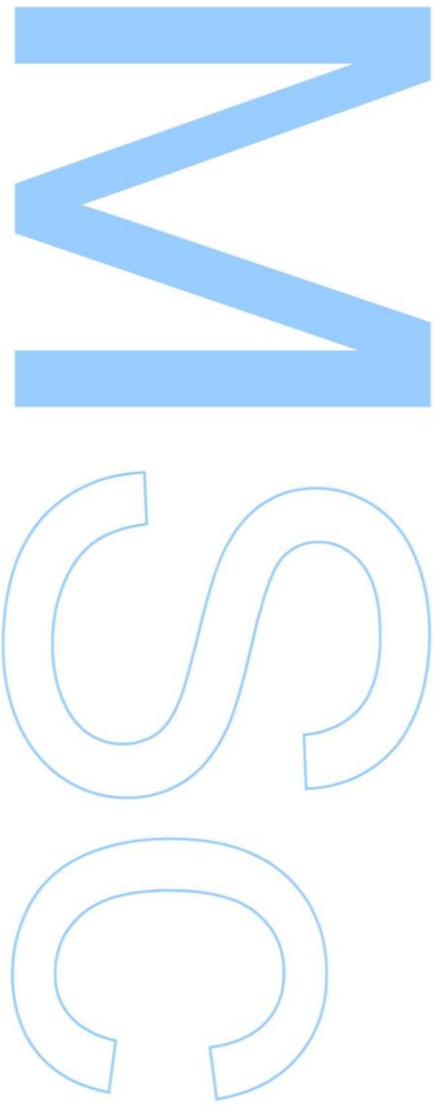
# Data Mining for Automatic Generation of Software Tests

Alberto Plácido Oliveira
Mestrado em Ciência de Computadores
Departamento de Ciência de Computadores
2020

**Orientador**
Alípio Mário Guedes Jorge, Professor, FCUP

**Coorientador**
Vítor Amorim, Colaborador, Randtech Computing




Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

# Agradecimentos

Em primeiro lugar, gostaria de agradecer à FCUP e aos seus docentes, que me forneceram mais conhecimentos e competências para enfrentar os desafios da minha vida profissional. Um agradecimento também a todos os meus colegas de curso que me ajudaram nos trabalhos de grupo e não só, sem os quais não seria possível estar aqui hoje. Uma palavra especial ao professor Alípio pela oportunidade proporcionada e por todo o apoio que sempre me deu. Um muito obrigado a todos os que fizeram parte deste projecto, ao Vítor Amorim, ao Ricardo Freitas, à professora Ana Paiva e ao professor Paulo Azevedo. Uma palavra de gratidão também para o Rui Maranhão e para o Arian Pasquali.

De forma mais pessoal, queria agradecer à minha família e à minha namorada, que sempre me suportou, apoiou e ajudou a fazer de mim o que sou hoje.

# Abstract

Software development is a complex and continuous process that requires frequent changes in the code [1]. Each change can introduce errors that affect the ability of the software maker to timely deliver a quality product [2].

To address this issue, test cases are developed and executed before each release [3]. Designing these tests manually, however, is prone to error, considering it demands a lot of time, costs and effort of human software testers [4]. Automating the testing processes is therefore a priority in order to reduce the high costs of a manual approach and guarantee that the test cases properly cover all requirements, establishing a high degree of confidence for a software system in terms of quality [5].

When the software is already in production, and has a graphical user interface (GUI), the information about how users interact with it (GUI event traces [6]) may serve as a source for test case generation, which can be used for new releases. User interaction data can be gathered and analytically exploited to generate and select the best test cases to perform [7]. This is the starting point to the work presented here.

Anywhere+ is an enterprise resource planning (ERP) platform, developed by RandTech Computing[1], with the purpose of simplifying the organization of information of an insurance company. With the expansion of clients and functionalities, a manual testing approach to ensure the system's quality became unfeasible. To automate the testing process, a tool for generating software test cases from user interaction data was developed. It relies on four steps: storing user interaction logs from the graphical user interface controls, captured by a browser plugin, processing these logs to discover sequential micro-patterns, merging

[1] `https://rtcom.pt`

them into a global Markov chain model and lastly, automatically generating test cases from this model. The evaluation of the tool is then performed to overview the effectiveness of this approach, which can be easily applied to other similar continuous software system's development processes. The obtained results point out a different number of generated test cases to obtain optimal values for each metric. Also, we were able to improve the process of generating test cases by introducing changes on the way test sequences are created. To further disseminate the results obtained from this work, we produced a scientific paper describing the implementation of this framework, which has been accepted and published in the IDEAL2020 conference [8]. Also, a demonstration of the tool's functioning was presented at the INTUITESTBEDS2020 conference [9].

**Keywords**: Software Testing, Frequent Pattern Mining, Markov Chains, Data Mining

# Resumo

O desenvolvimento de software é um processo complexo e contínuo que requere alterações frequentes no código [1]. Cada alteração pode introduzir erros que afetam a capacidade dum programador de software entregar atempadamente um produto de qualidade [2].

Para resolver este problema, são desenvolvidos e executados casos de teste antes de cada release. No entanto, desenvolver estes testes manualmente é uma tarefa propensa a erros, considerando que exige bastante tempo e esforço por parte de software testers humanos [4]. Automatizar os processos de teste é portanto uma prioridade de forma a reduzir os custos de uma abordagem manual e garantir que os casos de testes cobrem todos os requisitos, estabelecendo um alto grau de confiança para um sistema de software em termos de qualidade [5].

Quando o software já se encontra em produção e tem uma interface gráfica, a maneira como os utilizadores interagem com o sistema pode servir como fonte para gerar casos de teste [6], que podem usados para novas releases. Os dados de interação dos utilizadores podem ser recolhidos e explorados analiticamente para gerar e selecionar os melhores casos de testes para realizar [7]. Este é o ponto inicial para o trabalho apresentado aqui.

A plataforma Anywhere+ é um sistema ERP (enterprise resource planning, em português, sistema integrado de gestão empresarial), desenvolvido pela RandTech Computing[2], com o objetivo de simplificar a organização da informação de uma companhia de seguros. Com a expansão de clientes e funcionalidades, uma abordagem manual de testes para garantir a qualidade do sistema é inviável. Para automatizar o processo de testes, foi desenvolvida

[2] https://rtcom.pt

uma ferramenta para geração de testes de software a partir de dados de interação de utilizadores. Esta ferramenta está assente em quatro passos: guardar os logs das interações dos utlizadores no sistema a partir da interface gráfica, capturando-as através dum plugin no browser, processar estes logs para descobrir micro padrões sequenciais, criando um modelo de Markov global a partir da fusão destes micro padrões e, por último, gerar casos de teste automaticamente a partir deste modelo. A avaliação da ferramenta é então feita para se perceber a eficácia desta abordagem, que pode ser facilmente aplicada a outros processos semelhantes de desenvolvimento de software. Os resultados obtidos mostram que, para cada métrica, é necessário gerar um número diferente de casos de teste para se obterem valores ótimos. Foram também introduzidas melhorias no processo de geração de casos de teste através de mudanças na forma como são criadas as sequências de teste. Para disseminação dos resultados obtidos a partir deste trabalho, foi produzido um artigo científico que descreve a implementação desta framework. Este artigo foi aceite e publicado na conferência IDEAL2020 [8]. Foi também feita uma demonstração do funcionamento da ferramenta na conferência INTUITESTBEDS2020 conference [9].

**Palavras-chave**: Software Testing, Frequent Pattern Mining, Markov Chains, Data Mining

# Contents

**Abstract** **2**

**Resumo** **4**

**List of Tables** **10**

**List of Figures** **12**

**Acronyms** **13**

**1 Introduction** **15**

1.1 Context . . . . . 15

1.2 Motivations and Objectives . . . . . 16

1.3 Results and Contributions . . . . . 16

1.4 Structure . . . . . 17

**2 Machine Learning for Software Engineering** **18**

2.1 Supervised learning . . . . . 19

2.1.1 Naïve Bayes . . . . . 20

2.1.2 Decision Tree . . . . . 20

2.1.3 Random Forest . . . 22
2.1.4 Neural Networks . . . 22
2.1.5 $K$-Nearest Neighbors . . . 23
2.1.6 Logistic Regression . . . 24
2.2 Unsupervised learning . . . 25
2.2.1 Apriori . . . 25
2.2.2 $K$-means . . . 26

**3 Data driven software testing** **28**
3.1 Software Testing . . . 28
3.2 Black-box and White-box Testing . . . 32
3.2.1 Black-box Testing . . . 32
3.2.2 White-box testing . . . 34
3.3 Data driven software test generation . . . 36
3.3.1 Random/Monkey Testing . . . 36
3.3.2 Info-fuzzy networks . . . 37
3.3.3 Markov chain models . . . 38
3.3.4 Genetic Algorithm . . . 39

**4 Automated testing from user sessions** **41**
4.1 Data Driven GUI Software Testing . . . 41
4.1.1 Unit Testing . . . 42
4.1.2 Random/Monkey Testing . . . 43
4.1.3 Capture/Replay Testing . . . 43

4.1.4 Model-based GUI Testing . . . . . 44
4.2 Sequence Pattern Mining . . . . . 45
4.2.1 Horizontal approaches . . . . . 46
4.2.2 Vertical approaches . . . . . 48

**5 The use case** **50**

5.1 Framework structure . . . . . 50
5.2 Data acquisition . . . . . 52
5.2.1 Data Format . . . . . 53
5.2.2 XHTML Transformation . . . . . 54
5.2.3 Logging the GUI events . . . . . 54
5.3 Analysis Module . . . . . 57
5.3.1 Discovering frequent sub sequences . . . . . 57
5.3.2 Building the Markov Model . . . . . 58
5.3.3 Using the Markov Model to generate tests . . . . . 58
5.4 Deployment . . . . . 60

**6 Evaluation** **62**

6.1 PAC . . . . . 63
6.2 Kullback-Leibler's divergence . . . . . 64
6.3 Chi-square . . . . . 65
6.4 DDU . . . . . 67
6.4.1 Density . . . . . 68
6.4.2 Diversity . . . . . 68

6.4.3 Uniqueness . . . . . 69

6.4.4 Results . . . . . 69

6.5 Sequences' generation improvements . . . . . 70

**7 Conclusions** **73**

7.1 Future Work . . . . . 74

**References** **75**

# List of Tables

2.1 Software engineering problems and some machine learning algorithms to solve them . . . . . 27

6.1 PAC for each N parameter . . . . . 63

6.2 Kullback-Leibler's divergence results . . . . . 66

6.3 Comparison of PAC for the original and modified methods . . . . . 71

# List of Figures


3.1 Phases of the software testing life cycle . . . . . 29

3.2 Software development stages and its testing processes [10] . . . . . 31

3.3 Info fuzzy network example [11] . . . . . 37

3.4 Markov chain model example . . . . . 39

4.1 Database formats for sequence pattern mining [12] . . . . . 46

5.1 Software Deployment Framework . . . . . 51

5.2 Snippet of the Login.xhtml file of the application . . . . . 53

5.3 Output from XHTML generator to allow plugin recognition . . . . . 55

5.4 Final transformed HTML for plugin recognition . . . . . 56

5.5 Captured GUI interaction record . . . . . 57

5.6 JSON file representing a set of possible outcomes and their probabilities . . . 59

5.7 Generating Sequences Method . . . . . 60

6.1 Evaluation results for PAC . . . . . 64

6.2 Evaluation results for the Kullback-Leibler divergence . . . . . 65

6.3 Evaluation results for the chi square metric . . . . . 67

6.4 Histograms of PAC for the original and modified methods . . . . . 71

6.5 Histograms of chi-square for the original and modified methods . . . . . 72

6.6 Histograms of KL for the original and modified methods . . . . . 72

# Acronyms

| | |
|---|---|
| **CMAP** | Co-occurrences Map |
| **DDoS** | Distributed Denial of Service |
| **DDU** | Density Diversity Uniqueness |
| **EL** | Expression Language |
| **ERP** | Enterprise Resource Planning |
| **GSP** | Generalised Sequential Patterns |
| **GUI** | Graphical User Interface |
| **HTML** | Hypertext Markup Language |
| **IFN** | Info-fuzzy networks |
| **JAR** | Java ARchive |
| **JSF** | JavaServer Faces Engine |
| **JSON** | JavaScript Object Notation |
| **KL** | Kullback-Leibler |
| **MFS** | Mining Frequent Sequences |
| **PAC** | Proportion of Actions Covered |
| **PSP** | Prefix tree for Sequential Patterns |

| | |
|---|---|
| **SPADE** | Sequential PAttern Discovery using Equivalence classes |
| **SPAM** | Sequential PAttern Mining |
| **SPMF** | Sequential Pattern Mining Framework |
| **STM** | State Transition Matrix |
| **SUT** | System Under Test |
| **UI** | User Interface |
| **UML** | Unified Modeling Language |
| **XHTML** | eXtensible Hypertext Markup Language |
| **XPATH** | XML (eXtensible Markup Language) Path Language |

# Chapter 1

# Introduction

## 1.1 Context

Software development is a complex and continuous process that requires frequent changes in the code [1]. Each change can introduce errors that affect the ability of the software maker to timely deliver a quality product [2]. Errors in software can cause distrust in software users and can additionally also lead to heavy economic losses. For instance, the Mars Polar Lander shutdown in 1999 resulted in the loss of 165 million dollars [13]. Software errors can also result in the sacrifice of human lives, which was what happened in 2015, when the 2015 Seville Airbus A400M crashed during a test flight [14]. Taking into account that software development is becoming increasingly more agile [15], with systems undergoing constant changes, the moments for introducing errors are multiplying.

