit tests, and 3) Projects of different domains and sizes. Adopting these criteria allows us to make sure that we select a diversity of domain design and different development teams. Also, the criterion of having an appropriate amount of unit tests is fundamental to have a relevant historical base (dataset). The filter of projects with a relevant amount of unit tests means that we chose only projects that could have significant representation in the dataset, i.e. amount of instances.

### 3.1.4 Code Metrics extraction

This step consists basically in extracting or, when available, collecting the chosen metrics from the selected projects. This can be implemented in different forms and using

different tools. However, as shown in Figure 3.2, the predictive model has the established metrics as input data, so the process of extracting this set of metrics should be easy and quick to make it useful to the tester.

Another concern is that our solution can be used in various projects of different languages, domains, and sizes. This directly impacts the feature selection step since selecting language-specific source code metrics makes it impossible to apply the built model to other programming languages. To achieve broad adoption of our model, we selected only metrics widely used in the main programming languages. Thus, they can be generated by several tools. Furthermore, we chose not to create or alter a tool to generate specific metrics used as input data for the prediction model since these metrics could become dependent on this tool. Thus, limiting the application of the predictive model. Therefore, besides being widely known and applied in different contexts in academic works, all metrics chosen can also be generated by different tools. In the worst-case scenario, the tester would have to use more than one tool to extract all the metrics needed as input data for the predictive model. Even so, the solution would still be cost-effective.

An important point is that currently there is no tool in operation that calculates the exact value for PPC, our dependent variable, thus, it is necessary to define a strategy to obtain this value for the training phase. Such an strategy is shown in steps 5 to 7 of Figure 3.1 and will be detailed in Section 3.2.5. Once PPC is calculated, it is integrated to the other metrics (step 8).

### 3.1.5 Data Pre-processing and transformation

The next phase in the KDD process is the pre-processing and transformation of the collected data. The common goal of both steps is to make the final dataset the best for the training stage. Since the dataset is built using carefully generated data, we expect these steps to be quite simple and no specific adaptation is required.

### 3.1.6 Model construction and Evaluation

The remaining steps of the proposed methodology are applied as defined in the original KDD process. From the resulting dataset, different machine learning algorithms are used to generate a predictive model that can predict the PPC value for a method-level

test suite. This phase is known as data mining. Finally, as the last phase, we have the evaluation and interpretation of the results generated by each algorithm leading to the choice of the best predictive model according to the original goal.

## 3.2 Implementation of the Proposed Approach

The methodology detailed above can be applied in different ways, either by selecting a different set of tools, projects, or algorithms. In this section, we detail the most suitable implementation of the above methodology that we have identified. To this end, the following subsections give the implementation details for the dataset construction steps shown in Figure 3.1 where distinct solutions can be applied.

### 3.2.1 Projects selection

To collect the defined set of metrics, we selected 9 open-source Java projects to compose our training dataset, whose characteristics are reported in Table 3.2.1. We selected this set of projects because they meet all requirements defined in Section 3.1.3. Specifically, we selected application methods that had at least one associated test method. It directly impacts the size of the dataset. This selection was necessary because application methods that are not covered by any test method have a cover value of zero. Table 3.2.1, column *"Version"* is the elected release of the projects; column *"LOC"* reports the overall size of the used methods in the considered systems; column *"# Method"* shows the number of methods that were used from each project in this study. Finally, columns *"AvgCC"* and *"MaxCC"* show, respectively, the average cyclomatic complexity of all used methods and their maximum value of cyclomatic complexity. For better understanding, we detail the most relevant information and characteristics of each selected project in Appendix A.

### 3.2.2 Code metrics extraction

The set of 12 features (source code metrics) identified as the most aligned with the goals of the predictor model can be obtained using a single tool: *Understand*.

Although we used *Understand* only for metric extraction in this work, this tool is

| Project | Version | LOC | # Method | AvgCC | MaxCC |
|---|---|---|---|---|---|
| BioJava | 5.4.0 | 1552 | 104 | 3.18 | 13 |
| Commons-lang | 3.11.0 | 1313 | 94 | 3.97 | 12 |
| Commons-math | 3.6.1 | 14481 | 882 | 3.78 | 48 |
| JFreeChart | 1.5.0 | 25850 | 1080 | 6.03 | 50 |
| Checkstyle | 8.31.0 | 3181 | 202 | 3.55 | 8 |
| Commons-text | 1.9.0 | 3670 | 245 | 3.98 | 19 |
| Apache Dubbo | 2.7.8 | 1716 | 125 | 3.73 | 16 |
| Airship Client Library | 6.1.0 | 1074 | 74 | 4.01 | 8 |
| Exp4j | 0.4.8 | 5381 | 228 | 6.67 | 17 |

much more powerful than that. It is intended to aid in the maintenance and comprehension of vast volumes of legacy or newly generated source code. It is a maintenance-oriented, cross-platform, multi-language IDE. This tool includes support to many language versions and compilers, such as C/C++, C#, Java, Python, Web (HTML, PHP, CSS, Javascript, and XML). In addition to a wide variety of supported languages, 87 different metrics can be generated using this tool.

We have also analyzed in detail three other tools for metrics extraction: CKJM, Eclipse Metrics, and JHawk. CKJM calculates Chidamber and Kemerer object-oriented metrics by processing the bytecode of just compiled Java files (SPINELLIS, 2005). This tool does not offer a GUI and is an open-source project. Using the CKJM program, it is possible to extract eight different source code metrics, of which all are class-level. Since this tool does not generate method-level metrics, it is not helpful for our study.

Eclipse Metrics 1.3.10[1] is a free tool for metrics extraction and dependency analysis and is developed as a plugin for the Eclipse IDE. It supports just Java projects and 37 metrics, including metrics for design properties and quality attributes. Although it is a powerful tool that generates many metrics, we did not adopt it in this study because it is dependent on the technology (i.e., IDE-specific version). It requires knowledge and interaction with the compiling and execution environment of the Eclipse IDE, which imposes a limitation to the model usage afterwords.

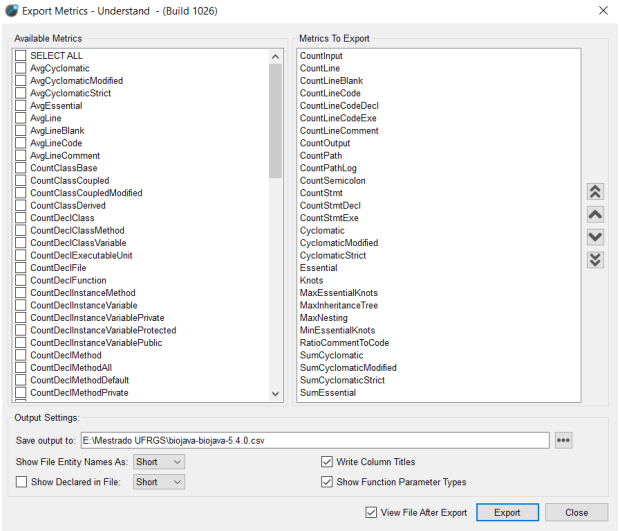
At last, JHawk[2] is a static code analysis tool that supports Java projects and can be applied to extract over 100 OOP metrics in different levels (i.e., method, class, and package). Although a free trial is available, We did not adopt this tool in this study for two main reasons: 1) like the other tools we analyzed (CKJM and Eclipse Metrics), it only supports Java projects; and 2) despite having a demo version, the tool does not offer

[1]<https://github.com/qxo/eclipse-metrics-plugin>
[2]<http://www.virtualmachinery.com/jhawkprod.htm>

a no-cost license for academic purposes. Currently, the academic license costs $99.00.

The metric extraction process is done on a per-project basis, so from each source-code project, we generated a CSV file containing the specified metrics. Using the *Understand* tool, we first import the project's source code. After that, by selecting Metrics > Export Metrics from the menus or pressing the Generate Detailed Metrics button in the Project Metrics Browser, we exported the metric data to a comma-delimited CSV file. The Export Metrics dialog is shown in Figure 3.3. As can be seen, extracting the metrics using the *Understand* tool is very simple and fast. We perform this process for each project, generating 9 CSV files containing the defined metrics.

Figure 3.3: Exporting metrics to a CSV file




Source: The author

### 3.2.3 Test requirements generation

As mentioned in Chapter 1, there is currently no operational tool that can generate the PPC value for a test suite. Therefore, we had to define a strategy to obtain this coverage value. Our strategy is based on the generation of test requirements and test paths for each method from a project. After obtaining TP and TR, we calculate the PPC value based on the number of TR directly covered by a given set of TPs. For generating the test requirements, we first considered the combined use of two tools: Control Flow Graph Factory[3] developed by Dr. Garbage and Graph Coverage Web Application[4] implemented by Wuzhi Xu. The first one generates the CFG in text format. We used the generated CFG

[3]<https://marketplace.eclipse.org/content/control-flow-graph-factory>
[4]<https://cs.gmu.edu:8443/offutt/coverage/GraphCoverage>

files as input in the web tool to generate the PPC and EC test requirements. However, due to the large number of methods that would need to generate the coverage value, we needed to automate the integration between these two tools.

The necessity of automating this integration was not one reason we decided not to adopt these two tools in this study. The motive is related to the limitations of these tools. The first issue with the Dr. Garbage plugin is that it is only compatible with Eclipse releases before Eclipse Mars, from 2015. Another explanation we did not use this tool was that it was unreliable; in many cases, we noticed that incorrect outputs were produced. Concerning the Graph Coverage Web Application, we noticed that it does not support creating the test requirements for highly complex methods (i.e., methods with a more significant number of nodes and edges).

To enable the generation of test requirements, we thus adapted a tool developed in Java by Seung Hun Lee and Evan Platt[5]. The main change made to the tool was to compute the TR based on the source code line number and not by the basic block. This tool receives as input data a Java file (i.e., Class) and provides its control flow graph and the selected coverage criteria, which the user specifies. It is pretty simple and objective, consisting of two main modules: 1) the graph generator, which parses the source code while performing some clean-up and adaption and produces its CFG; and 2) the test requirement generator, which generates the TR for a coverage criteria based on the graph built in the previous module. The supported criteria are node coverage, edge coverage, edge-pair coverage, and prime path coverage. The tool's final edition, with all of the modifications, can be found in our online repository[6]. Using this tool, we generated the test requirements in text file format for each selected method from the chosen projects.