Typically, the software industry's response is based on test cases that are executed before each release [3]. Although the automation of test checking is a common practice [16], the design of test cases is still mostly based on human expertise [4]. However, manually devising software tests demands a lot of time, costs and effort of human software testers [4]. Correctly selecting the tests to perform and evaluating their outputs is crucial in order to improve the software's quality with less cost [4]. When the software is already in production, and has a graphical user interface (GUI), the information about how users interact with it (GUI event traces [6]) may also serve as a source for test case generation which can be used for new releases. User interaction data can be gathered and analytically exploited to

generate and select the best test cases to perform [7].

## 1.2 Motivations and Objectives

Anywhere+ is an insurance ERP (enterprise resource planning) platform developed by RandTech Computing[1] with the intent of simplifying the organization of information of an insurance company (such as insurance contracts, policies, claims, and even the general management of the institution). With the expansion of clients and functionalities, a manual testing approach to ensure the system's quality became unfeasible. In this thesis, we propose a data mining based method and tool that automates the generation of software test cases from user interaction data. This framework module can be easily applied to other similar continuous software systems' development processes. With this project, we are able to gather data from user sessions and use it to design and perform automatic test cases. This method can be utilized on correcting software errors for the current or next releases from the platform.

## 1.3 Results and Contributions

The final results obtained from this work contain the implementation of this framework, based on a four steps approach, and its deployment. The first step is to store the user interaction logs. This is done through a browser plugin which captures this data from the graphical user interface controls. Secondly, a sequence mining technique is applied to unravel sequential micro-patterns from this data. These patterns are then used to build a global Markov chain model and finally, this model is used to automatically generate test cases, according to these usage patterns. For the evaluation of its performance, the tool is tested on real data in terms of coverage and plausibility of the generated patterns.

While the framework can easily be adapted for other Web-based GUI software applications, the packaging of the tool for the deployment stage provides an easy way to distribute and use it. The application of sequence mining to the logs and the creation of a Markov model

[1] `https://rtcom.pt`

provides a good way to understand the system and how it is used. By implementing various evaluation measures, we were able to assess and tune the performance of the system, which led us to improve the process of test generation. To further disseminate the results obtained from this work, we produced a scientific paper describing the implementation of this framework, which has been accepted for the IDEAL 2020 conference [8]. Also, a demonstration of the tool's functioning will be presented at the INTUITESTBEDS2020 conference [9].

## 1.4 Structure

This document is structured into six main chapters. This first chapter provides an introduction to the developed work in terms of context, motivations and objectives. Chapter 2, "Machine Learning for Software Engineering", presents an overview on machine learning, its different methods and approaches and lastly, a link between machine learning and software engineering. On chapter 3, "Data driven software testing", this link is extended between machine learning and software testing. The definition of software testing, its space among the software development stages and its methodologies can also been seen in this chapter. Chapter 4, "Learning to simulate user sessions for automated testing", connects the extraction of user sessions through GUI testing and their processing using frequent pattern mining algorithms, building automated models for the generation of test cases. Chapter 5, "The use case", describes each step of the proposed methodology applied to the task of automatic generation of software tests for the Anywhere+ application and its deployment, followed by chapter 6, "Evaluation", where the evaluation process of this methodology is approached, along with the changes on the method for generating test sequences. In the last chapter, conclusions and future work are presented.

# Chapter 2

# Machine Learning for Software Engineering

Machine learning can be seen as a science that uses computer programs to find interesting behaviours and patterns in data coming from observations of real-world interactions. This can be used to improve the performance of some tasks through experience [17]. Machine learning encompasses five tribes: inductive reasoning, connectionism, evolutionary computation, Bayes' theorem and analogical modelling. Each tribe contains a different machine learning paradigm, with its advantages and disadvantages. They all contain a unique contribution to a "master algorithm", a unique learning algorithm able to learn regardless about the problem [18].

Machine learning techniques and algorithms can be used in a large variety of different domains (for example, health, economics, industry). They prove to be of great interest in, for instance, data mining problems where large databases contain valuable implicit patterns (considering machine learning needs large amounts of information to achieve good accuracy [17] [19]) or domains where programs must dynamically adapt to changing conditions [20]. In terms of software engineering, machine learning can be used to predict or estimate values for tasks such as software quality, reusability, cost, development effort, defects/testing and project risk / releases timing [21]. Most machine learning algorithms can be divided into the categories of supervised learning and unsupervised learning.

## 2.1 Supervised learning

In supervised learning, the goal is to learn a mapping from inputs $x$ to outputs $y$, given a labeled set of input-output pairs (defined as the training set) [22]. Each training input is a vector of fields, called features, attributes or covariates. These attributes contain different types of information in various formats. A practical example of a training input could be a person's report (containing, for instance, name, address, phone number). A training input (and the output/response variable) can also be a complex structured object (an image, a sentence, an email message, a time series, a graph). Despite this, most methods assume that the target variable is either a categorical variable from a defined finite set (such as male or female), or a real-valued number (such as income level). When the target is categorical, the problem is known as classification, and when the target is a real-valued number, the problem is known as regression. Another variant of regression problems occurs when the output set has some natural order. It is known as ordinal regression [22].

Over the next sections, we will describe in more detail some of the most common supervised learning algorithms, referred in the literature as used in software engineering applications and studies. These are:

- Naïve Bayes
- Decision Tree
- Random Forest
- Neural Networks
- K-Nearest Neighbors
- Logistic Regression

### 2.1.1 Naïve Bayes

A naïve Bayes classifier is a probabilistic machine learning model, used for classification task, on the Bayes theorem. The Bayes theorem states that [23]

$$\Pr(A|B) = \frac{\Pr(B|A)\Pr(A)}{\Pr(B)}$$

The Bayes theorem computes the probability of $A$ happening, given that $B$ has occurred. A simple example would be to ask, what is the probability of a fruit being an apple, considering its colour is red and its shape is round. The assumption is that the features are independent (one feature does not affect any other feature), which is the reason why it is called naïve. An output target class probability is obtained through the productory of all features' probabilities. Although this independence assumption can be problematic (since in most cases, this does not happen), naïve Bayes has surprisingly outperformed other sophisticated classifiers over a large number of datasets, especially where the features are not strongly correlated [24].

For software engineering, naïve Bayes can be applied to handle missing data and mine software process activities software configuration management systems (such as SVN) [25], for software defect prediction (combined with principal component analysis) [26] and, also, presents great accuracy conducting text classification for automatic bug triage [27].

### 2.1.2 Decision Tree

A decision tree is a classifier based on recursive partitions of the instance space. The decision tree consists of a directed tree with a "root" node with no incoming nodes and all other nodes have a single incoming edge. If a node has outgoing edges, it is called internal or test node, while nodes with no outgoing nodes (except for the root) are called leaves, which are also known as terminal or decision nodes. Each internal node splits the instance space into two or more sub-spaces depending on the input attributes' values (typically, each test considers a single attribute, such that the instance space is partitioned according to that attribute's value. For numeric attributes, the condition defines a range value). Each leaf contains a class value which represents the expected target value. The training inputs

are classified by traversing the tree from the root to a leaf, according to the outcome of the tests on the internal nodes along the way [28].

Decision trees have the advantage of being simple to read and explain. A simple example would be to decide whether or not to attribute credit to someone by looking at their income per month and debt information. If the income per month is over 10 000€, a loan is granted, but if it is under 5000€, a loan is not granted. An income per month between 5000€ to 10000€ needs to evaluate the debt value to know if credit should or not be granted. If the debt is over 5000€, credit is not awarded, but if the debt's value is 5000€ or below, credit is awarded.

A decision tree can automatically be obtained from data using a TDIDT (Top Down Induction of Decision Trees) algorithm. This algorithm takes the entire training dataset as input and follows these steps: [29]

- Select an attribute to split on
- Finding a split value by maximizing some purity measure (for instance, the Gini index)
- Sort the instances in the training set into subsets, one for each value of attribute
- Return a tree with one branch for each non-empty subset, where each of these branches has a descendant subtree or a class value produced from applying the algorithm recursively
- Prune back the obtained decision tree to reduce overfitting

Khare and Oak [30] used decision trees for detection of real-time distributed denial of service (DDoS), protecting against data loses and identifying the source of the attack. Mhawish and Gupta [31] to explain predictions of distinguishing and detecting similar structure design patterns in software systems. Lastly, Singh et al. [32] propose a framework to assist on deciding whether to maintain or reengineer a software system.

Decision trees have a natural tendency to overfit. To address this issue and to improve classification accuracy, random forests were developed [33].

### 2.1.3 Random Forest

One way to reduce the high variance of a decision tree's estimate (a model which presents high accuracy for a certain training set, which lowers considerably for another training sets) is to average together many decision tree's estimates. For example, by training $N$ different trees on different subsets of the training data, chosen randomly with replacement. The output target class is obtained through voting of the all decision trees (each vote counting as $\frac{1}{N}$). This technique is called bagging [34]. However, bagging can result in highly correlated predictors (depending on the subsets of data each decision tree uses for training), which limits the possible amount of variance reduction [22]. The technique known as random forests [33] tries to decorrelate the different training subgroups by making decision trees learn from a randomly chosen subset of input variables at each decision node (using the same TDIDT approach mentioned before), as well as a randomly chosen subset of the training set data. These models often have very good predictive accuracy [35], and have been widely used in many applications.

In the software engineering industry, random forests have been used to assess, in real time, a network security situation [36], and to reliably estimate software development effort [37]. Random forests also present great accuracy detecting code smells (poor software system design and/or code implementation, which lowers the quality of the produced source code) [38] and identifying source code authorship with few available samples per author [39].

Random forests (as all methods that use multiple trees) lose their clean interpretability and easy comprehension properties. However, there are various post-processing measures to address this issue, such as a partial dependence plot [40].

### 2.1.4 Neural Networks

Neural networks are information-processing systems, inspired by biological neural networks, designed to recognize patterns by operating in a similar way to the functioning processes of the human brain [41] [42].

Neurons receive an input from predecessor neurons that have an activation threshold, an activation function and an output function. The connections between neurons contain weights and biases which overview on how a neuron transfers output to other neuron. The network computes the input, returns the output and sums the predecessor neurons function with the weight.

A loss function is used to estimate the error between the prediction and the correct results. Starting from the output layer and going backwards, this loss function modifies the weights and thresholds of the variables in the network. This is done according to their contribution for the originally predicted output. This process is a way to adapt the connections of a neural network to the training data, making it able to "understand" the data and its characterizing relations [43].

There are various types of neural networks with multiple different applications. Pang et al. [44] employed a technique based on a deep neural network for predicting vulnerable software components. Amudhavalli et al. [45] combines a neural network with an artificial bee colony algorithm for software effort estimation. For the same goal of software effort estimation, Rijwani and Jani [46] use a multi layered feed forward neural network.

### 2.1.5 $K$-Nearest Neighbors

The $K$-nearest neighbors algorithm classifies a new training input $x$ by finding the nearest $K$ (being $K$ a positive integer previously chosen) neighbors in the training data, by using a distance metric such as, for instance, the Euclidean distance, and assigning the most common output class among these $K$ neighbors to the input object (for classification problems). This method can also be used for regression, where the output is an average of the $K$ nearest neighbors' values for the output target variable.