### 3.2.4 Test paths generation

As for the PPC value, we have not found an oporational tool that provides th test paths for a given code and test suite. Thus, we created ExecutionFlow[7], a tool that aims to produce a project's test paths at the method level (NIEMIEC; SILVA; COTA, 2021). Our tool is based on a debugger, aspect-oriented programming, and a testing framework to make this work possible. A solution that generates an application's test paths is strongly

[5]<https://github.com/evplatt/TRGeneration>
[6]<https://bitbucket.org/mwolfart/trgeneration>
[7]<https://github.com/williamniemiec/ExecutionFlow>

dependent on the programming language, mainly due to syntactic differences. To meet its purpose in this work, ExecutionFlow generates test paths for Java projects.

Aspect-oriented programming is used in the proposed tool to obtain information about the methods and constructors used in each Junit test method. It is essential to use a platform that supports aspect-oriented programming in order to use it. We used the AspectJ via the AJDT plugin for Eclipse IDE, which facilitates aspects-oriented programming in Java. This plugin supports obtaining the location of compiled and source-code files, method signature with its arguments, application method name, and the line number that the method is called. The debugger uses this information to establish which methods and constructors will be used to compute the test paths. In summary, to generate the test paths using ExecutionFlow the tester must:

1. Import the project to Eclipse and convert it to an AspectJ project
2. Add the file 'aspectjtools.jar'[8] to the build path
3. Add the jar of ExecutionFlow to the AspectJ build path
4. Run a test (or a test package or all tests) using JUnit

An important point is that the set of TRs and TPs of an application method must be generated based on the same CFG. To ensure this, we apply the same preprocessing process in both tools.

### 3.2.5 PPC calculation

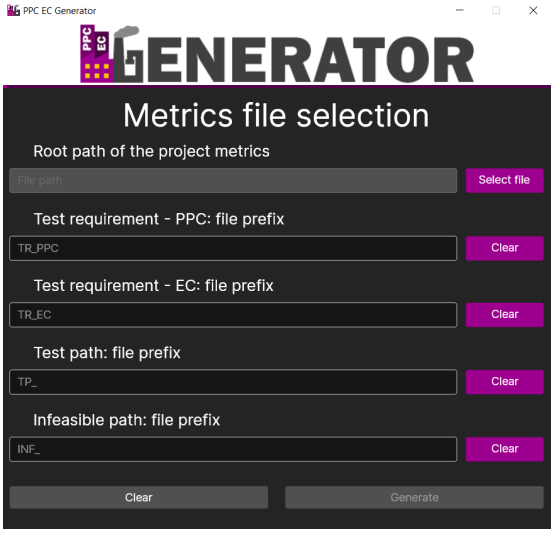
By comparing the set of test paths to the set of test requirements, the code coverage value for a criterion can be determined. As mentioned earlier in Section 3.1, we selected two code coverage metrics in this study: PPC and EC. To simplify, The same tool used to generate the PPC test requirements was used to generate the EC TRs. As it will be detailed in Section 3.2.7.

We developed a simple script to generate the PPC and EC value, which has as input data a set of test requirements and test paths. As one can see in Figure 3.4, this application takes as input the project's root directory path that will be collected the TR and TP files for each method. Additionally, this tool has as input the prefix name of the TR of PPC, TR of EC, TP, and infeasible path files. The use of prefixes was the best way we found to automatically and quickly generate the coverage values for all the selected

[8]https://mvnrepository.com/artifact/org.aspectj/aspectjtools

methods of a given project. In this manner, our application goes through all folders and subfolders, orienting by the determined prefixes.

Figure 3.4: Interface of the tool developed to calculate coverage value.




Source: The author

In this study, we did not perform an infeasible path analysis for the PPC criteria, given that it is a manual analysis and would not be possible to perform for all the methods of the selected projects. Thinking of future works and the broader adoption of the tool, we developed this functionality, leaving this input as optional. As for the other inputs, they are all required. Using this tool, we generated for each project a CSV file containing the application's method signature and the corresponding PPC and EC values.

### 3.2.6 Data integration

Our aim in this step is to create a consolidated dataset that can be used as input in the training phase. The CSV file containing the source code metrics provided by Understand and the CSV file containing the code coverage metrics (i.e., PPC and EC values) for each of the 9 projects are the key inputs for this step. Integrating the two datasets for each project and merging them to form a single dataset is not as straightforward as it seems. The key issue with this process is that the structure of the CSV file created by Understand differs significantly from the structure of the CSV file containing the code coverage metrics.

The Understand tool's CSV file has the following columns, as shown in Table 3.3: *Kind*, which specifies the type of source code unit; *Name* which is the signature of the method or constructor; and the another 12 columns, each one referring to the established

source code metrics (SCM). The CSV file structure containing the code coverage metrics is shown in Table 3.4, with the method signature or constructor as the first column and the EC and PPC values as the following two columns. Given the structure of these two different CSV files, we had to integrate them to generate a dataset according to the structure shown in Table 3.5. To generate this final dataset for each project, we implemented a script that is publicly available on our GitHub repository[9].

Table 3.3: Structure of the CSV file generated by the Understand tool.

| **Kind** | **Name** | **SCM 1** | **...** | **SCM 12** |
|---|---|---|---|---|
| Public Method | project.api.WebSettings.toString() | 2 | ... | 0 |
| Private Constructor | project.web.Open.validateAndBuild() | 4 | ... | 2 |
| Public Method | project.test.WebSettingsTest.toString() | 0 | ... | 0 |
| Public Method | project.RequestUtils.resolve(URl) | 5 | ... | 1 |
| Public Method | project.test.OpenTest.validateAndBuild() | 3 | ... | 2 |
| Public Method | project.test.RequestUtilsTest.resolve(URl) | 2 | ... | 4 |

Source: The author

Table 3.4: Structure of the CSV file containing EC and PPC values.

| **Application method** | **Test method** | **EC** | **PPC** |
|---|---|---|---|
| project.main.WebSettings.toString() | project.test.WebSettingsTest.toString() | 0.55 | 0.40 |
| project.main.Open.validateAndBuild() | project.test.OpenTest.validateAndBuild() | 1.00 | 1.00 |
| project.main.Open.validateAndBuild() | project.test.validateAndBuild() | 1.00 | 1.00 |
| project.main.RequestUtils.resolve(URl) | project.test.RequestUtilsTest.resolve(URl) | 0.8 | 0.92 |

Source: The author

Table 3.5: Structure of the resulting dataset.

| **Application method** | **SCM 1** | **...** | **SCM 12** | **Test method** | **SCM 1** | **...** | **SCM 9** | **EC** | **PPC** |
|---|---|---|---|---|---|---|---|---|---|
| project.main.WebSettings.toString() | 2 | ... | 0 | project.test.WebSettingsTest.toString() | 0 | ... | 0 | 0.55 | 0.40 |
| project.main.Open.validateAndBuild() | 4 | ... | 2 | project.test.OpenTest.validateAndBuild() | 3 | ... | 2 | 1.00 | 1.00 |
| project.main.RequestUtils.resolve(URl) | 5 | ... | 1 | project.test.RequestUtilsTest.resolve(URl) | 2 | ... | 4 | 0.8 | 0.92 |

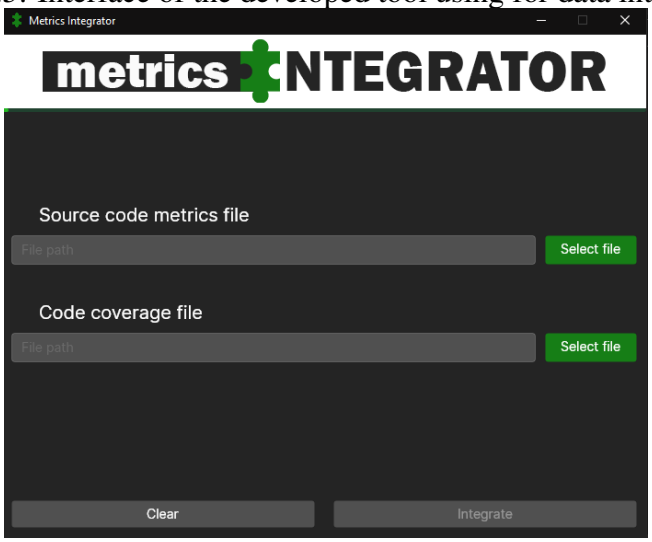
Source: The author

As we can see in Figure 3.5, our script requires two files as input. The first file is a CSV with the source code metrics generated by the *Understand* tool, structured as shown in Table 3.3. The second one is the CSV containing the test metrics, structured as shown in Table 3.4. Using this tool, we generate the resulting dataset for each one of the 9 chosen projects. Finally, we merge these nine datasets, generating only one dataset (i.e., training dataset). Regarding the structure of the dataset, it consists of 23 columns:

[9]<https://github.com/keslleylima/MetricsIntegrator>

12 application method source code metrics, 9 test method source code metrics, and 2 test coverage metrics (PPC and EC).

Figure 3.5: Interface of the developed tool using for data integration.




Source: The author

### 3.2.7 Data Cleaning and Transformation

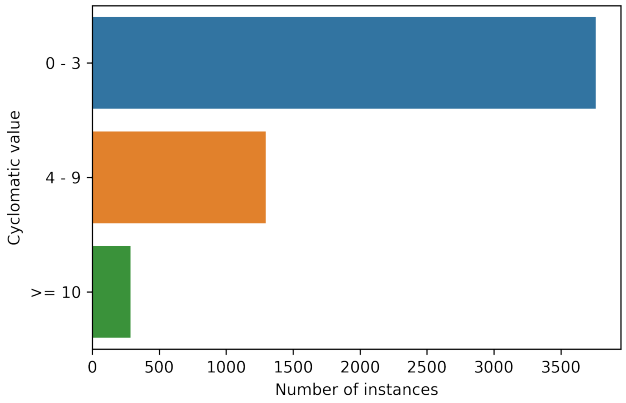
As a result of the data integration process, a dataset with 5.067 unique instances was generated. When multiple data sources are at play, data cleaning activities are essential to ensure the dataset's quality. First, we remove all instances that have coverage values as "NaN". Registers with coverage value 'NaN' are cases that the test requirements file was empty. Therefore, to avoid a coverage value that does not match the actual value, we decided to remove these. We also removed the registries for both test criteria (i.e., EC and PPC) with zero value or registries with PPC > EC.