$K$-nearest neighbors typically obtains high accuracy when using a large training set, but the same doesn't happen with small training sets. It also can't learn if one feature of the objects is more important than another. Feature weights have to be assigned by the developer, consistent with its importance. Also, $K$-nearest neighbors doesn't work well with imbalanced classes (classes where the dominant class has a much bigger representation

than other classes) [17].

$K$-nearest neighbors has been used, by Gupta et. al [47], combined with term frequency-inverse document frequency (TF-IDF) (which measures the importance of a term in a document regarding a documents' collection [48]) to search and select relevant requirements for reuse over a cloud platform. It presents high performance on software vulnerability prediction based on Bellweather analysis [49] and for imputation of missing values on software quality datasets [50]. Finally, Hasanluo and Gharehchopogh [51] propose a method using hybrid particle swarm optimization (uses a defined number of particles to find the best absolute value for an optimization problem [51]) and $K$-nearest neighbors to solve the software cost estimation problem.

### 2.1.6 Logistic Regression

Logistic regression is a generalization of linear regression (used for regression problems) for when the dependent variable is binary (like for example, pass/fail).

In the model representation, and considering logistic regression looks for probabilities between 0 to 1, the relationship between the output and the various features/independent variables is represented by a logistic function [52].

$$P(y^{(i)} = 1) = \frac{1}{1+b^{-(\beta_0+\beta_1 x_1^{(i)}+\ldots+\beta_p x_p^{(i)})}}$$

The best values for $\beta$ coefficients are estimated from the training data using maximum-likelihood estimation. This method seeks to minimize the error in the probabilities predicted by the model according to the provided training data [53].

Logistic regression has multiple applications in the software engineering area. Rahman et. al [54] use logistic regression for error correction on a tool designed for automatic detect transaction errors in SAP systems and propose corrections. Cruz and Ochimizu [55] apply logistic regression models in order to predict fault-prone object oriented classes in software projects. Christiansen et al. [56] predict risks in software development projects through the usage of logistic regression. Lastly, Babu and Reddy [57] propose a logistic regression approach for software reliability analysis, assisting software developers to predict when to conclude testing phases and do software releases.

## 2.2 Unsupervised learning

Unsupervised learning algorithms learn useful properties/patterns of datasets containing many features. It can be used to learn, for example, the entire probability distribution that generated a dataset, either explicitly, as in density estimation, or implicitly, for tasks like synthesis or denoising. Clustering (which consists of dividing the dataset into clusters of similar examples) or association rule mining are also examples of unsupervised learning [17].

Some of the most common unsupervised learning algorithms, recurrently referred in the literature as used in software engineering, will be presented in the next sections, such as:

Over the next sections, we will describe in more detail some of the most common supervised learning algorithms, referred in the literature as used in software engineering applications and studies. These are:

- Apriori
- $K$-means

### 2.2.1 Apriori

The Apriori algorithm was developed by Agrawal and Srikant [58] in order to find all frequent itemsets in a dataset for discovering association rules. It applies an iterative search where k-frequent itemsets are used to find k+1 itemsets.

Apriori reduces the search space (improving its efficiency) by utilizing the Apriori property, which states that all subsets of a frequent itemset must be frequent and if an itemset is infrequent, all its supersets will be infrequent.

In terms of software engineering, Asif and Ahmed [59] applied Apriori in order to find associations between software risk factors and risk mitigation, Al'Zubi et al. [60] proposed a recommender system based on Apriori to improve on projects' requirement gathering process and, lastly, Anand and Dinakaran [61] used Apriori to handle agile requirement prioritization.

### 2.2.2 $K$-means

The $K$-means clustering algorithm's goal is to partition $N$ observations into $K$ clusters, in which each observation belongs to the cluster with the nearest mean. It begins by initializing $K$ points, called means, randomly. Then, it categorizes each item to its closest mean and update the mean's coordinates, which are given by the averages of the items categorized in that mean so far. This process is repeated for the $N$ observations and obtaining, at the end, the final clusters as a result.

Yoon et al. [62] proposed an approach based on $K$-means to detect outliers on software measurement data. Li [63] applied K-means to categorize open source software project developers [64]. Finally, Chen et al. [65] developed a $K$-means based approach for adaptive random testing in object oriented software systems.

Despite all of these presented examples (which can be reviewed in a summarized version on table 2.1), machine learning has had so far surprisingly little impact on software engineers themselves. However, small but possibly significant changes may be emerging on the link between software engineering and machine learning. To speed up this process, it is important to understand how and for what problems can machine learning be applied and if they are relevant for software engineers [66].

| **Software Engineering Tasks** | **Machine Learning Algorithm** |
|---|---|
| Software development effort | Naïve Bayes [25], Random Forest [37] [39], Neural Networks [45] [46] Apriori [60] [61], K-Means [64] |
| Software quality | Naïve Bayes [27], Random Forest [38], K-Nearest Neighbors [50], K-means [62] |
| Software defect prediction | Naïve Bayes [26], Logistic Regression [54] [55], K-means [65] |
| Software reliability | Decision Tree [30], Random Forest [36], Neural Networks [44], K-Nearest Neighbors [49] |
| Software reusability | Decision Tree [31], K-Nearest Neighbors [47] |
| Software maintenance task effort | Decision Tree [32] |
| Software cost | K-Nearest Neighbors [51] |
| Software project risk estimation | Logistic Regression [56], Apriori [59] |
| Software release timing | Logistic Regression [57] |

**Table 2.1:** Software engineering problems and some machine learning algorithms to solve them

Over the next chapter, it is presented an overview on software testing and its link with machine learning.

# Chapter 3

# Data driven software testing

## 3.1 Software Testing

Software testing is a process (or a series of processes), designed to assure that software does what it was designed to do, in a consistent and predictable way [10]. To achieve this purpose, we assume a software system contains errors and define testing as a mechanism of executing it with the intent of finding errors. A successful test case is one that finds a way to make the system fail. The output from this methodology will be a software system with a certain degree of confidence about its correct functioning and performance [10].

Ideally, complete testing (testing every possible permutation of a program) would be the perfect choice. However, for most cases, this is simply not possible. The number of potential inputs for the majority of programs is too large, and might even be literally infinite [67]. Creating test cases for all the possibilities of a complex application would be impractical in terms of time and human resources to be economically feasible.

Because of this, it is impossible to guarantee the absence of all errors. Test-case design becomes of crucial importance, following the natural strategy of trying to make tests as complete as possible (finding the subset of all possible test cases with the highest probability of detecting the most errors), with the constraints of time and cost [10].

Software testing, like software development, follows a life cycle. The software testing life cycle is a part of the software development life cycle and can be defined as a sequence of

activities carried out to perform systematic software testing [67].

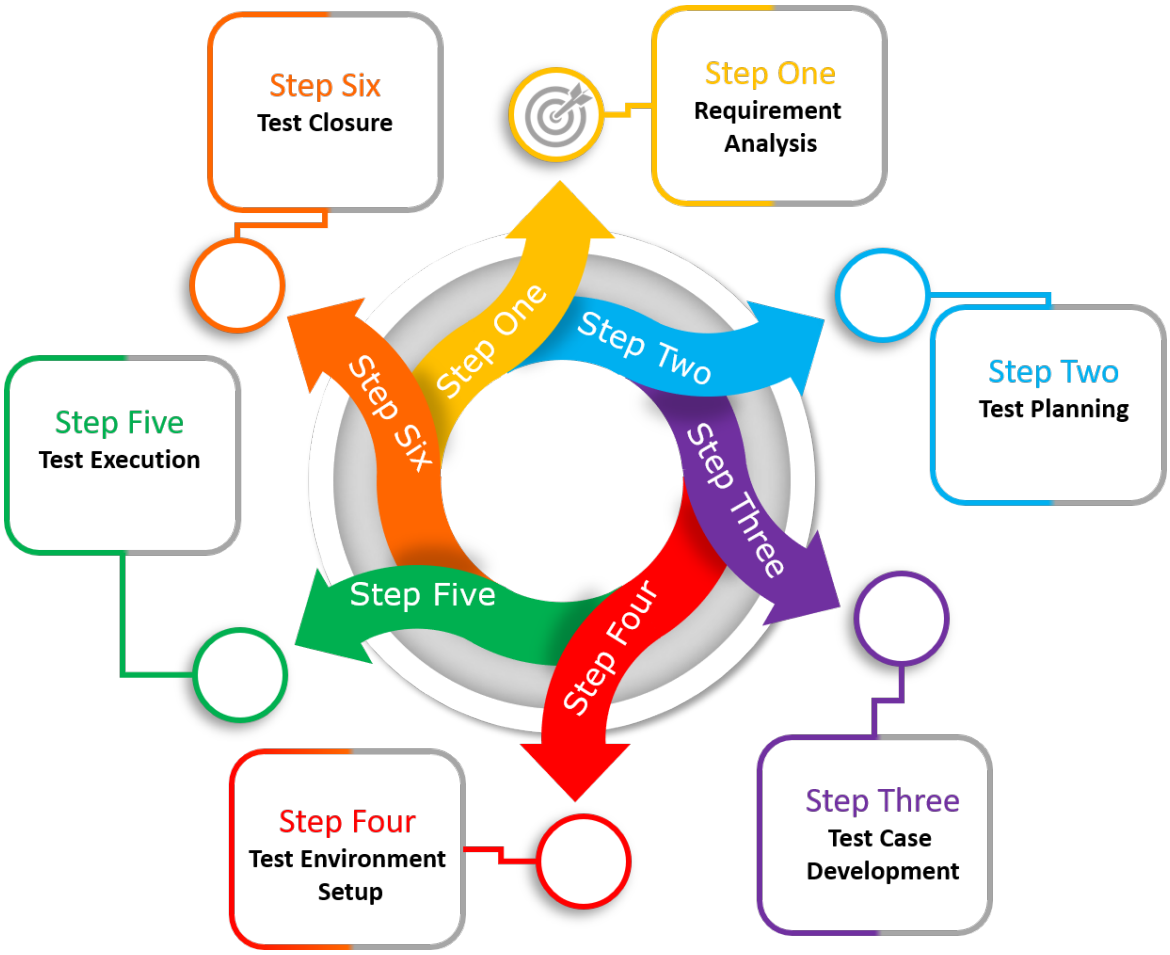



**Figure 3.1:** Phases of the software testing life cycle

In figure 3.1, the different life cycle stages are represented. Firstly, the test team starts by performing requirements' analysis to check its testability, followed by test planning. Test case development involves the creation of test cases according to the test planes and test environment setup proposes the conditions for which the software is tested. Text execution is performed based on the previous stages' results, and after correcting the software according to the failed test cases, test closure finalizes the process, by documenting all these phases [67]. The final software product can then be released with confidence that it will achieve correctly its development purposes.

There are four levels of testing, designed for multiple stages of development: [68]

- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing

**Unit testing** provides independent tests for the smallest testable parts (units) of an application. It focuses on smallest element of the software system called module (like for example, for a calculator application, modules would be addiction, subtraction, division and multiplication).

**Integration testing** exposes defects on interactions between integrated units from different models. It can be done through two approaches- the non-incremental/Big-bang approach, which consists on testing each module independently and then merging with all the other modules and form the program, and the incremental approach, which combines the next module to be tested with the set of previously tested modules, and only then the tests are carried out. This can be done in two ways: top down (it starts with the top, or initial, module in the program, and merges with previously tested modules which have functions calls performed by the top module) and bottom up (begins by testing the modules in the program that do not call other modules, and merges are made with modules that have all of their subordinate modules previously tested).

**System testing** deals the testing of complete system before releasing the software. There are different types of system testing are followed, such usability testing (which is focused on the users' ease to use the application and the ability of the system to fulfill its purposes), stress testing (testing the software under heavy load conditions to verify its robustness and error handling) and regression testing (verify what has been done so far to ensure the changes made over the course of the development process have not caused any new bugs and that no old bugs will appear from later additions of new software modules).

**Acceptance testing** can be done in various ways (such as alpha testing, which simulates the behaviour of real users, or beta testing, where the software is released to a limited number of users to obtain feedback on the application's quality) and provides important quality measures to observe the customer's satisfaction with the product [10].

In figure 3.2, it is possible to observe the various software development stages and its testing processes.