Another relevant piece of information from this raw dataset is the average cyclomatic complexity of the application methods, which is 3.39[10]. As we can see in more detail in Figure 3.6, a significant part of this raw dataset consists of registers for which the application method has a complexity value up to 3. Due to this predominance of instances with a low complexity value, the built regression model may have difficulty predicting the PPC value for application methods with a medium or high complexity value. To solve this problem, we applied the under-sampling technique to adjust the ratio between the different categories of cyclomatic complexity represented, altering the class distribution of our dataset. We applied under-sampling to the dataset using the following parameters:

[10]This average CC is from the raw dataset, not the final dataset, which was detailed in Figure 3.2.1

*random_state:* 0 and *sampling_strategy:* 1:50, 2:800.

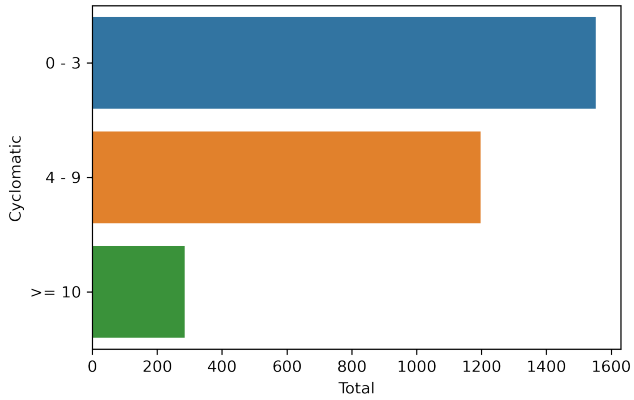
Figure 3.6: Composition of the raw dataset in 3 different distribution classes according to the Cyclomatic Complexity value of the application methods.




Source: The author

As a result of the over-sampling task and the other performed data cleaning tasks, our resulting dataset consists of 3034 registries and an average application method complexity of 4.79. As we can see from Figure 3.7, we have improved the class distribution in the dataset through over-sampling. We consider that the transformations performed to the dataset are beneficial for generating a regression model that has a good predicting performance for methods with different values of cyclomatic complexity.

Figure 3.7: Composition of the resulting dataset in 3 different distribution classes according to the Cyclomatic Complexity value of the application methods.

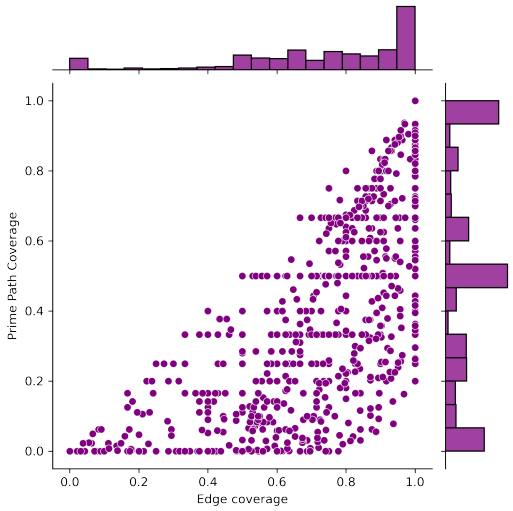



Source: The author

Figure 3.8 shows the relationship between EC and PPC for all test suites in the dataset. As expected, PPC is lower than EC for most cases. We note that, since PPC subsumes EC, a low EC value must indicate a low PPC value, but a high EC may not

imply a high PPC. For instance, for high complexity methods, EC generates more trivial test requirements than PPC. This can be observed in Figure 3.8 where PPC values present large variation for EC values in the range [0.8,1].

Figure 3.8: Dataset composition based on the PPC and EC coverage of the test suites.




Source: The author

# 4 MODEL CONSTRUCTION AND VALIDATION

## 4.1 Model Construction strategy

The dataset built by executing the steps detailed in Chapter 3 is used as the training set in the data mining step of the proposed methodology. Our training dataset will be used as input to build three different predictive models:

- **SCM-based model:** a predictive model using the set of 21 source code metrics of the application methods and test methods as independent variables. This model aims to predict the PPC value using only source code metrics, which can be easily extracted by different tools. Therefore, we hope to generate a predictive model with good predictive performance without using test coverage metrics.
- **SCM+EC-based model:** the second predictive model was generated considering the set of 21 source code metrics, the same as model 1, plus the EC value. The inclusion of the EC value aims to improve the prediction performance of this model since EC is directly related to the PPC value. As previously stated, because PPC subsumes EC, a low EC value may indicate a low PPC value, while a high EC value may not imply a high PPC value. The effort to extract the necessary set of metrics required as input for this model, although low, is higher than model 1 since it will be necessary to use more than one tool to extract the 22 metrics: one tool to extract the 21 source code metrics and one to obtain the EC value at the method level.
- **EC-based model:** our third regression model predicts the PPC value of a test suite using just the EC value as input. As shown in our prior work (SILVA; COTA, 2020), the PPC value of a test suite at the method level can be estimated with reasonable accuracy using only the EC value. As a result, we evaluate only one feature (EC value) as a feasible model to predict the PPC value in this study too.

We use numerical metrics to predict PPC, which is also numerical, so we adopt regression analysis algorithms. We consider four different machine learning algorithms to generate these three different predictive models: SVR, ANN regression, KNN regression, and RFR. We chose these algorithms because they are widely known and adopted in related work in the testing area, for example, in mutation score prediction (Zhang et al., 2019) and EC prediction (GRANO et al., 2019). All these algorithms are briefly described below.

- **Support Vector Regression:** Support Vector Machine can also be employed as a regression approach while retaining all of the algorithm's fundamental characteristics. This variant is known as Support Vector Regression (SVR). With a few minor changes, it employs the same categorization concepts as the Support Vector Machine. For example, because the output is a real number, it becomes difficult to predict the information at hand, which has an endless number of possibilities. In the case of regression, a margin of tolerance (epsilon) is specified in order to approximate the Support Vector Machine that the issue would have already requested.
- **Regression Artificial Neural Network:** Neural networks can be used not only for classification but also for regression. An artificial neural network (ANN) is an algorithm designed to simulate how a human brain analyses and processes information. Similar to the human brain, ANNs are made up of linked nodes. The input ones are subject to receiving different structures and forms of information that are based on an internal weighting system. The neural network attempts to learn about the information in order to provide an output report. Backpropagation refers to the set of learning rules used by ANNs. During the training phase, the ANN attempts to detect various patterns in the data provided to it. During the supervised phase, the ANN compares the actual output to the desired output to find discrepancies. These disparities are compensated for in a variety of ways.
- **K-nearest Neighbours Regression:** The k-nearest neighbors algorithm is a simple, easy-to-implement supervised machine learning algorithm that can be used to solve both classification and regression problems. In this study, we applied the K-nearest Neighbours Regression. Each sample is a pair in this algorithm, including an input vector(instance) and the desired output value.
- **Random Forest Regression:** Random Forest Regression (RFR) is a supervised learning algorithm that performs regression using the ensemble learning approach. The ensemble learning approach combines predictions from several machine learning algorithms; in this context, it combines numerous decision trees to make a more accurate prediction than a single model. The Decision Tree algorithm has a significant drawback in that it leads to over-fitting [1]. This issue may be mitigated by using the RFR instead of the Decision Tree Regression. Furthermore, the Random Forest technique is faster and more resilient than previous regression models (SAMMUT;

[1] An overfitting scenario occurs when, on the training dataset, the model has excellent predictive performance, but when we use the validation data, the performance is poor.

WEBB, 2017).

Each of the three based models (SCM, SCM+EC and EC) will be generated using the four different ML algorithms. Thus, we will generate a total of 12 predictive models, four models for each set of input metrics.

## 4.2 Validation strategy

For each candidate model, one needs to define how to evaluate the prediction performance of that model. To this end, in this study, we divided our validation methodology into internal and external validation. Internal validation aims to quantify optimism in model performance, considering performance for a single population (i.e., dataset). External validation, on the other hand, aims to evaluate how the model performs for similar populations (DEBRAY et al., 2015).

Internal validation techniques are cost-effective since they split a single input dataset into parts, with some used for training the predictor (training data) and the remainder used for validation (test data). This technique is repeated until each part has been employed as testing data at least once. K-fold cross-validation is another term for this technique (ANGUITA et al., 2012). It has a single parameter called k that refers to the number of groups that a given data sample is split into. In this work, we applied 10-fold cross-validation, i.e., K=10.

External validation involves applying separately obtained datasets (hence, external) to validate the performance of a model built on initial training data. External validation is usually considered significant evidence for how the model performs in different scenarios. Since the validation set comes from an independent source, any feature set that is falsely selected due to the input training data (e.g., technical or sampling bias) would likely fail. Hence, a positive performance in it is regarded as proof of generalizability. We generated a dataset consisting of 293 registers from 3 additional projects to perform the external validation. The projects chosen for external validation were selected based on the same criteria that we adopted in the training model step. In addition, we were concerned about selecting projects as different as possible from the set of projects used in the training phase. Table 4.1 details the main information of the selected methods from these three projects.

Table 4.1: Characteristics of the projects used for the external validation.

| Project | Version | LOC | # Method | AvgCC | MaxCC |
|---|---|---|---|---|---|
| Apache Accumulo | 2.0.1 | 1465 | 91 | 3.92 | 13 |
| Commons IO | 2.6 | 893 | 98 | 2.48 | 8 |
| Apache Tika | 1.26 | 2128 | 104 | 5.25 | 16 |

Source: The author

To evaluate the prediction performance of the generated regression models and consequently to be able to assess which one is the best, we adopted five largely used regression model evaluation metrics:

- **Mean Absolute Error (MAE)** is the average of all absolute errors. Absolute Error is the difference between the measured value and the actual value. The Mean Absolute Error is calculated by adding up all the absolute errors and dividing them by the number of errors. The MAE measures the average magnitude of the errors in a set of forecasts without considering their direction. The MAE is a linear score which means that all the individual differences are weighted equally in the average.
- **Mean Square Error (MSE)** is defined as the mean or average of the square of the difference between actual and estimated values. It tells how close a regression line is to a set of points. It does this by taking the distances from the points to the regression line (i.e., error rate) and squaring them. The squaring is necessary to remove any negative signs. It also gives more weight to more significant differences. MSE is the most commonly used loss function for regression models. The lower the MSE, the better the forecast.
- **Mean Squared Logarithmic Error (MSLE)** is, as the name suggests, a variation of the MSE. The application of the logarithm makes MSLE only care about the relative difference between the true and the predicted value (i.e., percentual difference). This means that MSLE will treat minor differences between small actual and predicted values approximately the same as big differences between large actual and predicted values. It also penalizes underestimates more than overestimates.
- **Median Absolute Error (MedAE)** because it is resistant to outliers, the MedAE is especially interesting. By taking the median of all absolute differences between the target and the prediction, the loss is calculated.
- **R-squared (R2 Score)** is a statistical measure that represents the proportion of the variance for a dependent variable that is explained by an independent variable or variables in a regression model. The R2 Score varies between 0.0 and 1.0. The best

possible R2 Score is 1.0 and it can be negative because the model can be arbitrarily worse.

# 5 RESULTS

## 5.1 Internal Validation

As explained, we apply the for selected ML algorithms to each set of input metrics (SCM, SCM + EC, EC) to generate the best possible models. As a result, we initially generated 12 predictive models. Our goal was to identify the best combination by algorithm and input set to predict PPC. It is worth pointing out that we adopted default hyperparameters for all four algorithms adopted in this work. We did experimented a few tuning activities but they did not led to significant improvement in predictive performance.

Table 5.1 presents the KNNR algorithm is the one that generates the best model for 2 out the 3 desired models: SCM-based and EC-based model. For the SCM+EC-based model, the best algorithm is RFR. As expected, using the EC test metric significantly improves the performance of a model that aims to predict the PPC value. This outcome is not surprising since both EC and PPC are control flow coverage criteria and are strongly related.

Table 5.1: Prediction performance of the 12 generated models

| Model | Algorithm | MAE | MSE | MSLE | MedAE | R2 Score |
|---|---|---|---|---|---|---|
| Using SCM | SVR | 0.12 | 0.03 | 0.01 | 0.13 | 0.59 |
| | RANN | 0.13 | 0.05 | 0.02 | 0.00 | 0.20 |
| | **KNNR** | **0.05** | **0.01** | **0.00** | **0.00** | **0.69** |
| | RFR | 0.09 | 0.01 | 0.00 | 0.09 | 0.64 |
| Using SCM + EC | SVR | 0.15 | 0.02 | 0.01 | 0.13 | 0.63 |
| | RANN | 0.05 | 0.01 | 0.00 | 0.00 | 0.70 |
| | KNNR | 0.14 | 0.04 | 0.02 | 0.02 | 0.18 |
| | **RFR** | **0.04** | **0.00** | **0.00** | **0.05** | **0.91** |
| Only EC | SVR | 0.11 | 0.02 | 0.01 | 0.05 | 0.63 |
| | RANN | 0.13 | 0.05 | 0.02 | 0.01 | 0.22 |
| | **KNNR** | **0.08** | **0.02** | **0.00** | **0.03** | **0.77** |
| | RFR | 0.10 | 0.01 | 0.00 | 0.10 | 0.67 |

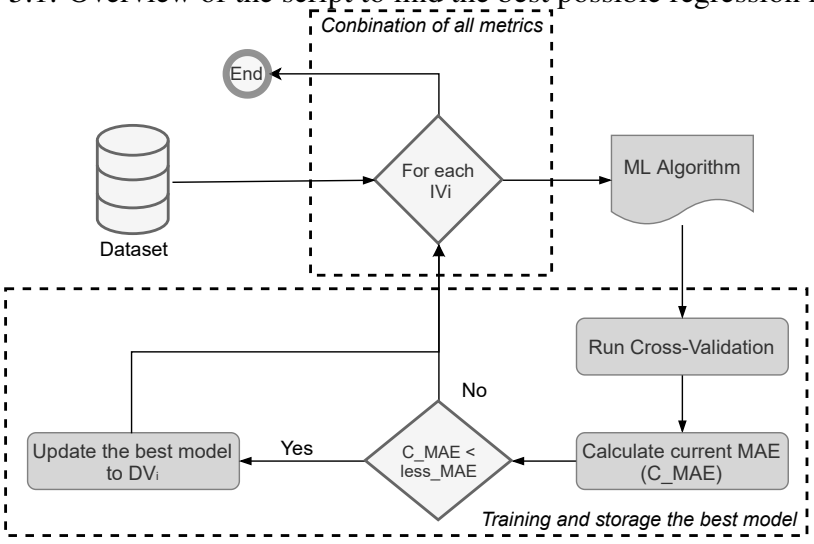
Source: The author

Since our main goal is to generate a model with good prediction performance and that is also easy to use by the tester, using those algorithms (KNNR and RFR), we seek to generate the best possible models for the SCM-based and SCM+EC-base model such:

1) They outperform or perform similar to the models previously obtained and 2) use a reduced set of source code metrics.

To this end, we developed a script in Phyton that performs all possible combinations among the independent variables from our training dataset.The mining process is depicted in Figure 5.1. To find the best feature combination, one or more independent variable (IVi) is selected in the dataset, and the regression algorithm (RAi), RFR for the SCM-based model and KNNR for the SCM+EC-based model, is run for this combination. For each combination (IVi, PPC, RAi), the script applies the ten-fold cross-validation procedure and selects the model with the best MAE value as the best model for the combination (IVi, PPC).

Figure 5.1: Overview of the script to find the best possible regression model.




Source: The author

Although our metrics combination algorithm is used only once, it has a high operational cost since to obtain the best feature combinations, the algorithm performs $2^n$ combinations, where n is the number of features/columns in the training dataset. Therefore, we decided to define a subset that would be employed as input to this algorithm. To define this subset of features, we performed the permutation feature importance technique[1] for the SCM-based model and the feature importance technique[2] for the SCM+EC-based model. Table 5.2 presents the results of this analysis when we were able to select the 10 metrics with the best relationship with PPC. As we can see in Table 5.2, expect for the EC value, none of the source code metrics significantly correlates with the PPC value.

[1]<https://scikit-learn.org/stable/modules/permutation_importance.html>
[2]<https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html>

Table 5.2: TOP-10: Feature importance of the SCM-based model and SCM+EC-based model.

| **TOP-10: SCM** | | | **TOP-10: SCM + EC** | | |
|---|---|---|---|---|---|
| **Metric** | **Set** | **Importance** | **Metric** | **Set** | **Importance** |
| Fan-in | *A* | 0.0335 | EC | *S* | 0.7504 |
| LOC | *A* | 0.0280 | NPATH | *A* | 0.0488 |
| NPATH | *A* | 0.0228 | STMT | *A* | 0.0408 |
| LOC | *T* | 0.0182 | Fan-out | *A* | 0.0327 |
| STMT | *T* | 0.0161 | STMTd | *A* | 0.0202 |
| STMTx | *T* | 0.0153 | Knots | *A* | 0.0177 |
| STMT | *A* | 0.0140 | STMTx | *A* | 0.0161 |
| Fan-out | *T* | 0.0112 | Fan-in | *A* | 0.0135 |
| STMTx | *A* | 0.0007 | LOC | *A* | 0.0117 |
| Fan-out | *A* | 0.0006 | MaxNesting | *A* | 0.0009 |

Source: The author

Using this script and the feature subsets shown in the Table 5.2, we find the best model that uses only source metrics as the input data (i.e., SCM-based model) and the best model that uses EC and source code metrics (i.e., SCM+EC-based model). As shown in Table 5.3, for the model that uses only SCM, the best combination achieved an MAE of 0.0494 using only nine source code metrics. We achieved a very similar prediction performance compared to the previous best model using 21 metrics, based on the MAE value. But, reduced the set of metrics used as input data, by half. This best model requires as input data 3 metrics from test methods and 6 metrics from application methods. Still from Table 5.3, one can draw the following rationale concerning the best model:

- NPATH metric of the application method is correlated with the testing effort because a complex code — in this study, a method — is more difficult to understand, maintain and test. A higher value for this metric indicates that this unit of code requires more testing effort (e.g., test paths, test data) when covering the control flows due to the conditional expressions and the boolean operators (e.g., If, ElseIf, For, While, Case);
- The Fan-in and Fan-out metrics of application methods describe how difficult it will be to replace a method, as well as how modifications to one method can impact other methods. Regarding Fan-out metric of the test methods, a higher value for this metric indicates that a test method calls more methods and possibly has more

global variables. Thus, it may indicate that the method is more robust and tends to cover more paths of a given method in the application.

- Similar to Fan-out of the test method, the number of executable statements (STMTx) and lines of code (LOC) are size-related, describing that the test method is likely more robust and can cover more paths.
- The STMT, STMTx and LOC metrics demonstrate the size of the method, which is also an indication of the difficulty to understand and test that method. Consequently, a low-value for these metrics indicates a likely high value for PPC. As we observed in the dataset, test methods with higher values of STMTx are the ones that implement more test paths, thus, potentially covering more test requirements.

Table 5.3: Top 5: SCM-based model.

| **Best Feature Combination** | **MAE** |
|---|---|
| App method: Fan-in, LOC, NPATH, STMT, STMTx, Fan-out<br>Test method: LOC, STMTx, Fan-out | 0.0494 |
| App method: Fan-in, LOC, NPATH, STMT, STMx, Fan-out<br>Test Method: LOC, Fan-out | 0.0496 |
| App method: Fan-in, NPATH, STMT, STMTx, Fan-out<br>Test Method: LOC, Fan-out | 0.0496 |
| App method: Fan-in, LOC, NPATH, STMT, STMTx, Fan-out<br>Test method: LOC, STMT, Fan-out | 0.0500 |
| App method: Fan-in, LOC, NPATH, STMT, Fan-out<br>Test method: LOC, Fan-out | 0.0503 |

Source: The author

As shown in Table 5.3, all the five built models could be used as our predictive SCM-based model, as they have very similar prediction performance using different combinations of metrics. For external validation, we adopted the first model in the TOP 5 ranking because it has the lowest MAE.

We performed the same process as described before to obtain the best predictive model using source code metrics and EC value. After running our script, the top 5 models are displayed in Table 5.4 for the SCM+EC-based model. Regarding the best combination of features, the MAE for the best model is 0.0164, lower (i.e., better) than the model using all metrics, which had an MAE of 0.04. This best model has a better prediction performance to the model using all metrics, but using as input data only 6 metrics (5 source

code metrics and EC value). For external validation, we decided to adopt this model. In combination with a better prediction performance, the main reason for this choice is that our goal is to generate a model with the least possible effort to use. Regarding the best model in this feature combination ranking, we came to the following findings:

- As a consequence of the performed feature importance analysis, the TOP-10 best metrics do not have any metrics that describe the test method. We believe that due to the strong correlation between the EC and PPC values, using the EC value as an input that already relates the characteristics of a test method to the achieved PPC value.
- For this model, two different metrics make up the best combination, which were not present in any of the 5 best combinations of the model using SCM: Knots and Maxnesting.
- Knots metric is based on measuring the complexity and unstructuredness of the control flow of an application method. As a result, a low value for this metric indicates the PPC value is likely to be high.
- We believe that MaxNesting is among the best combination set because it indicates an application method with excessive control flow (e.g., if/else nesting), complexity (e.g., loop structures), or a mix of the two.