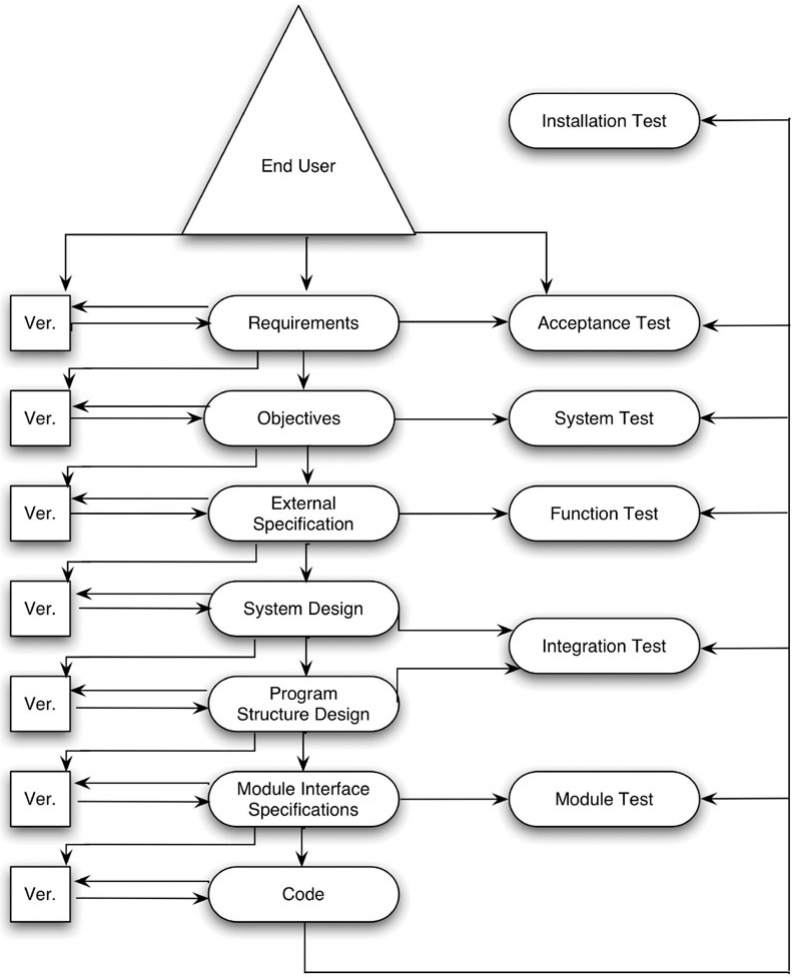



**Figure 3.2:** Software development stages and its testing processes [10]

Since the use case falls into the category of integration testing, it is important to view its two testing methods: black-box testing and white-box testing.

## 3.2 Black-box and White-box Testing

### 3.2.1 Black-box Testing

The black-box (also known as data-driven or input/output-driven testing) testing strategy focuses on finding cases for which the program does not behave as it was expected from its specifications [10]. The program is viewed as a black-box (not taking into any consideration the internal program structures) and tests are derived from the relationship between input and expected output (according to the software requirements).

As seen before, complete testing is impossible, and therefore, so is the approach of exhaustive input testing. This presents two implications: it is impossible to guarantee a program is error free and economics is a fundamental factor in program testing (the goal is to maximize the number of found errors using the smallest number of test cases possible) [10].

In order to achieve this, a well selected test case must:

- Reduce the number of other developed test cases
- Cover a large set of other possible test cases (uncovering the presence or absence of errors over and above this specific set of input values)

From these two stipulations, a black-box methodology known as equivalence partitioning is formed. The second stipulation develops a set of appealing conditions to be tested, while the first one develops a minimal set of test cases which cover these conditions. The test cases designed by equivalence partitioning are found by identifying the equivalence classes and then defining the test cases. Valid and invalid equivalence classes are identified by taking each input condition and partitioning it into different groups (for instance, if there is a specification stating that an input number is bigger or equal than zero, a valid equivalence class would be the input number being bigger than zero, while an invalid equivalence class would be the opposite). The test cases are then designed for covering all existing equivalence classes [10].

Typically, test cases that explore boundary conditions are of more interest than test cases that do not. Boundary conditions are values located on the edges of equivalence classes

(for example, a specification for an input number being that this number is ranged from one to ten. It is more relevant to test for input values one and ten than for the middle values of the equivalence class). Boundary-value analysis diverges from equivalence partitioning in two aspects:

- Boundary-value analysis requires testing on and around each edge of the equivalence class (rather than randomly selecting any element in an equivalence class as a representative)
- Boundary-value analysis derives test cases not only from the input conditions, but also from the output results

A weakness from boundary-value analysis and equivalence partitioning is that they do not explore combinations of input circumstances. Testing input combinations is not a simple task because of the immense number of combinations.

Cause-effect graphing is a method for systematically generating test cases which represent combinations of conditions. A cause-effect graph is a formal language into which a natural-language specification is translated. It helps pointing out incompleteness and/or ambiguities in the specification.

Since cause-effect graphing requires the translation of a specification into a Boolean logic network, it gives additional insight into the specification. Although it produces a set of useful test cases, typically it does not produce all of the useful test cases that might be identified. The steps for build a cause-effect graph are the following:

- Dividing specifications into small pieces (such as verifying if an item is placed in a shopping cart)
- Identifying causes (input conditions) and effects (output conditions). After this step, assigning a unique number for each cause and effect. Causes and effects of a specification are then analyzed and linked, forming a boolean graph (which is the cause-effect graph)
- This graph is annotated with constraints describing impossible combinations of causes and/or effects

- By tracing state conditions in the graph, the graph is converted in a decision table, where each column represents a test case [10]

Equivalence partitioning, boundary-value analysis and cause-effect graphing all have their advantages and disadvantages, which complement each other. Making a combination of these methodologies is a good procedure to obtain the best set of test cases.

### 3.2.2 White-box testing

The white-box (also known as logic-driven testing) testing strategy focuses on exploring and inspecting the internal structure of the software program. It derives test data from examining of the system's logic (and often not taking the specification into consideration).

Similarly to the black-box strategy, exhaustive path testing (testing all the possible paths of control flow through the program) is infeasible. Firstly, because there can be an astronomical number of unique logic paths for a given program. Secondly, even if every path in a program was tested, the program could still contain errors, mainly because exhaustive path testing does not guarantee a program matches its specification [10]. So how should the white-box testing strategy be approached?

Code coverage presents various different criteria to assess a white-box testing strategy. It measures the degree to which a test suite exercises the source code of a software system [69]. A reasonable goal would be to execute every statement in the program at least once. This is known as the statement coverage criterion. This criterion can be useful in finding unused code statements or branches, but may not detect all errors in a program, since it depends on the program's input (like for instance, on "if-then" structures). A way to address this issue is by using the decision/branch coverage criterion. It requires each program's decision to have a true and a false outcome, and for each statement to be executed at least once, invoking each one of all the programs' points of entry.

Decision/branch coverage does not take into account how the variables (or possibly expressions containing boolean operands such as "AND", "OR", "XOR") in the conditional statement are evaluated. This can be done using the condition coverage criterion, which needs enough test cases to verifies if each condition in a decision takes on all possible

outcomes at least once, invoking each point of entry at least once. This criterion does not satisfy the decision criterion (for example, for a decision "IF ($A$ & $B$)", the condition coverage would be fulfilled by a test case of $A$ being true and $B$ being false and another test case where $A$ is false and $B$ is true. This, however, would not trigger the "THEN" code branch to execute).

To address this issue, a criterion called decision/condition coverage was created. To be fulfilled, it needs enough test cases ensuring each condition in a decision takes all possible outcomes at least once and each decision take all possible outcomes at least once, with all points of entry in the program being invoked at least once. A weakness from this criterion is that boolean operators can mask the evaluation of other conditions (such as an expression, for a condition expression using an "AND" operator, all it takes is for one of the condition to be false for the condition result to be false). Multiple-condition coverage fixes this by needing enough test cases so that all possible combinations of condition outcomes in each decision are tested, and all points of entry invoked at least once. [10]

Despite some authors stating that code coverage is not strongly correlated with test suite effectiveness [70], code coverage is a useful metric to identify under-tested parts of a software system. Some other measures, such as DDU [71], can also be used to assess a test suite effectiveness. DDU is an acronym for density diversity uniqueness, and links all of these three metrics. It applies a normalized $\rho$' metric, derived from the $\rho$ metric, which captures the density of a system, the Gini-Simpson index to perceive diversity and a metric proposed by Baudry et. al [72] to identify the number of dynamic basic blocks in a system to estimate uniqueness. These three metrics are combined through multiplication and measuring the probability of a bug being found in that test suite.

Summarizing, although input testing can be seen as superior to path testing since it takes specifications into consideration [10], neither one can ensure complete testing. Combining elements of black-box and white-box testing appears to be the most interesting strategy, since both uncover complementary issues [73]. After covering both black-box and white-box methodologies and their metrics to measure test suite effectiveness and maximize its performance, and taking into account the use case's scope, it is important to focus on how the test cases are generated for black-box testing.

## 3.3 Data driven software test generation

Having referred some techniques of how to maximize the performance of generated test cases, the question now is how to generate these test cases and perform them. There are two approaches: manual or automated. Defining test cases is a very challenging and time consuming task, which, considering software systems' sizes, time constraints and frequent changes in requirements, becomes infeasible. Therefore, the automated approach gains relevance, since it not only reduces the effort cost of testing, but also ensures that test cases properly cover all requirements [5].

There are two main approaches for the automation of generating test cases. The first approach comes from designing test cases from requirements and design specifications. This approach takes advantage of UML artifacts (such as sequence diagrams [74], state and activity diagrams [75] [76]). The second approach develops test cases using code [77]. This is where the power of machine learning comes into play. Taking advantage of the capabilities of machine learning to deal with large volumes of data and its efficiency to identify hidden patterns of knowledge, this can be done through various different techniques and there are also software tools for this task. Some of these techniques include:

- Random/monkey testing
- Info-fuzzy networks
- Markov chain models
- Genetic algorithm

### 3.3.1 Random/Monkey Testing

The simplest form of automated test generation is perhaps to select program inputs at random [78]. This simple and easy to implement concept has the ability to produce a large number of test cases that interact with the system in multiple different ways, many of them being interaction scenarios a typical user is unlikely to perform. Although this is distant from a manual approach, which primarily focuses on the system's core functionalities and

the most common scenarios, randomly generated tests have a high chance to discover new software defects due to their wide diversity and ability to cover numerous possible interaction scenarios [79].

Random testing can also be undirected (with no heuristics to guide its test cases generation) [80] or directed, using for instance, an adaptive strategy to increase the rate of defects' detection from basic random testing [81].

### 3.3.2 Info-fuzzy networks

Info-fuzzy networks (IFN) have a tree-like structure, where components include the root node, a changeable number of hidden layers (one layer for each selected input) and the target (output) layer representing the possible output values. The same input attribute is used across all nodes of a given layer (level), representing conjunctions of values from that input attributes, similarly to internal nodes in standard decision trees, while each target node is associated with a value (class) in the domain of a target attribute, or disjoint intervals in the attribute range if the target attribute is continuous).

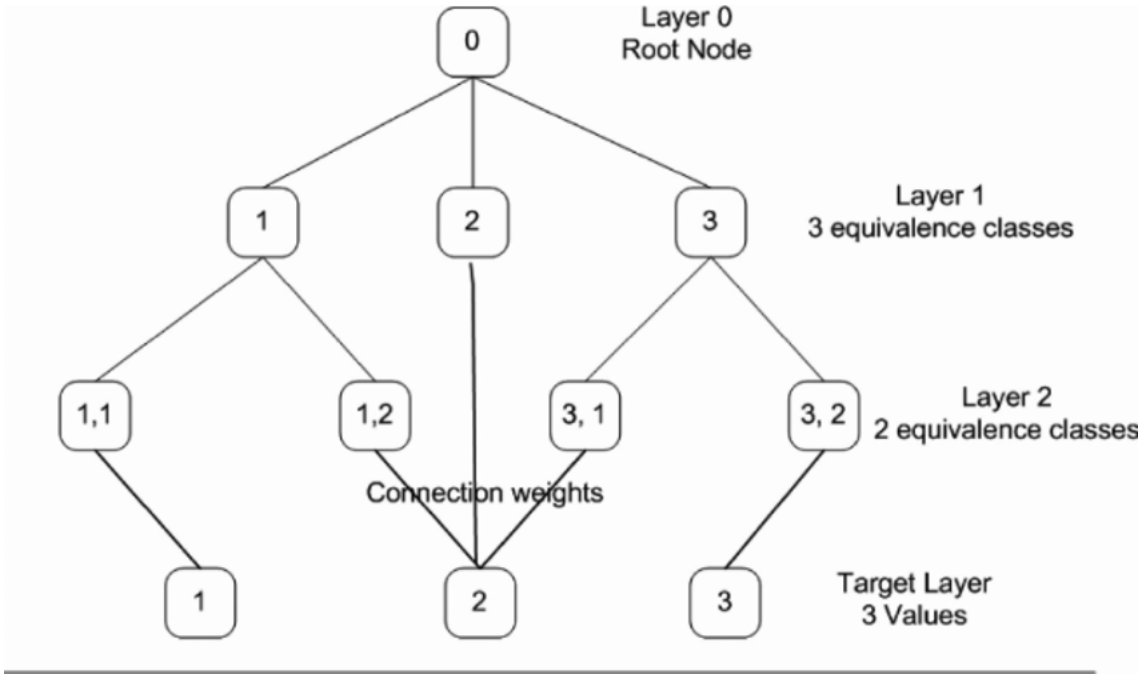



**Figure 3.3:** Info fuzzy network example [11]

Last et al. [4] used info-fuzzy networks to predict output values given test-cases. These test-cases are formed through the usage of a composed execution data from input randomly generated values and the output produced by the legacy system for these input values. The IFN algorithm is repeatedly executed in order to find a subset of input variables relevant to each output and the corresponding set of non-redundant test cases. Actual test cases are then generated according to the automatically detected equivalence partitioned classes.

### 3.3.3 Markov chain models

The Markov property states that the probability distribution of the next state of a process relies only upon the current state. Therefore, a Markov model captures the time-independent probability of being in state $s_1$ at time $t + 1$ knowing that the state at time $t$ was $s_0$. The relative frequency of event transitions during program executions provides a probability estimate for each possible immediate state. It provides flexible and easily understandable representations of the operational profiles of given programs or software systems [82].

In the example in Figure 3.4, the probability of reaching state $A$ from the initial state $A$ is 0.4, 0.2 to reach state $B$ and 0.4 for state $C$. Second order probabilities can also be derived: for instance, given initial state $AA$, the probability of reaching state $A$ is 0 and 0.5 to reach either state $B$ or $C$.

Zhou et al. [83] use a Markov chain usage model based on improved state transition matrix (STM) (a table-based modeling language) for measuring software reliability. Test case generation and adequacy determination is done by using the previously created Markov chain usage model. An improved Kullback discriminant was chosen as the judgment criteria of convergence from the test chain to the usage chain in order to measure if the testing process is sufficient. Also, Siegl et al. [84] use a Markov chain usage model to describe at best all possible usage scenarios of a system under test (SUT) and provide a basis to systematically derive test cases without human interaction. This approach, applied in a software system coming from the automotive domain, provides good indicators before and after the test cases are executed and it also creates test cases by random sampling the Markov chain usage model that would not have been tested otherwise. Markov chain models will be a key aspect of the use case's framework to generate test cases.

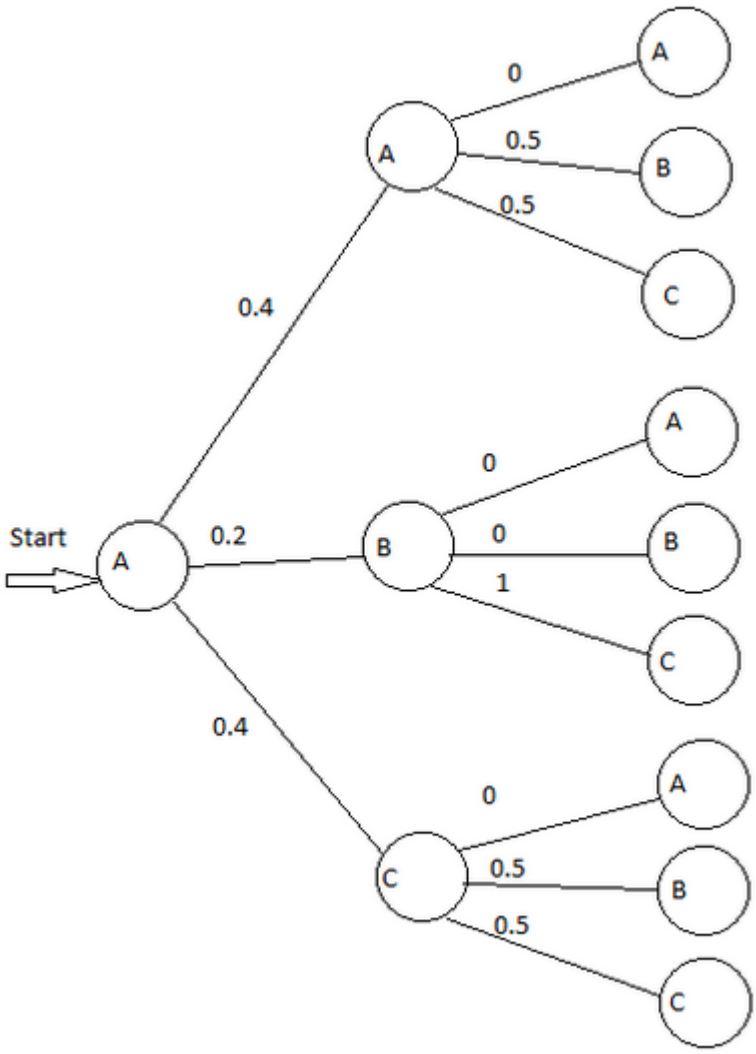



**Figure 3.4:** Markov chain model example

### 3.3.4 Genetic Algorithm

Genetic algorithm is an optimization technique which can be applied to various problems. It uses a survival of the fittest technique, where the best solution survives. The two main requirements of a genetic algorithm are: an encoding used to represent a solution from the solutions' space and an objective fitness function which measures how good a solution is [77]. There are three different types of operators used in the process of a genetic algorithm: selection, mutation and crossover.

The selection operator chooses the best solutions based on the fitness value of each solution, obtained from the fitness function. After selection, crossover combines two different

solutions to generate a new test case. Mutation changes the fitness function to be applied on the solutions to generate new test cases [85].

Khan and Amjad [86] combine a genetic algorithm approach with the $K$-means clustering algorithm to divide a set of randomly generated test cases (considered as effective by the genetic algorithm) and divide them in different clusters. Each cluster finds a group of test cases with high path coverage and then applies the genetic algorithm to produce new test cases in each cluster for increasing the path coverage. The genetic algorithm was also combined with naïve Bayes by Arwan and Rusdianto [87]. The naïve Bayes determines the number of iteration levels for independent path determination which can be generated by the genetic algorithm for basis path testing (verifying and obtaining paths that must be skipped/tested at least once to ensure an error free code) [87]. Decision trees [88] [89] and neural networks [90] [91] also present valuable assets in the generation and automation of test cases.

After analysing the definition of software testing, its life cycle, types and processes, we can see the gains from using machine learning in software testing. Machine learning, however, demands large quantities of test data to increase its effectiveness. A way to obtain large amounts of test data without spending too much time would be gathering data from user sessions. In the next chapter, user sessions will be analysed in terms of how to obtain them, process them and their relationship with graphical user interface (GUI) testing.

# Chapter 4

# Automated testing from user sessions

## 4.1 Data Driven GUI Software Testing

A convenient way to capture user sessions and their inputs is through to the graphical user interface (GUI). GUIs make applications easier to use, increasing the software's productivity accessibility.

GUI software testing is the process of testing a software system containing a graphical front-end interface to ensure that it meets its specification [92]. GUI testing falls into two different categories: usability testing and functional testing. Usability testing assesses the quality of a graphical interface in terms of user interface design, while functional testing evaluates if the graphical user interface does what the software application is intended to [67].

Functional manual GUI software testing performed by developers or testers is a very time consuming task and it is often error prone, considering most test scenarios are typically left out (for example, consider a form which contains n interface controls, it is required a factorial number (n!) of test cases to cover all possible combinations [93]) [94]. Therefore, automated functional GUI software testing methodologies prove to be more efficient, reliable and cost effective.

Functional testing can be divided into four types: GUI system testing (performing system testing through the GUI), GUI regression testing (testing the GUI to verify if changes in the software have not caused any new bugs), input validation testing (evaluates how well the

software recognizes and behaves to invalid inputs) and finally, GUI testing (assessing how well the GUI works. For example, checking if all the GUI controls work as intended and if the software allow the user to navigate to all permitted screens while blocking inappropriate navigations disallowed) [67].

There are various approaches to functional GUI testing, such as:

- Unit Testing
- Random/Monkey Testing
- Capture/Replay Testing
- Model-based GUI Testing

### 4.1.1 Unit Testing

As it was seen on the previous chapter, unit testing provides independent tests for the smallest testable parts (units) of an application, which in this case are individual GUI controls (such as buttons, text boxes, check boxes) of a software's GUI.

Test adapters can be easily implemented at the unit test level, capturing the commands performed on the system's GUI. Ramler et al. [95] highlight and observe their usage, concluding that they provide results for all levels of testing, but the complexity and corresponding effort rise when it comes to implement test adapters at further testing stages.

The GUI unit testing task for Java applications can be simplified by the use of the Abbot framework[1]. This framework presents a Java test library which has been implemented with methods to reproduce user actions and examine the state of GUI components through test scripts (which need to be written by the tester). It also provides an interface to use a script to control the event playback in order to enhance integration and functional testing [96].

[1] `http://abbot.sourceforge.net`

### 4.1.2 Random/Monkey Testing

As it was also seen on the previous section, random/monkey testing examines the system's functioning by generating and using program inputs randomly. For GUI testing, these can be, for instance, keyboard events (key pressed, for example), mouse events (click, double click), filling a text box, tick a check box. One of the challenges presented by testing a GUI using random/monkey testing is guaranteeing the elements meet specific condition to be usable for interaction (for instance, verifying if an element is enabled and visible on the screen). Only after determining which elements are actually usable on a certain moment, random selection and randomly choosing an interaction from all of the possible interactions can take place on an element.

Random testing can also be undirected (with no heuristics to guide its test cases generation) [80] or directed, using for instance, an adaptive strategy to increase the rate of defects' detection from basic random testing [81].

Wetzlmaier et. al [97] built a framework for random/monkey testing. This framework initializes the system under testing (SUT), explores the GUI (firstly, it enumerates all windows that belong to the SUT's processes , determines the top-most window and starting from this window, it traverses the GUI's hierarchy of elements and extracts information from them, such as their name, type, size and location on the screen) and performs random interactions. Whenever a fault is detected, the framework verifies if it is a fatal failure (stopping the SUT and verifying what triggered this error) or not (taking note of the fault, but continuing the testing process until it is complete). There are also available multiple open source and commercial tools for random/monkey testing, such as JCrasher [98], Jartege [99], Randoop [100].

### 4.1.3 Capture/Replay Testing

Capture-replay is a broadly accepted method for regression GUI testing. [96] The concept is to capture some user inputs, replay them through the GUI after code changes have been made and monitor the differences. There are capture-replay tools available on the market,

such as Jacareto[2], Pounder[3] or Marathon[4]. Despite being used and broadly distributed, this method presents various issues.

Firstly, the capture procedure requires testers to perform manually multiple and intensive interactions with the GUI through mouse events and keystrokes. Both writing the test cases and recording them by hand is an error prone task, very time consuming and with an associated high effort. The automated capture/replay tool records the interactions in the form of test scripts for later usage. These test scripts are often hard-coded, which often leads testers to edit and debug them (for example, for different test inputs), which also requires time and effort.

Generating all the possible test cases for all of the GUI components through capture/replay testing is unfeasible, especially considering that the manual GUI interactions often contain mistakes, such as wrong clicks or keystrokes. And even if the execution is successful, the captured results don't verify if the software's business functions (often non-GUI modules) were triggered correctly by the GUI events. Changes on a GUI's component location can also render useless the previously recorded test scripts.

In conclusion, although capture/replay tools are easy to find and use, they are prone to be inefficient and ineffective [96].

### 4.1.4 Model-based GUI Testing

Model-based GUI testing is built upon generating test cases automatically from a graph-based model of possible user interactions with the GUI. Each node represents a particular event which can be triggered by interacting with a GUI's widget. Edges between nodes link two events that can be executed consecutively. Test cases are generated by selecting an event that can be triggered without any preconditions and traversing a path of the model. A test oracle is necessary to indicate if the sequence of events is valid or not [101]. Works such as the TOM framework [102] and tools such as TCG [103], GUITAR [104] and Spec Explore [105] are examples of model-based GUI automation tools.