Table 5.4: Top 5: SCM+EC-based model.

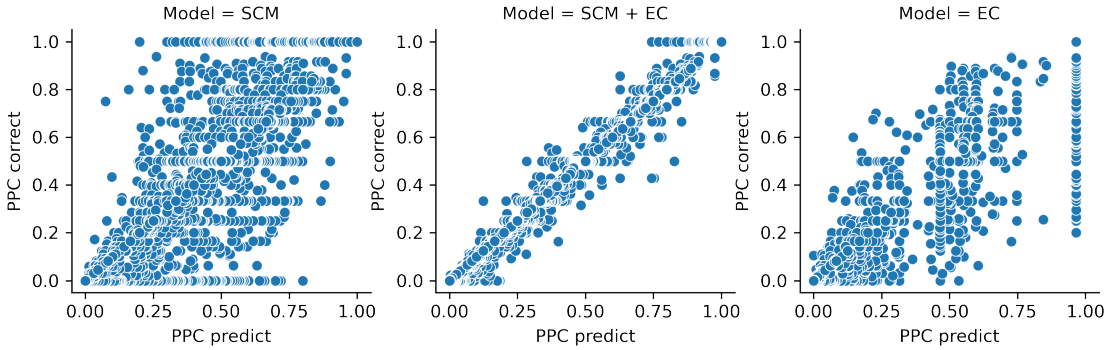
| **Best Feature Combination** | **MAE** |
|---|---|
| App method: STMTd, Knots, Fan-in, LOC, MaxNesting<br>Test suite: EC | 0.0164 |
| App method: STMT, STMTd, Knots, Fan-in, LOC, MaxNesting<br>Test suite: EC | 0.018 |
| App Method: STMTx, Fan-in, LOC<br>Test suite: EC | 0.0186 |
| App Method: Fan-out, STMTx, Fan-in, MaxNesting<br>Test suite: EC | 0.0223 |
| App Method: Knots, STMTx, Fan-in, LOC<br>Test suite: EC | 0.0228 |

As shown in Figure 5.2, the model that uses only source code metrics has an excellent prediction performance because despite having more challenges in predicting

the exact value of PPC, the predicted value in most situations is close to the actual value. By not using any test coverage metrics, this model has metrics that have a low correlation with the PPC value. Therefore, despite solid results in the internal validation, this model may not have a similar prediction performance for projects with different source code characteristics (e.g., domain, size, programming style, and design).

EC-based model its turn also performed well in terms of prediction. We believe that a model which predicts the PPC value using only one metric as input data may be applied in real-world scenarios by the tester. As previously stated, PPC and EC are highly correlated, allowing one to predict the value of another. However, it is more prone to errors because it only uses one metric as input data. In Figure 5.2, for example, the model has learned that when the EC value is equal to or near to 1, the PPC value is likewise equal to or close to 1. This relationship, however, is erroneous since there are situations when the EC value is high but the PPC value is low, especially in high complexity methods. We believe the model learned this way because the dataset has a significant proportion of instances with low or medium cyclomatic complexity, which may be enhanced by adding methods with high cyclomatic complexity. Finally, the SCM+EC-based model has the best prediction performance. This excellent result shows that the use of EC combined with source code metrics improves the prediction performance, generating predictions with low error rates.

Figure 5.2: Prediction performance of the best models in internal validation.




Source: The author

As a result of the internal validation process, we built different models to predict PPC. We generated 6 models for the SCM-based model: 1 using all metrics, as shown in row 3 of Table 5.1, and 5 using the best combinations (Table 5.3). For the SCM+EC-based model, 1 model using all metrics, as shown in row 8 of Table 5.1, and 5 models using the best combinations (Table 5.4). For the EC-based model, 1 model was generated, as detailed in row 11 of Table 5.1.

## 5.2 External Validation

To validate our model in practice, we considered 293 test suites at the method level of different application methods from three open-source projects that are not part of the training dataset: Apache Accumulo, Commons IO and Tika. The average complexity of the application methods in this dataset is 3.91. To be clear, although we have generated different predictive models throughout the internal validation activities, to simulate the use of our three different models in a real-world scenario, we will use only the models listed in Table 5.5.

Table 5.5: The selected models adopted in the external validation.

| Model | Metrics required as input data |
|---|---|
| SCM-based | App method: Fan-in, LOC, NPATH, STMT, STMTx, Fan-out<br>Test method: LOC, STMTx, Fan-out |
| SCM+EC-based | App method: STMTd, Knots, Fan-in, LOC, MaxNesting<br>Test suite: EC |
| EC-based | Test suite: EC |

Source: The author

By applying the models shown in the Table 5.5, the PPC values for the test suites in the validation dataset were predicted. Table 5.6 details the prediction performance of each of these regression models. Compared to the results obtained in the internal validation, only the model using only source code metrics had a considerable variation in prediction performance, in which it achieved an MAE of 0.28. We infer that the SCM-based model has the worst prediction performance because none of the source code metrics strongly correlates with PPC value. As shown in Table 5.2, the metric with the highest correlation with PPC has an importance value of 0.0335 (3,35%). On the other hand, the SCM+EC-based model has the EC value with an importance value of 0.7504 (75%). Thus, although it is possible to predict PPC using only source code metrics, as expected, it is inevitable that using test coverage metrics (in this work, EC) with a strong correlation with PPC value, increasing the prediction performance.

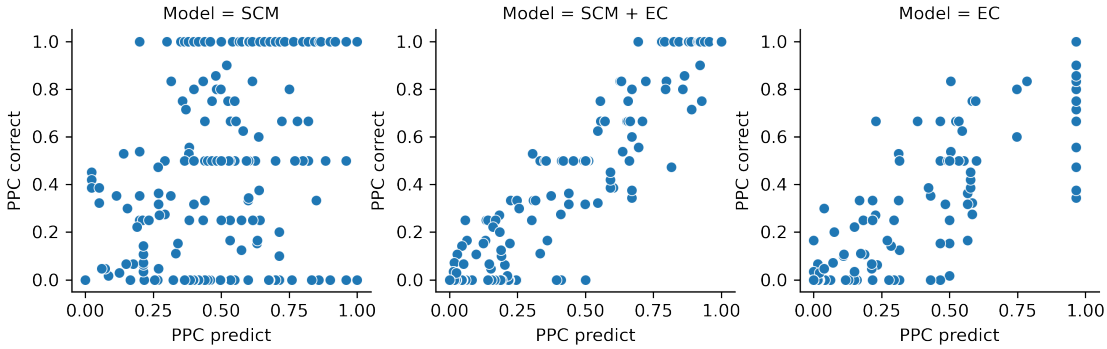
Table 5.6: The predictive performance of the models in the practical scenario.

| Model | MAE | MSE | MSLE | MedAE | R2 Score |
|---|---|---|---|---|---|
| SCM-based | 0.28 | 0.13 | 0.06 | 0.23 | 0.13 |
| SCM+EC-based | 0.06 | 0.01 | 0.00 | 0.01 | 0.92 |
| EC-based | 0.09 | 0.02 | 0.01 | 0.03 | 0.85 |

Source: The author

Figure 5.3 displays the prediction performance of the three different models in the external validation. As expected, there was a slight decrease in the prediction performance

Figure 5.3: Prediction performance of the best models in external validation




Source: The author

since they are unknown projects by the model. The models using EC (using SCM + EC and using only EC) as data input obtained similar performance to those reported in the internal validation. In contrast, the model using only source code metrics as input significantly increased the error rate. We believe this increase may be due to two main points: 1) The model was applied to predict the PPC of methods from different domains than the projects used in the training phase; 2) SCM have a low correlation with PPC. Based on these results, we can conclude that the three generated models can be used to predict the PPC value of a test suite at the method level.

## 5.3 Threats to Validity

In this section we discuss the threats to the validity of our study and describe strategies used to mitigate. A probable mistake in our feature collection poses the greatest threat to internal validity. We collected features and applied our strategies using consolidated third-party solutions to limit this threat, assuming these tools produce accurate results. We created a script to calculate the PPC and EC value for each test suite. We thoroughly examined our implementation and feel that, based on our findings, our implementations have a low risk of faults. To mitigate this risk, we followed a methodical development approach and conducted numerous tests and validations, including comparing the EC calculation to the same measure generated by the EclEmma tool [3].

The number of projects considered in the experiments is an external threat that may compromise the generality of our findings. Systems with similar development teams, domains, architecture styles, and sizes are more likely to produce similar results. To address this issue, we selected systems with distinct properties (e.g., domain and size).

[3]<https://www.eclemma.org/>

We adopted a training dataset of nine different open-source projects of various sizes and scopes to train our predictive models. However, four of the nine Java projects belong to the Apache Software Foundation. This might introduce some bias in the generalizability of the results. The reason for this configuration is the difficult to find appropriate projects to be used. We spent a great deal of effort looking for open-source projects with meaningful test suites that are operational and we continue this search. On the other hand, the selected Apache projects have many contributors that differ from one project to another, thus opening the window of different coding and testing styles. Hence, in the future, we plan to enlarge this training dataset by including more projects. Nevertheless, we consider that such selected systems represent, at minimum, other Java projects of different sizes, and our experimental results are generalizable to other projects.

# 6 RELATED WORK

## 6.1 Code Coverage Tools

This section will discuss the main works that propose as a solution to calculate the exact value of one or more coverage criteria.

Cobertura is a free open source Java tool that calculates the percentage of code accessed by tests (DOLINER et al., 2005). It may be used to identify parts of a Java application that are not covered by the tests. This tool calculates the node coverage value and also the edge coverage value. The node coverage (line coverage) metric shows how many statements are executed in the Unit Test run, while the edge coverage (branch) metric focuses on how many branches are covered by a set of tests. To start calculating code coverage in a Java project, one needs to declare the Cobertura Maven plugin in the pom.xml file. Besides the code coverage, this Java tool also shows the McCabe cyclomatic code complexity of each class and the average cyclomatic code complexity for each package and for the overall product.

JaCoCo is a free open source code coverage library for Java developed by the EclEmma team (ECLEMMA, 2013). JaCoCo runs as a Java agent, and it is responsible for instrumenting the bytecode while running the tests. This Java tool drills into each instruction and shows which lines are exercised during each test. JaCoCo essentially provides three important metrics: Node coverage (line coverage), Edge coverage (branch coverage) and Cyclomatic complexity. To run JaCoCo and calculate these metrics, it is necessary to declare this maven plugin in the pom.xml file.

Although JaCoCo and Coverage are the most popular Java code coverage tools, there are several others. For Python programs, the best tool for measuring code coverage (branch and node) is Coverage.py (BATCHELDER, 2009), which generally measures while running the tests. In order to calculate the code coverage value of a given program, this tool monitors it, noting which parts of the code have been executed, then analyzes the source to identify code that could have been executed but was not.

Lastly, PESTT (GAMEIRO; MARTINS, 2012) is an open-source Eclipse plugin for unit testing of Java methods. This tool supports the testing of methods based on the CFG. PESTT differs from other coverage analysis testing tools because it is not just a node and branch coverage analyzer. More than that, it supports many other "more powerful" coverage criteria: Edge-Pair coverage, Prime Path coverage, Simple Round Trip coverage,

Complete Round Trip coverage, and Complete Path coverage. Unfortunately, this tool does not work anymore due to the technology dependency of third-party tools. Table 6.1 summarizes the significant information about the presented tools, describing the name, supported language and information about the levels of coverage measurement provided by the tools.

Table 6.1: Summary table of the code coverage tools.

| | | **Measurements** | | | |
|---|---|---|---|---|---|
| **Tool name** | **Language** | **NC** | **EC** | **Stronger criteria** | **Method-Level** |
| Cobertura | Java | X | X | - | Yes |
| JaCoCo | Java | X | X | - | Yes |
| Coverage.py | Python | X | X | - | Yes |
| PESTT | Java | X | X | X | Yes |

Source: The author

The most significant benefit of these presented tools is that they all offer a solution that calculates the exact code coverage value. However, while the major existing tools work reasonably well, except for PESTT, these tools have certain limitations. The foremost limitation of the code coverage tools is that they apply only to a limited set of programming languages; most tools support only one programming language. This limitation is a direct result of using instrumentation techniques to capture coverage information by monitoring program execution. The execution program is monitored by inserting probes into the source code before or during its execution. A probe is typically a few code lines that, when executed, generate a record or event that indicates that program execution has passed through the point where the probe is located. There are three kinds of instrumentation techniques: Byte code, source code and on the fly (YANG; LI; WEISS, 2009). This language limitation and the high effort required for the tool's natural evolution can be considered the leading cause of increasing academic papers with alternative solutions, such as using predictive models to estimate the value instead of calculating the exact value.

As mentioned above, several tools calculated the code coverage value, usually node coverage and edge coverage. With a similar purpose, there are also many mutation testing tools. Mutation testing is considered a powerful criterion to evaluate a test suite's quality, as it injects possible faults in a program emulating real programming mistakes and checks whether the test cases can detect those faults. The mutation score represents the best alternative to code coverage for measuring the ability to detect faults (Jia; Harman,

2011) and identifying test data that can be employed to find real faults (GEIST; OFFUTT; HARRIS, 1992). On the other hand, this technique is also known as an expensive evaluation method with limited application. A number of mutation testing tools has also been proposed: MuJava (MA; OFFUTT; KWON, 2005), PIT[1], MAJOR (Just; Schweiggert; Kapfhammer, 2011), Judy (Madeyski; Radyk, 2010), Jester (MOORE, 2001), and CREAM (DEREZIŃSKA; SZUSTEK, 2012).

## 6.2 Machine learning solutions

Durelli et al. (Durelli et al., 2019) presents a systematic mapping of the use of machine learning in software testing and report that the most investigated topics are on test case design, test oracle construction, test case evaluation, and test case refinement. In this context, a few authors have proposed the use of ML to estimate the mutation score of a test suite using different source data. Jalbert and Bradbury (JALBERT; BRADBURY, 2012) present an approach to predict the mutation score of a unit under test based on a combination of source code, test suite metrics, and node coverage information. The ML algorithm used is SVM and receives as input the mutants, as well as source code and test suite metrics. It should be noted that the mutation generation tool used does not exclude equivalent mutants. The authors classify all mutation scores as low (0%-33%), medium (33%-66%), or high (66%-100%). Consequently, the predictive model estimates the mutation score of an unknown unit of code as low, medium, or high. As a result, the model achieves an accuracy of 58.27% for classes and 54.82% for methods. Their model can only predict mutation scores in a cross-version scenario (new release of the same project used during the training phase).

With a similar purpose, Zhang et al. (Zhang et al., 2019) propose an approach called Predictive Mutation Testing, which uses the RF algorithm to build a classification model. The model can be used to predict whether any new mutant is killed or not based on the same set of features used in the training phase. The learner receives as input 14 metrics that are related to the three conditions to kill a mutant: execution, infection, and propagation. The training process is a robust phase that demands considerable computational effort. This phase involves various third-party tools and own applications for three different goals: mutant generation and execution, feature collection, and ML framework. The authors obtain the mutation scores by ignoring the influence of equivalent mutants

[1] <http://pitest.org/>

and label each mutant as killed or alive. The model presents a good performance and predicts whether a method-level mutant would be killed or remain alive by the new test suite for a new project (cross-project scenario) or release of a project (cross-version scenario).

Grano et al. (Grano; Palomba; Gall, 2019) investigate the feasibility of using production and test-code-quality indicators to build a model able to distinguish effective and non-effective test cases, measured using mutation score. As the dependent variable, they rely on PIT to collect the mutation score in percentage and assume all mutants not killed are possibly not-equivalent. From that, they classify all test cases as effective — i.e., test cases that have a mutation score falling within 0.8 and 1.0 — and non-effective, when the mutation score is falling within 0.0 and 0.5. Regarding the independent variables, they considered 67 factors of five dimensions: code coverage (only the node coverage), test smells, code metrics, code smells, and readability. They employ three algorithms (RF, KNN, SVM) and build two different models: one containing all the factors and one excluding the node coverage. Based on a cross-validation strategy (cross-version scenario only), they report a Mean Absolute Error (MAE) of 0.051 for the solution with all factors, and 0.137 for the dataset without node coverage factor.

Grano et al. (GRANO et al., 2019) also propose the use of regression analysis to predict the branch (edge) coverage of an automatically generated test set. To obtain the dependent variable for the training phase, they rely on RANDOOP and EvoSuite to generate the test suites and JaCoCo to calculate the edge coverage value. As independent variables, they use 79 metrics belonging to four different categories that represent the complexity of the program and might be correlated with the edge coverage that will be achieved by a given class under test. During the training stage, four different regression algorithms are considered: Huber Regression, SVM, Multilayer Perceptron, and RF. Experimental results for cross-validation scenario reveal that algorithm RF is the best model, achieving an average MAE of 0.15 and 0.21 for the test suites generated by, respectively, EvouSuite and RANDOOP.

Previous work has shown the high potential of using machine learning techniques to evaluate the quality of a test set, mainly for mutation testing. Still, current approaches may require too much effort from the development team to be used in a practical scenario. Indeed, multiple tools (for code analysis, test analysis, mutant generators, etc.) are still necessary to generate all inputs required by the devised predictive models. As mentioned, many of those third tools are technology-dependent and difficult to maintain.

# 7 CONCLUDING REMARKS

In this work, we investigated the use of regression analysis to predict prime-path coverage for a method-level test set. We used a KDD-based approach to explore the possibility of using a set of source-code and test metrics to infer the PPC value.

Regarding the metrics that make up the dataset, we performed a detailed metrics analysis, resulting in the selection of 12 source code metrics and one test coverage metric. Our training dataset consists of 3.034 instances from 9 different projects. This resulting dataset was applied in the training phase by employing four different regression algorithms to build three different models: SCM-based, SCM+EC-based and EC-based.

In order to get an even smaller set of metrics as input data for the predictive model, we select the TOP-10 feature importance metrics and apply all possible combinations of these metrics. As a result, the model that presents the lowest error rate uses nine source code metrics and edge coverage as input and the Random Forest Regression algorithm as the predictor. This best model obtained an MAE of 0.049 (4.9%) in the internal validation and 0.065 (6.5%) in the external validation.

## 7.1 Contributions

In addition to the discussions presented in the previous chapters, we present the main insights from our proposal and its contributions.

**PPC predictor.** As initially mentioned, we have not currently found any operational tool that calculates the exact value of PPC criteria. Our goal was to provide a solution that is not dependent on some programming language and requires minimal maintenance effort. Although we adopted only the three models that obtained the best results in the external validation, all generated models are available online in our replication package[1]. Thus, as the main result of this work, we offer different predictive models that use different metrics as input, not restricting the use of our models to any particular metric extraction tool.

**Public dataset.** Building the dataset was undoubtedly the most challenging and demanding step, because there is no working and reliable tool that calculates the PPC value. Therefore, we believe the availability of the dataset, both the raw and training datasets, is an important contribution to this work. Our dataset was generated using known

[1] https://github.com/keslleylima/PPC-Prediction

real-world projects, and the whole process of collection and integration was carefully performed. Therefore, our dataset can be used to support diverse challenges in current research. One promising research direction is association analysis to discover common patterns — i.e., a relationship that describes powerfully associated features in the data. Our dataset structure can be easily adapted to support other research questions. The proposed dataset structure involves the PPC value of a test method. However, one can process the data and generate the PPC value of a test suite at the method level, i.e., the test coverage obtained by all test methods related to a single application method under test.

**Tools support.** An important contribution of our work is that we developed different auxiliary tools that can be used for different purposes. We made available the following solutions through this work: 1) Tool that generates test requirements for different graph coverage criteria. 2) Tool that generates test paths for Java projects. 3) Data integration tool, which can be easily adaptable to integrate different data sources. 4) Tool that calculates the PPC and EC value using the TP and TR files as input. It can also be adapted to incorporate other coverage criteria. At least, 5) Script to obtain the best predictive model (i.e., the best combination of features) using as input a set of metrics, ML algorithm, and a prediction performance metric. Some of these tools were developed in the scope of a final project and a research project of undergraduate students in the Computer Science course at UFRGS (NIEMIEC; SILVA; COTA, 2021). The students were co-advised by the author.

**Relationship between source code metrics and PPC value.** It is widely known the strong relationship between two of these criteria: PPC and EC. On the other hand, the relationship between source code metrics and PPC has not been consolidated. As we saw in our results, despite promising results in internal and external validation, the predictive model that uses only source code metrics does not have any metric that directly relates to the PPC value. Thus, a good model that uses only source code metrics as input data tends to need more features than one that also uses test coverage metrics. Even so, a model that uses only source code metrics, such as the one we created, is easier to use since many working tools generate a wide variety of these metrics.