[2] https://sourceforge.net/projects/jacareto/

[3] https://sourceforge.net/projects/pounder/

[4] https://sourceforge.net/projects/marathonman/

These graph-based models can be generated automatically by the applying reverse engineering techniques or manually by the testers, typically by deriving the GUI models from the requirements specification. The model's quality is crucial for detecting errors. A good model is able to test large parts of a system using a minimal amount of the possible user interactions, while a poor model may generate a large quantity of test cases that assess repeatedly a fragment of the system, without ever investigating the functioning of other fragments.

As it was seen in the previous chapter, complete testing is unfeasible [67]. This means that to efficiently test a system, it is preferable to use techniques that identify important sequences of events which cover the most relevant user interactions and sequences of events that lead to program errors [101]. Over the next section, sequence pattern mining will be defined and some of its most important methodologies will be presented.

## 4.2 Sequence Pattern Mining

Sequential pattern mining is an important sub domain of data mining. Data mining focuses on extracting useful patterns on large amounts of data [106], while sequential pattern mining aims to extract these patterns in the form of totally ordered or partially ordered subsequences, in terms of time [107]. Some problems that can be approached using sequential pattern mining analysis are:

- Having a set of alerts and state conditions issued by a monitored system before an error, are there any sequences or subsequences that may help predicting an error before it occurs?
- Can a user's behaviour be identified as suspicious by analysing the sequence of its input commands?
- Can users who make purchases on a certain website be characterized in terms of the sequence of browsed webpages within the site?

Frequent sequence patterns are typically found taking into consideration a minimal support

parameter [108]. This parameter is either defined as the number of sequences that contain a certain subsequence (absolute support), or a ratio between the number of sequences containing a certain subsequence divided by the total number of sequences in the database (relative support) [108]. Only patterns which exceed this predefined threshold are considered relevant. This value should be high enough to prune patterns of no interest, but low enough to keep all of the relevant sequential patterns existent in the data [109]. In order to generate them, there are multiple frequent sequence mining algorithms. These algorithms can be categorized by the database format (horizontal or vertical) and their search approach (breadth-first search or depth-first search) [107].

Horizontal formatting organizes the database by object id and then by transaction timestamp, containing the sequences. Time is therefore represented by the transaction timestamp and the order of elements in a sequence. For example, in the figure below, the object with an id of 1 contains the transaction sequences ($a$,$c$) at time $t1$ and ($e$,$h$,$w$) at time $t2$. By comparison, the vertical format keeps, for each label, the set of all object ids where the label appears and its corresponding timestamps (for instance, label "a" is present on the objects with id 1 at time $t1$, 3 at time $t3$, 4 at time $t4$, and 5 at time $t2$).

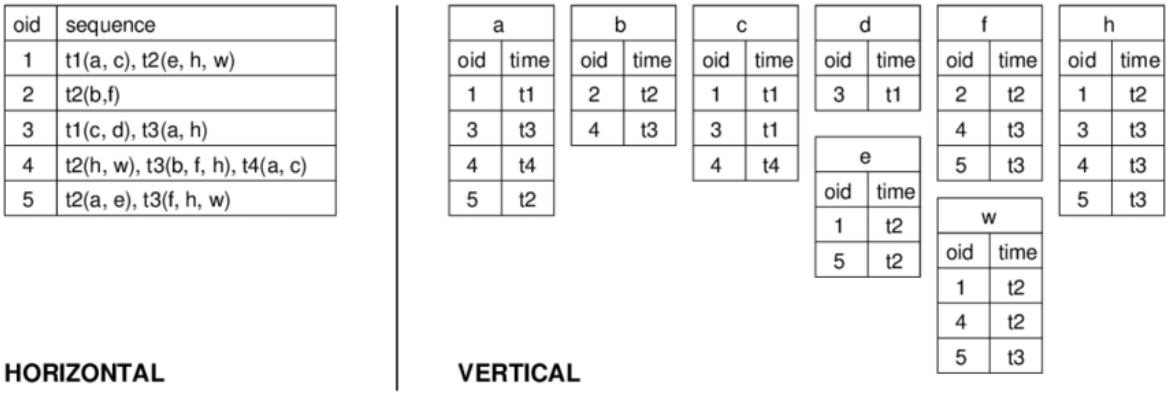



**Figure 4.1:** Database formats for sequence pattern mining [12]

### 4.2.1 Horizontal approaches

In horizontal formatted databases, the sequential pattern mining process is carried out using a breadth-first search. Apriori-based algorithms (such as AprioriAll, AprioriSome,

DynamicSome) [110] are considered to be breadth-first search since all $k$-sequences are established together in each $k$-th iteration of the algorithm, based on the previous iteration. These algorithms use a 5-stage process:

- Sorting the transactional database according to the object id
- Obtaining the 1-itemsets, from the sorted database, with support over the predefined threshold
- Transforming the sequences into the found itemsets they contain
- Generating all frequent sequential patterns from this transformed database
- Pruning sequential patterns that are contained in other super sequential patterns

Despite proving to be inefficient, the Apriori algorithm is the basis of many later on developed efficient algorithms for sequential pattern mining. Some of its limitations are that some observed patterns were of no real value, in terms of being too widespread time events, and, for datasets containing an associated hierarchy, it might be relevant to find patterns on one level or multiple levels of the hierarchy. Also, there are many domains where transactions occurring in between a predefined time window can be seen as a single transaction [107].

To fix all of these issues, Agrawal and Srikant [111], using Apriori as a basis, developed the GSP (Generalised Sequential Patterns) algorithm. Despite adding bounds between time events, a sliding time window to define a single transaction and the possibility of handling different hierarchies defined by the user, GSP and later on developed and improved versions of it (such as PSP [112] or MFS [113]) have weaknesses in terms of generating a huge set of candidate sequences, which require multiple database scans, and the high number of short length patterns, which makes these algorithms inefficient for mining long sequential patterns [109].

The frequent pattern growth paradigm rose to address these problems. This model erases the need for the candidate generation and pruning steps from Apriori type algorithms by compressing the database that represents the frequent sequences into a frequent pattern tree and then dividing this tree into a set of projected databases. The projected databases obtained from this divide-and-conquer methodology are then mined separately [114].

Following this framework, Pei et al. [115] proposed and developed the FreeSpan algorithm. It follows the previous idea of using the frequent sequences to recursively create a set of smaller projected databases for parallelized mining. FreeSpan has two different methods for the creation of these projected databases: Level-by-level projection or Alternative-level projection. Any of these methods split the data and the set of tested frequent patterns on each projected database. For any possible execution, FreeSpan only makes three scans on the original database. Despite the significant cost of operating the projected databases, FreeSpan is efficient and is considerably faster than GSP [109].

Afterwards, Pei et. al. presented an improved version on FreeSpan. PrefixSpan [116], like its' predecessor, is a form projection based algorithm. The main concept is to check only the prefix subsequences and project only their postfix subsequences into the projected databases, instead of projecting the sequence database. This approach ensures that only candidate sequences existing in a projected database are generated or tested. Longer sequential patterns are built from shorter frequent ones, which reduces the search space. Like FreeSpan, the biggest computational cost comes from operating the projected databases. This cost is lowered by using two optimizations for reducing the size, number and memory cost of projected databases. PrefixSpan presents the same results as GSP and FreeSpan, running significantly faster than both algorithms [107] [116].

### 4.2.2 Vertical approaches

Vertical database format algorithms generally perform better than algorithms that use an horizontal format database. The vertical format has the benefits of narrowing down the data to only the relevant, generating patterns and calculating their supports without having to execute more database scans, sparing high computational costs and improving the performance of vertical formatted algorithms on larger databases over horizontal ones [117].

These changes on the data layout caused the appearance of depth-first search algorithms. Constraints also started becoming a part of the mining process to both reduce the processing time and the number of results [118]. The Sequential PAttern Discovery using Equivalence classes algorithm, most commonly known as SPADE, and cSPADE, its variant containing constraints, were developed by Zaki [119]. It uses a vertical database format

and consists on decomposing the search space into sub-lattices according to the dataset's equivalence classes, which can be processed independently in main memory. This approach mostly requires three database scans, but it can also need only a single scan on some pre-processed databases. Either a breadth-first search or a depth-first search strategy is employed To find the sequential frequent patterns in the sublattices. Experiments show that SPADE is about twice as fast as GSP [119].

Ayres et. al. [120] author another depth-first traversal algorithm, called Sequential PAttern Mining, known as SPAM. It uses a vertical bitmap compression, with efficient support counting and candidate generation. Each bitmap contains a bit for each transaction in the dataset. If an item is present in a transaction, the bit is set to 1, otherwise it is set to 0. This bitmap representation requires a lot of memory, which makes it efficient for databases with long sequential patterns. Unlike other algorithms, which output all patterns of length one, then all patterns of length two, and so forth, SPAM outputs sequential patterns of different length in an incremental, online way. The results show a trade off between space and time: SPAM is more efficient compared to SPADE and PrefixSpan on large databases, but it consumes more space [120].

Despite the advantages of vertical algorithms, a performance bottleneck of these algorithms is that they use a generate-candidate-and-test approach. To tackle this problem, Fournier-Viger et al. [117] proposed a data structure called CMAP (Co-occurrences Map, CMAP). This structure is capable of keeping a co-occurrences map of items extracted from a single database scan and provides a new approach to the sequence pruning stage based on the co-occurrences' properties. This generic optimization gave origin to improved versions of vertical algorithms, such as CM-SPADE and CM-SPAM [117].

After introducing the motivations and objectives over the introduction section, bonding the importance of machine learning to the software industry, and more specifically to the software testing area, studying both the GUI testing and sequence pattern mining fields, it is now time to view the use case. The framework's organization, the tool's used methods and performance evaluation metrics will be presented in the course of the next chapter.

# Chapter 5

# The use case

Recapping this work's motivations and objectives, Anywhere+ is an insurance ERP developed by RandTech Computing (`https://rtcom.pt`) with the intent of simplifying the organization of information of an insurance company. Since a manual testing approach became impractical with the expansion of clients and functionalities of the system, a tool for generating software test cases from user interaction data was developed. The tool, conceived in a mix between capture and replay testing and model based testing, is part of a framework designed for automated testing purposes.

## 5.1 Framework structure

Randtech's update and test environment framework was composed in order to automate the complex software update process of the Anywhere+ solution. This includes code, databases and tests, and enables safer, more frequent and faster updates. It follows a modular structure, and is flexible enough to be easily adapted to other deployment flows. Figure 5.1 presented below shows its scheme.

In the installer module, four smaller modules can be found. The XHTML generator is capable of recognizing a set of patterns present on the user interface design, and it is responsible for identifying a set of embedded business concepts in the system. The XHTML files which compose the application's UI will automatically be transformed, for both users

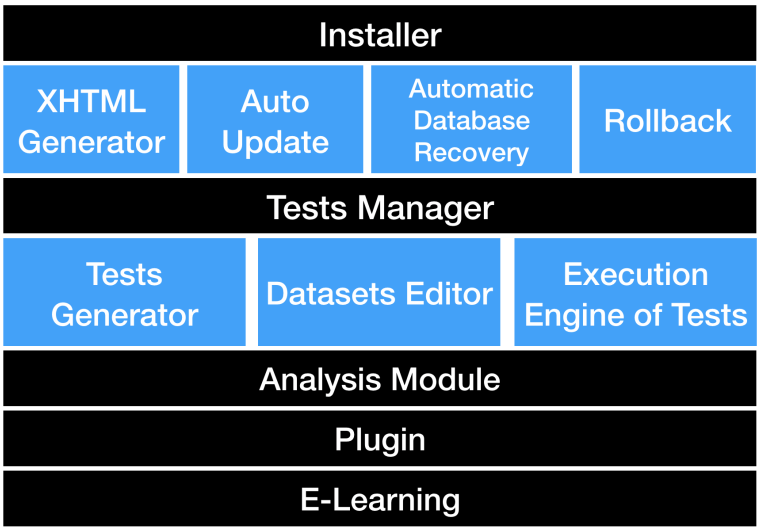



**Figure 5.1:** Software Deployment Framework

and developers, at the time of installation of any version of the application. These previously unraveled business concepts serve as a base for the plugin's purpose. Able to recognize the XHTML generator's business concepts, the plugin, which was developed for browsers, captures and registers, in a file, the various user actions in the application. Through the continuous usage of the application by its various users, thousands of records will be generated, building the various test case scenarios which will be recognized and treated by the analysis module. This analysis module, discussed in this work, uses a frequent sequences' pattern mining algorithm to discover recurrent sequential patterns in the data, and builds a Markov model, which automatically produces test sequences. These tests are used to identify bugs in the current release and in preparation for the next deployment release, as long as the structure of the GUI is not changed.