## 7.2 Future Work

Future efforts will include both a horizontal and vertical extension of our work: with the former, we intend to expand the training dataset even further; with the latter, we

intend to detect even more sophisticated features to improve the model's precision. Our goal in the future is to increase the dataset size and the average cyclomatic complexity. It is a big challenge to achieve this due to project selection; the projects with appropriate tests usually follow good coding standards and best practices, keeping high cohesion and low coupling methods. Moreover, additional potential work is to explore different regression algorithms, focusing more on hyperparameter tuning activities.

Regarding possible future work focused on software testing, we consider that demonstrations of how the tester/developer can apply the model during the software development and maintenance process can contribute to the model's adoption — for example, simulating how the tester can use the PPC value in practice to improve test suite quality. Another opportunity for future work is to use projects in different programming languages and not only in Java.

## REFERENCES

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. 2. ed. [S.l.]: Cambridge University Press, 2016.

ANGUITA, D. et al. The 'k'in k-fold cross validation. In: I6DOC. COM PUBL. **20th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN)**. [S.l.], 2012. p. 441–446.

BATCHELDER, N. **Coverage.py — Coverage.py 4.5.4 documentation**. 2009. <https://coverage.readthedocs.io/en/v4.5.x/>. (Accessed on 02/01/2021).

BREIMAN, L. Random forests. **Machine learning**, Springer, v. 45, n. 1, p. 5–32, 2001.

CHANG, C.-C.; LIN, C.-J. Libsvm: A library for support vector machines. **ACM transactions on intelligent systems and technology (TIST)**, Acm, v. 2, n. 3, p. 27, 2011.

CHEN, M.-S.; HAN, J.; YU, P. S. Data mining: an overview from a database perspective. **IEEE Transactions on Knowledge and data Engineering**, IEEE, v. 8, n. 6, p. 866–883, 1996.

DEBRAY, T. P. et al. A new framework to enhance the interpretation of external validation studies of clinical prediction models. **Journal of clinical epidemiology**, Elsevier, v. 68, n. 3, p. 279–289, 2015.

DEREZIŃSKA, A.; SZUSTEK, A. Object-oriented testing capabilities and performance evaluation of the c# mutation system. In: SZMUC, T.; SZPYRKA, M.; ZENDULKA, J. (Ed.). **Advances in Software Engineering Techniques**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 229–242.

DOLINER, M.; OTHERS. **Cobertura - a code coverage utility for java**. 2005. <https://cobertura.github.io/cobertura/>. (Accessed on 01/31/2021).

DURELLI, V.; DELAMARO, M.; OFFUTT, J. An experimental comparison of edge, edge-pair, and prime path criteria. **Science of Computer Programming**, v. 152, p. 99 – 115, 2018.

Durelli, V. et al. Machine learning applied to software testing: A systematic mapping study. **IEEE Transactions on Reliability**, v. 68, n. 3, p. 1189–1212, 2019.

ECLEMMA. **JaCoCo Java Code Coverage Library**. 2013. <https://www.eclemma.org/jacoco/>. (Accessed on 01/31/2021).

FAYYAD, U.; PIATETSKY-SHAPIRO, G.; SMYTH, P. From data mining to knowledge discovery in databases. **AI magazine**, v. 17, n. 3, p. 37–37, 1996.

GAMEIRO, R.; MARTINS, F. **PESTT EDUCATIONAL SOFTWARE TESTING TOOL**. Dissertation (Master) — LaSIGE & University of Lisbon, Faculty of Sciences, 2012.

GEIST, R.; OFFUTT, J.; HARRIS, F. Estimation and enhancement of real-time software reliability through mutation analysis. **IEEE Transactions on Computers**, IEEE, n. 5, p. 550–558, 1992.

GILBERT, D. The jfreechart class library. **Developer Guide. Object Refinery**, v. 7, 2002.

Grano, G.; Palomba, F.; Gall, H. Lightweight assessment of test-case effectiveness using source-code-quality indicators. **IEEE Transactions on Software Engineering**, 2019. ISSN 2326-3881.

GRANO, G.; PALOMBA, F.; GALL, H. C. Lightweight assessment of test-case effectiveness using source-code-quality indicators. **IEEE Transactions on Software Engineering**, IEEE, 2019.

GRANO, G. et al. Branch coverage prediction in automated testing. **Journal of Software: Evolution and Process**, 2019.

HAILPERN, B.; SANTHANAM, P. Software debugging, testing, and verification. **IBM Systems Journal**, IBM, v. 41, n. 1, p. 4–12, 2002.

HOSMER, D.; LEMESHOW, S.; STURDIVANT, R. **Applied logistic regression**. [S.l.]: John Wiley & Sons, 2013.

JALBERT, K.; BRADBURY, J. Predicting mutation score using source code and test suite metrics. In: **Proceedings of the First International Workshop on Realizing AI Synergies in Software Engineering**. [S.l.]: IEEE Press, 2012. (RAISE '12), p. 42–46.

Jia, Y.; Harman, M. An analysis and survey of the development of mutation testing. **IEEE Transactions on Software Engineering**, v. 37, n. 5, p. 649–678, Sep. 2011. ISSN 2326-3881.

JUST, R. et al. Are mutants a valid substitute for real faults in software testing? In: **Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering**. [S.l.: s.n.], 2014. p. 654–665.

Just, R.; Schweiggert, F.; Kapfhammer, G. Major: An efficient and extensible tool for mutation analysis in a java compiler. In: **2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)**. [S.l.: s.n.], 2011. p. 612–615.

KIRK, M. **Thoughtful machine learning: A test-driven approach**. [S.l.]: " O'Reilly Media, Inc.", 2014.

LAFITA, A. et al. Biojava 5: A community driven open-source bioinformatics library. **PLoS computational biology**, Public Library of Science, v. 15, n. 2, p. e1006791, 2019.

Li, N.; Praphamontripong, U.; Offutt, J. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In: **2009 International Conference on Software Testing, Verification, and Validation Workshops**. [S.l.: s.n.], 2009. p. 220–229.

LIU, H.; TAN, H. B. K. Covering code behavior on input validation in functional testing. **Information and Software Technology**, Elsevier, v. 51, n. 2, p. 546–553, 2009.

MA, Y.-S.; OFFUTT, J.; KWON, Y. R. Mujava: an automated class mutation system. **Software Testing, Verification and Reliability**, v. 15, n. 2, p. 97–133, 2005.

Madeyski, L.; Radyk. Judy - a mutation testing tool for java. **IET Software**, v. 4, n. 1, p. 32–42, 2010.

MALL, R. **Fundamentals of software engineering**. [S.l.]: PHI Learning Pvt. Ltd., 2018.

MCCABE, T. J. A complexity measure. **IEEE Transactions on software Engineering**, IEEE, n. 4, p. 308–320, 1976.

MOORE, I. Jester-a junit test tester. **2nd International Conference on eXtreme Programming and Flexible Processes in Software Engineering (XP)**, 2001.

NEJMEH, B. A. Npath: a measure of execution path complexity and its applications. **Communications of the ACM**, ACM New York, NY, USA, v. 31, n. 2, p. 188–200, 1988.

NIEMIEC, W.; SILVA, K.; COTA, E. Executionflow: A tool to compute test paths of java methods and constructors. In: . New York, NY, USA: Association for Computing Machinery, 2021. (SBES '21). ISBN 9781450390613. Available from Internet: <https://doi.org/10.1145/3474624.3476014>.

PINTO, L. S.; SINHA, S.; ORSO, A. Understanding myths and realities of test-suite evolution. In: **Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering**. [S.l.: s.n.], 2012. p. 1–11.

ROMANO, S.; SCANNIELLO, G. Smug: a selective mutant generator tool. In: IEEE. **2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)**. [S.l.], 2017. p. 19–22.

SAMMUT, C.; WEBB, G. I. **Encyclopedia of machine learning and data mining**. [S.l.]: Springer, 2017.

SILVA, K.; COTA, E. Predicting prime path coverage using regression analysis. In: **Proceedings of the 34th Brazilian Symposium on Software Engineering**. [S.l.: s.n.], 2020. p. 263–272.

SPINELLIS, D. Tool writing: a forgotten art?(software tools). **IEEE Software**, IEEE, v. 22, n. 4, p. 9–11, 2005.

STONE, M. Cross-validatory choice and assessment of statistical predictions. **Journal of the Royal Statistical Society: Series B (Methodological)**, Wiley Online Library, v. 36, n. 2, p. 111–133, 1974.

STRUG, J.; STRUG, B. Machine learning approach in mutation testing. In: **Testing Software and Systems**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 200–214.

STRUG, J.; STRUG, B. Classifying mutants with decomposition kernel. In: **Artificial Intelligence and Soft Computing**. [S.l.]: Springer International Publishing, 2016. p. 644–654. ISBN 978-3-319-39378-0.

Strug, J.; Strug, B. Using classification for cost reduction of applying mutation testing. In: **2017 Federated Conference on Computer Science and Information Systems (FedCSIS)**. [S.l.: s.n.], 2017. p. 99–108.

TAN, P.-N.; STEINBACH, M.; KUMAR, V. **Introduction to data mining**. [S.l.]: Pearson Education India, 2016.

VARELA, A. et al. Source code metrics: A systematic mapping study. **Journal of Systems and Software**, v. 128, 04 2017.

VIRGÍNIO, T. et al. Jnose: Java test smell detector. In: **Proceedings of the 34th Brazilian Symposium on Software Engineering**. [S.l.: s.n.], 2020. p. 564–569.

VIRGÍNIO, T. et al. On the influence of test smells on test coverage. In: **Proceedings of the XXXIII Brazilian Symposium on Software Engineering**. [S.l.: s.n.], 2019. p. 467–471.

WOODWARD, M. R.; HENNELL, M. A.; HEDLEY, D. A measure of control flow complexity in program text. **IEEE Transactions on Software Engineering**, IEEE, n. 1, p. 45–50, 1979.

YANG, Q.; LI, J. J.; WEISS, D. M. A survey of coverage-based testing tools. **The Computer Journal**, OUP, v. 52, n. 5, p. 589–597, 2009.

Zhang, J. et al. Predictive mutation testing. **IEEE Transactions on Software Engineering**, v. 45, n. 9, p. 898–918, 2019.