The application's development is done according to the principles of continuous integration (all of the developers' working copies are combined using a shared repository [121]). Therefore, after triggering the build, the XHTML generator will be invoked to generate the business concepts, which will be packed on a deploy file, together with the application's installation JAR and the recovery scripts of the database. The local AutoUpdate service will proceed to install the deploy file, automatically recovering the database (done by the automatic database recovery routine, based on the recovery database scripts). It also invokes the execution engine of tests, which will execute the several tests' batteries generated based

on the previous defined test models derived from the tests' generator.

The tests' generator creates test batteries according to the datasets existent in the datasets' editor and a series of criteria (like, for example, executing the test steps with different users/permissions). The execution engine of tests triggers a set of actions on the browser, identifying business concepts, taking screenshots of the browser window, as well as execution times. This process is also useful for the e-learning component, which is able to provide visual tips to guide the user through the various actions of the application. In case there's a previous test execution, this one will be used as a baseline to perceive disparities and the test might fail depending on the selected criteria. In this case, the tester may define if this current execution is the right one, designating this as the new baseline, for which the test will be considered as success. The execution engine of tests also allows executing the tests in parallel, as long as there isn't any dependency in between tests.

In the case of failure, the AutoUpdate will invoke the rollback model, leaving the application and the database in its previous state. In case of success, the installer will notify the remote AutoUpdates of the various clients/environments that there is a new available version of the application to be installed. By accepting an AutoUpdate, a similar process to installation will be triggered. This process can be used by any client environment, independently of their operating systems or equipment, by just a small number of clicks by the user.

In the next section, the process of generating and recognizing business concepts, as well as the capturing and formatting of data by the plugin, will be approached.

## 5.2 Data acquisition

In order to define and execute the software tests designed to feed the analysis module, it is fundamental that the framework can recognize the various business concepts. To accomplish this, a syntax for the XHTML generator has been defined. This syntax is capable of recognizing a set of patterns used on the UI design of the Anywhere+ application.

### 5.2.1 Data Format

The XHTML files which compose the Anywhere+ application are transformed by inserting the reference to the business concepts. Despite its complexity, this transformation occurs transparently and automatically, both for programmers as for users, and is triggered for each build of the application.

An example of this transformation, applied by the XHTML generator to a snippet of the login XHTML file which contains the username section of the file, is presented in figure 5.2.

```
1  <h:inputText
2    id="TxtUserName"
3    onkeypress="return go(event);"
4    required="true"
5    requiredMessage="#{MSG['login.mand.mail']}"
6    validatorMessage="#{MSG['login.invalid.email']}"
7    value="#{BUSER.username}">
8  </h:inputText>
```

**Figure 5.2:** Snippet of the Login.xhtml file of the application

The <h:inputText> represents a component which will be transformed into an input field to be rendered by the browser. It contains error handles and the value of the text field (value attribute). These attributes use expression language (EL), a popular scripting language on Java™ frameworks with the format #{expression} or :{expression} and the JavaServer Faces engine (JSF) will evaluate each expression when the page is rendered for the browser.

There are two types of dominant expressions on the application's UI pages: references to JavaBeans in the format nameBean.nameField (as it can be seen from this example, the BUSER.username field) and references labels/texts of a dictionary (such as the labels 'login.mand.mail' and 'login.invalid.email')

### 5.2.2 XHTML Transformation

To ensure the highest possible number of concepts, the XHTML generator will consider as business concepts any code snippet of expression language found on the XHTML file. Here lies an essential difference concerning other approaches: the concepts are identified solely by themselves. Since they are not identified by their HTML element path (for instance, by using XPATH), the plugin is immune to changes on a component's location from the GUI.

Considering that the business concepts are EL expressions, it is necessary to prevent the transformation of the EL expressions when they are simply EL expressions and nothing more. In order to achieve this, it has been defined that non-concept expressions begin with ':{' instead of beginning with '#{'.

To make sure business concepts are not lost along the many transformations occur from the initial XHTML to the final HTML rendered by the browser, we aggregated the concepts in an HTML tag. The concept is now represented by an association between concept name and concept value. On figure 5.3, it is possible to see the output from the XHTML generator applied to the previous snippet of the login XHTML file, involving the username segment.

After the final transformation to HTML performed by the JSF engine, the final HTML snippet is obtained, visible in figure 5.4. The page contains a form button to press after introducing a username on the business concept "rtcconcept_value3_login". If no username is introduced, a "Username is required" message is triggered and a record of rtcconcept_requiredMessage1_login=":MSG['login.mand.mail']" is saved by the plugin. A similar thing happens if an invalid username is introduced. In the case of inserting a valid username, the plugin records an "rtcconcept_value3_login="#BUSER.username" action, with "BUSER.username" being the inserted username.

### 5.2.3 Logging the GUI events

After understanding the XHTML generation and recognition of business concepts processes', it is also relevant to know how the plugin captures and formats the data originated from the user interactions. After the final transformation, it is important to observe which concepts

```
<em requiredMessage1_login="#{MSG['login.mand.mail']}"
    rtcconcept_requiredMessage1_login=":{MSG['login.mand.mail']}"

  validatorMessage2_login="#{MSG['login.invalid.email']}"
  rtcconcept_validatorMessage2_login=":{MSG['login.invalid.email']}"

  value3_login="#{BUSER.username}"
  rtcconcept_value3_login="#{BUSER.username}">

  <h:inputText
    id="TxtUserName"
    onkeypress="return go(event);"
    required="true"
    requiredMessage="#{MSG['login.mand.mail']}"
    validatorMessage="#{MSG['login.invalid.email']}"
    value="#{BUSER.username}">
  </h:inputText>

</em>
```

**Figure 5.3:** Output from XHTML generator to allow plugin recognition

were apprehended and recognized by the application's plugin. For this purpose, changes on the page elements' style are made whenever the plugin is active. There are two yellow and two red colour styles which are used around the displayed elements: yellow styled elements signal correctly recognized concepts while red styled elements display concepts which were not apprehended. The absence of colour is also an indicator, since it shows that the element did not contain any concepts or that they were not being noticed by the plugin.

As previously mentioned, the plugin component is responsible for capturing the various user actions in the application (such as tracking mouse clicks and keyboard events), recording them on a text file. These records follow a simple and optimized structure for the analysis task: firstly, it contains the *timestamp*, *session id*, *tab id* and the *business concept* separated by a comma. Secondly, there are three components separated by semicolon: *action/com-*

```
<em requiredMessage1_login="Username is required"
    rtcconcept_requiredMessage1_login=":{MSG['login.mand.mail']}"

  validatorMessage2_login="Invalid username or password"
  rtcconcept_validatorMessage2_login=":{MSG['login.invalid.email']"

  value3_login=""
  rtcconcept_value3_login="#{BUSER.username}">

    <input type="text" class="form-control" required="required"
          onkeypress="return go(event);" autofocus="">

</em>
```

**Figure 5.4:** Final transformed HTML for plugin recognition

*mand*, *target* and *value*. The action field is mandatory and tracks the action performed (for instance, clicking on a link). Target is the business concept used by the action (a dropdown menu, for example) and the value is what will be used on the action (such as text inserted on a textfield).

The action field is mandatory, but target and value parameters may or not be void depending on the captured action. Each interaction is registered on a line of the text file and they serve as an input for the analysis module. In Figure 5.5, an example of a recorded interaction can be seen. It features a *clickLink* action on the concept **rtcconcept_rtctextvalue*= : {MSG[OPTION.translationCode]}: arearowid25 :optionrowid119* with a value of *"Tomadores"*.

After overviewing the process of generation and recognizing business concepts, followed by the plugin's operating mode, the functioning of the analysis module will be addressed in the subsequent section.

```
[ts=2019-01-21T09:59:01.010Z,
sid=S_-2en-FjlByP2_bdWJ0k_zU_bV88TQUdhj6-x2h.rtcva,
tid=11,
attr=pt:rtcconcept_rtctextvalue33_menuw8,
n=1,
count=1]

[clickLink;*rtcconcept_rtctextvalue*=
:{MSG[OPTION.translationCode]}
:arearowid25:optionrowid119=Tomadores;]
```

**Figure 5.5:** Captured GUI interaction record

## 5.3 Analysis Module

### 5.3.1 Discovering frequent sub sequences

The analysis module takes the captured interactions by the plugin component and builds a Markov model. This Markov model is composed by states, which are user interactions, and their connections represent transitions between the user interactions.

The first step is to pre-process the interaction events from the text file and look for frequent sequences with a given maximum length. Based on last chapter 4's studies over sequence pattern mining, the chosen algorithm for this task was CM-SPAM [117]. As it was previously observed, CM-SPAM is a vertical optimized algorithm, based on SPAM, and presents the fastest processing speed compared with multiple algorithms, with high effectiveness over the found results.

The CM-SPAM algorithm is contained in the Java™ SPMF library. In order to obtain the frequent sequences of events, the analysis module invokes this library's method with a defined minimum support threshold of 1.

### 5.3.2 Building the Markov Model

After applying the CM-SPAM algorithm, the resulting sequences are chained into a single Markov model. As previously stated over chapter 3, the Markov property states that the probability distribution of future states of a process relies only upon the current state, being the probability estimate for each possible immediate state given by the relative frequency of event transitions during program executions [82]. In the analysis module, the Markov model is represented as a dictionary structure, $< key, values >$, where $key$ is the current initial state (associated with the previous actions) and $values$ is a list of future actions associated to their probabilities. The states in the application are the actions performed in the different concepts.

This model is built by uncovering all the initial states from the sequences and then, for each initial state, exploring the next states. In order to prevent impossible test cases to be generated, each test case sequence has an initial state which is selected from all possible initial states of the events' sequences. The transition probabilities are measured according to the number of transitions from a determined current state to a next state dividing by the total number of transitions originated from the same current state. The model is then output in a JSON format file, and an example can be seen on Figure 5.6. The figure shows that after reaching the state of "click+*rtcconcept_label*:MSG['generic.policies']", we have three possible future states: two of them with 0.4 probability of being chosen as the next state, and the remaining one with a probability of 0.2.

It is also important to refer that, when used to reproduce sequences, Markov models can lead to infinite cycles of alternating states. To address this issue, an end-of-sequence token was adopted, always adding the possibility of exiting an infinite cycle.

### 5.3.3 Using the Markov Model to generate tests

Given the Markov model, tests are generated in multiple ways using a proportional sampling criterion. This approach, proposed by Zhou et al. [83], divides the choice interval space, making it between [0,1] and splitting by the occurrence probabilities of each action (for example, considering a model where the current state has three possible choice, $A$, $B$ and

```
{
  " ": [
    {
  "Target": "click+*rtcconcept_label*:{MSG['generic.policies']}",
  "Probability": 1
    }
  ],
  "click+*rtcconcept_label*:{MSG['generic.policies']}": [
    {
  "Target": "click+*rtcconcept_label*:{MSG['generic.policies']}",
  "Probability": 0.4
    },
    {
  "Target": "click+*rtcconcept_rtctextvalue*=:{MSG['menu.policies']}
     =Apolices",
  "Probability": 0.2
    },
    {
  "Target": "click+*rtcconcept_titlesearch*=:{MSG['claim.searchPolicy']}"
     ,
  "Probability": 0.4
    }
  ]
```

**Figure 5.6:** JSON file representing a set of possible outcomes and their probabilities

$C$, being the probabilities for a given state to be chosen of $A$ = 0.3, $B$ = 0.2 and $C$ = 0.5. The interval will be split into intervals of [0,0.3], ]0.3,0.5] and ]0.5,1]). A random number between 0 and 1 is generated and the action is chosen according to where the generated value fits (for example if the generated value is 0.4, the chosen action will be $B$).