# APPENDIX A — SELECTED JAVA PROJECTS

## A.1 Projects used in the training phase

BioJava[1] is a mature open-source project that provides a Java library framework for the processing of biological data. This project aims to simplify bioinformatics interpretations by implementing data structures, parsers, and routines for everyday tasks in ontologies, genomics, structural biology, and more (LAFITA et al., 2019). For instance, BioJava includes robust analysis and statistical routines for parsing standard file formats and packages for manipulating sequences and 3D structures.

Commons Math[2] is a library that contains mathematics and statistics components addressing the most common difficulties not support in the Java programming language or Commons Lang. Apache Commons Math is the most significant open-source library of mathematical functions and utilities for Java. Because of its relevance, this project is widely used in different software engineering studies.

Another popular Apache library is Commons lang[3], which repertoire is pretty rich, providing a complete package of utility classes for java.lang API. It supports array and number manipulation, string manipulation methods, reflection, and concurrency.

JFreeChart[4] is a free 100% Java chart library that makes it simple for developers to display quality charts in their applications. This library supports the generation of different charts, such as bar charts (horizontal, vertical, regular, and 3D-effect), line charts, pie charts, time series charts, Gantt charts, scatter plots, and symbol charts (GILBERT, 2002).

Checkstyle[5] is a development tool to help programmers write Java code by checking a source code against a configurable set of rules (best practices). This tool can check several aspects of a given source code, for example, formatting issues, code layout, class, and method design problems.

Regarding selecting the last 5 projects described above — i.e., Biojava, commons-lang, commons-math,Jfreechart, and Checkstyle — we picked them because they are Java projects widely used by recent mutation testing studies (JUST et al., 2014; ROMANO; SCANNIELLO, 2017) and test code coverage studies (VIRGÍNIO et al., 2019; GRANO

[1] <https://github.com/biojava/biojava>
[2] <https://commons.apache.org/proper/commons-math/>
[3] <https://commons.apache.org/proper/commons-lang/>
[4] <https://www.jfree.org/jfreechart/>
[5] <https://checkstyle.sourceforge.io/>

et al., 2019; GRANO; PALOMBA; GALL, 2019). Moreover, these are projects that have different sizes, domains and diversity of method complexity value.

Apache Commons Text[6] is a library focused on algorithms working on strings, containing many useful utility functions for working with strings beyond what the Java language offers.

Apache Dubbo[7] is a high-performance, Java-based open-source remote procedure call framework from Alibaba. Among other things, it helps enhance service governance and makes it possible for a traditional monolith application to be refactored smoothly to a scalable distributed architecture.

Exp4j[8] is a mathematical expression evaluator for the Java programming language. It implements Dijkstra's Shunting-yard algorithm to translate expressions from infix notation to Reverse Polish notation and calculates the result using a simple Stack algorithm. Using this library, the developers can create and use their custom operators and functions.

Finally, Airship Java Client Library[9] is the official supported Java library for Urban Airship Connect. It is a Java library for using Airship's messaging platform and related features. Airship is a platform for customer engagement, lifecycle marketing, and analytics and data solutions.

## A.2 Projects used in the external validation

Apache Accumulo[10] is a sorted, distributed key/value store that provides robust, scalable data storage and retrieval. With Accumulo, developers can store and manage large data sets across a cluster.

Apache Commons-io[11] is the Apache Commons components derived from Java API and provides different utility classes for File IO's ordinary operations, covering a wide range of use cases. It contains file filters, file comparators, utility classes, stream implementations, and other uses that help develop IO functionality.

At least, Apache Tika is a toolkit for detecting and extracting metadata and structured text content from different documents using existing parser libraries.

[6] <https://commons.apache.org/proper/commons-text/>
[7] <https://dubbo.apache.org/zh/>
[8] <https://github.com/fasseg/exp4j>
[9] <https://github.com/urbanairship/java-library>
[10] <https://github.com/apache/accumulo>
[11] <https://commons.apache.org/proper/commons-io/>

# APPENDIX B — RESUMO ESTENDIDO

## Prevendo o valor de cobertura de caminhos primo por meio da análise de regressão em nível do método

A abordagem da engenharia de *software* no processo de desenvolvimento fornece uma solução de alta qualidade a um preço justo. Uma das atividades deste desenvolvimento é o teste de *software*, estimado em pelo menos cerca de metade de todo o custo de desenvolvimento (HAILPERN; SANTHANAM, 2002). Apesar dos altos custos, o teste de software desempenha um papel vital no processo de desenvolvimento de software para garantir a qualidade e confiabilidade do *software*.

O teste de software visa exercitar o comportamento do software para evitar *bugs* futuros, revelando o máximo de falhas usando a menor quantidade de recursos. No entanto, medir a qualidade do teste é difícil porque sua eficácia depende da qualidade do conjunto de testes: alguns conjuntos são melhores na detecção de falhas do que outros. O poder de uma suíte de teste refere-se ao seu potencial para detectar falhas: uma suíte de potência forte pode detectar mais *bugs*. Consequentemente, para um teste eficaz, a noção de critérios de cobertura de teste é importante porque fornece um meio de medir esse potencial.

Um critério de cobertura de teste é uma regra ou grupo de regras que descreve os requisitos de teste (TR); cada requisito é um elemento de software específico que um caso de teste deve satisfazer ou cobrir. Ammann e Offutt (AMMANN; OFFUTT, 2016) compilam vários critérios definidos na literatura em quatro estruturas genéricas que podem ser usadas para representar (ou modelar) um software em teste (SUT), ou seja, domínio de entrada, grafos, expressões lógicas e gramáticas. A estrutura de teste mais popular é a de grafos, e os critérios baseados em grafos exigem que o testador cubra o grafo de alguma forma. Os critérios mais comuns baseados em grafos são a cobertura de nó (NC) e a cobertura da arestas (EC), que podem ser medidas por muitas ferramentas de teste disponíveis para a maioria das linguagens de programação. Esses dois critérios podem ser considerados triviais e geram os requisitos mínimos de teste para um conjunto de testes. Em contraste, existem outros critérios capazes de explorar usos mais intrincados do software, como os critérios que cobrem os caminhos. As informações sobre a cobertura de teste ajudam a equipe de desenvolvimento a encontrar pontos fracos e redundâncias no conjunto de testes atual. Assim, a adoção de critérios de cobertura adequados tem várias

vantagens para melhorar a qualidade do software e controlar o custo dos testes.

Apesar dos benefícios mencionados, existem dificuldades relacionadas à mensuração de critérios mais poderosos. Por exemplo, para avaliar a cobertura baseada em grafos de um conjunto de testes, é necessário primeiro gerar o grafo que representa o SUT, em seguida, extrair os requisitos de teste para um critério específico e, em seguida, traçar o caminho de execução neste grafo fornecido por cada caso de teste. Os requisitos de teste cobertos pelos caminhos de testes são contados e, ao final deste processo, a cobertura para aquele critério pode ser calculada.

A execução manual desse processo é normalmente muito cara, enquanto sua automação tem se restringido basicamente à cobertura de nó e arestas. Atualmente, esse processo é fortemente dependente da tecnologia, pois requer conhecimento e interação com o ambiente de compilação e execução do SUT.

Durelli et al. (DURELLI; DELAMARO; OFFUTT, 2018) compara a eficácia e o custo de três critérios de cobertura de grafo: cobertura de arestas, cobertura de par de arestas e cobertura de caminhos primo (PPC). Como resultado, eles inferiram que o PPC é mais eficaz, especialmente em programas que têm fluxos de controle complicados. Em termos de custo, eles concluíram que não há muita diferença em termos de número de TRs, mas o PPC leva a TRs muito mais inviáveis do que a cobertura de arestas e pares de arestas.

Para mitigar os desafios das ferramentas dependentes de tecnologia e ajudar os testadores a melhorar suas suítes de teste, trabalhos recentes propuseram a estimativa do valor de cobertura ao invés de seu cálculo exato.

Dado este contexto, consideramos o PPC como um critério valioso porque 1) subsume a maioria dos critérios baseados em grafos (AMMANN; OFFUTT, 2016); 2) fornece informações mais precisas do que os critérios básicos; e 3) permite que o testador avalie o alcance e as omissões do conjunto de testes com mais convicção. Ainda assim, hoje em dia, não existem ferramentas práticas para apoiar o testador na avaliação do conjunto de testes com base neste critério. Acreditamos que haja duas razões principais para isso. Em primeiro lugar, construir e manter uma ferramenta de cobertura de teste usando o critério PPC é um desafio devido à dependência tecnológica necessária de ambientes de desenvolvimento integrado (IDEs) e ferramentas de terceiros. Em segundo lugar, como poucos trabalhos mostram evidências práticas sobre as vantagens do uso do critério PPC, os testadores acreditam que critérios mais fracos são suficientes e a cultura para seu uso não é criada. Para preencher essa lacuna, investigamos neste trabalho o uso de algoritmos

de ML para prever o valor de PPC de uma suíte de testes em nível de método.

A partir do framework KDD, propõe-se uma metodologia para a geração de modelos preditivos para PPC, utilizando como entrada métricas de código fonte. A metodologia parte da análise de métricas de código fonte e sua relação com o objetivo da predição, com o objetivo de escolher um sub-conjunto mais adequado para o problema em mãos. Assim, foi definido um conjunto de 12 métricas que podem ser facilmente extraídas por diversas ferramentas. Neste trabalho, utilizou-se a ferramenta *Understand* para essa extração. Como não existe nenhuma ferramenta em funcionamento que calcula o valor exato de cobertura de PPC, nós desenvolvemos uma abordagem baseada em diferentes ferramentas para obter o valor desta métrica de teste necessária para o treinamento do modelo. Após a geração do *dataset*, foram utilizados diferentes algoritmos de regressão com o objetivo de identificar o melhor modelo preditivo para prever PPC. A partir dos experimentos de validação, pode-se concluir que a metodologia proposta permite a geração de modelos preditores para PPC. As contribuições deste trabalho são resumidas a seguir:

- Analisamos a relação entre as métricas de código-fonte e o PPC de um conjunto de testes no nível do método;
- propomos uma metodologia para apoiar o testador na avaliação do poder de um conjunto de testes usando um modelo de predição e um critério mais forte;
- Validamos a metodologia proposta realizando experimentos para avaliar o modelo preditivo em um cenário realista;
- Mostramos que o PPC pode ser estimado usando métricas de código-fonte e métricas básicas de cobertura de teste e fornecemos os modelos preditivo desenvolvidos;
- Fornecemos um conjunto de dados público que envolve o critério PPC em projetos de código aberto, disponibilizando-o online.