The generated tests correspond to interaction paths that could be followed by the platform's users. Figure 5.7 shows the sequence generator method, based on the proportional sampling adopted criterion. This algorithm takes three parameters, *N*, *L* and *markov*. *N* represents the order of the Markov model, i.e., the number of actions to be taken into account for the next action of the Markov chain. *L* stands for the number of sequences to

be generated, and *markov* is the Markov model generated by the Analysis module.

```
1  input: N, L (number of tests), markov (model)
2  output: sequences (set)
3  begin
4     initial_States = get_initial_states(markov)
5     sequences = {}
6     for 1 to L:
7       s = {}
8       current_state = Sample(initial_States)
9       s.add(current_state)
10      while !s.contains(EoS) and length(s) =< N  :
11        n_States = get_next_states(markov[current_state])
12            if (length(n_States == 0)
13                break
14            else
15                next_state = Sample(markov[n_States])
16                s.add(next_state)
17                current_state = next_state
18            end if
19      end while
20      sequences.add(s)
21    end for
22    return sequences
23 end
```

**Figure 5.7:** Generating Sequences Method

Lastly, after understanding how the analysis module functions, its deployment is presented. This topic will be approached in the last section of this chapter.

## 5.4 Deployment

The analysis module was packaged under Docker. Docker is an open source platform designed to facilitate the process of developing, distributing and running applications. The applications are packaged with all the supporting dependencies into a standard form called a container. These containers are executed and kept running continuously in an isolated way on top of the operating system's kernel. [122] This makes Docker containers independent of the machine's operating systems.

To create a containerized version of the analysis module, two files are necessary: a Dockerfile and a Docker-compose file. The Dockerfile contains the parameters for building the container using a Rocker distribution. Rocker is a Docker environment designed to run R-platform built Docker images (an instance of a given program which is executed on a container). [123] Firstly, the dockerfile creates a folder within the layered filesystem of Docker and copies the contents of the analysis module to this folder. It then installs the R package which contains the developed analysis module' R files.

The layered filesystem of Docker creates a barrier of access between the machine and the Docker container. In order to surpass this, the usage of a volume is required. This volume allows the output file from the running Docker container to be written on the local hosting machine's filesystem. The specifications for its construction are available on the Docker-Compose file. Whether is the first time to run the containerized Docker image or in the event of having modified the analysis' module R files, the *docker-compose build* command has to be executed on a command terminal. After making the build, the Docker image of the analysis module can be executed on a Docker container. To do this, it is necessary to execute the *docker-compose up* command. After having the container in a running state (which can be verified on another terminal, using the *docker ps* command), another command terminal is necessary to execute the program. By using the *docker exec* on the running container, it is then possible to specify the input file, output file and the maximum size of the Markov chains, and run the program using the RScript command, obtaining its results on the previously defined output file.

After developing the analysis module, the packaging of the tool for the deployment stage provides an easy way to distribute and execute the application. After setting up the tool in its production environment, we proceed to evaluate the results obtained from this method of generating tests. This topic will be addressed in chapter 6 of this work.

# Chapter 6

# Evaluation

The initial evaluation of the approach was focused on code coverage and plausibility. Large numbers of tests were generated and these two dimensions were observed in terms of evolution. The goals were to assess the quality of the generated tests, as well as determining the minimum number of tests that must be generated to ensure quality.

The metrics used are Proportion of Actions Covered (PAC) and the Kullback-Leibler's (KL) divergence [124]. PAC is the ratio between the number of distinct actions in real sessions and the number of distinct actions in sessions built by the analysis module.

This metric does not measure the code coverage directly since it is based on the captured users' actions. This means that if a part of the code is never involved in real sessions, it is not tested. Despite this, the more PAC grows, the more code is tested. The Kullback-Leibler's divergence [124] is a comparison measure between two probabilities' distributions. By using it, it is possible to compare the distribution of events in the real session with the generated ones. The formula to calculate this divergence is presented below.

$$D_{\mathrm{KL}}(P||Q) = \sum_{i} P(i) log \frac{P(i)}{Q(i)}$$

## 6.1 PAC

The interactions' file used for the evaluation section was result of a one week gathering of user actions and it contains 19662 recorded actions. A test for the code coverage was performed using a growing number of $N$ and a maximum value for $L$.

- $N = \{1, 2, 3, 4, 5, 6\}$
- $L = 600$

N should be a not too high nor not too low value. Too low values won't cover enough events and its different possibilities, while too high values will have high computational costs and can become too complex, not revealing useful behaviour patterns. The parameter L may vary depending the application to be tested. Prior experimentation is needed to observe which value can lead to satisfying results.

For each value of $N$, a number of $0$ to $600$ tests were generated and cumulatively measured the coverage of the tests. As detailed before, this is done by comparing the set of GUI events occurring in each subset of the generated tests with the full set of events. In figure 6.1, the growth of PAC with the increasing number of generated tests can be visualized, reaching a value of $1$ to $N = 1$ and nearly $1$ for the remaining $N$. The final obtained values of the PAC metric are presented on table 6.1.

| N | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| PAC | 1 | 0.996 | 0.996 | 0.993 | 0.993 | 0.993 |

**Table 6.1:** PAC for each N parameter

This metric is empirically cumulative (for a specific $N$ value, we can never have more code covered with fewer sequences). This means that the most important value is the last one obtained, but it is also important to have into account the behavior presented by the plot. The value of PAC starts to grow steadily and then tends to stabilize. The highest code coverage value was obtained using $N = 1$, where it reaches a value of full code coverage. Although there is no clear relation between $N$ and the measured value of PAC, higher values

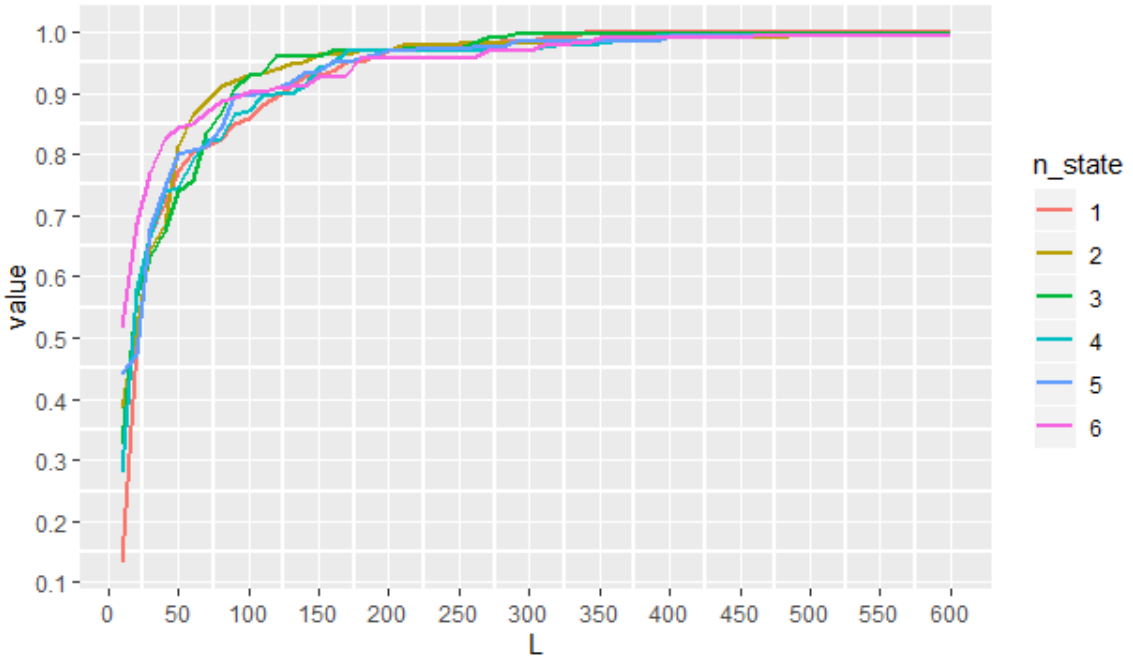



**Figure 6.1:** Evaluation results for PAC

of $N$ do not seem to pay off in terms of coverage. This stems from the fact that high values of $N$ have more restricted and specific actions. In any case, a number of 600 test cases already offers excellent coverage.

## 6.2 Kullback-Leibler's divergence

An experiment similar to the previous one was executed for measuring the plausibility of the generated tests. This was done by measuring the $KL$ of the produced distribution of events given the observed one. The obtained results are shown in Figure 6.2.

A KL value close to 0 indicates that the generated sequences are a good representation of real sequences. From the plot in Figure 6.2 and the values on table 6.2, it is easy to see that KL tends to zero for all values of $N$. As it can also be seen on the graphic, with $L > 2500$ test cases, plausible distributions for all $N$ are obtained. This shows that various safe pairs, $L$ and $N$, can be found. By combining both evaluation dimensions, and taking into account that lower values of $N$ and $L$ are preferable for computational reasons, right combinations for this use case would be $N \in \{1, 2, 3\}, L \geq 2500$.

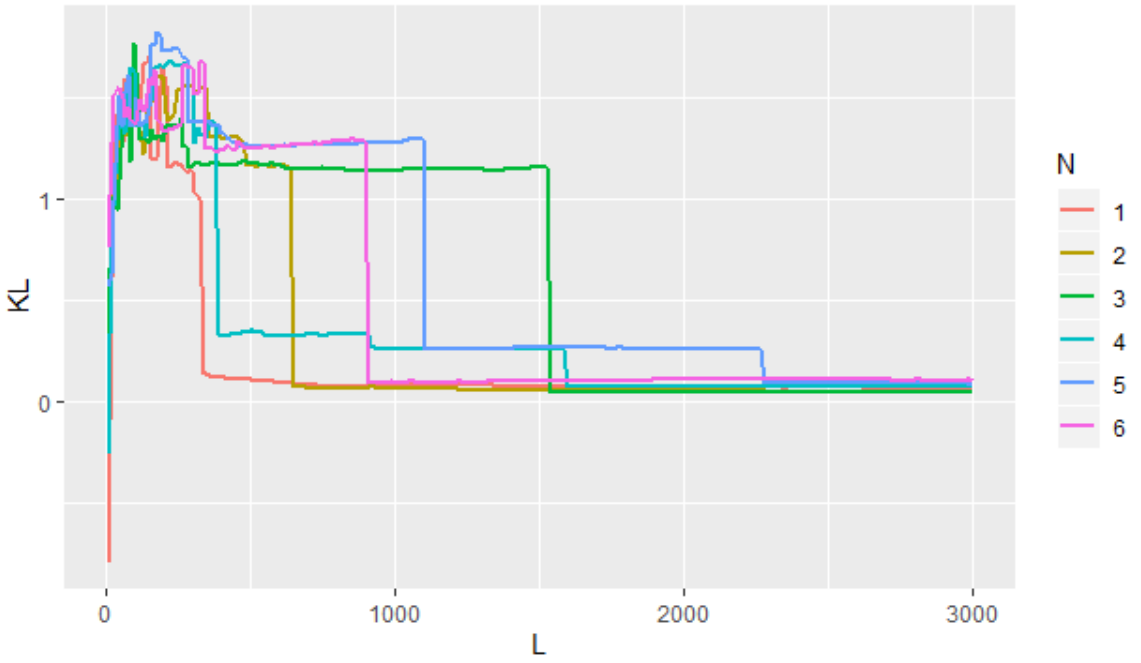



**Figure 6.2:** Evaluation results for the Kullback-Leibler divergence

The evaluation done by these two metrics is profitable, but it still lacks in some aspects. The Kullback-Leibler's divergence does not compare distributions of micro sequences but only of individual items. Also, there is no metric to evaluate the effectiveness of a test suite (its capability to find a bug). Two more metrics will be introduced in order to handle these issues.

## 6.3 Chi-square

To address the case of the Kullback-Leibler's divergence not being able to compare distributions of micro sequences, the chi-square statistical test was implemented. This nonparametric test can be used for two specific purposes: testing the hypothesis of no association between two or more groups or for some determined criteria (such as, for example, checking if two variables are variables), and to test goodness-of-fit (verifying how well the observed distribution of data fits with the distribution that is expected). Since it is used to analyze categorical data (such as the binary data from the use case, of knowing if a business concept was used in a test or not), it suits the goal of understanding if there is a relation

| N | L=750 | L=1500 | L=2250 | L=3000 |
|---|---|---|---|---|
| 1 | 0.079 | 0.078 | 0.073 | 0.070 |
| 2 | 0.067 | 0.061 | 0.053 | 0.050 |
| 3 | 1.157 | 1.163 | 0.045 | 0.044 |
| 4 | 0.327 | 0.261 | 0.074 | 0.072 |
| 5 | 1.269 | 0.268 | 0.262 | 0.089 |
| 6 | 1.282 | 0.103 | 0.113 | 0.108 |

**Table 6.2:** Kullback-Leibler's divergence results

between the expected probability distribution, produced from the original user interactions, and the original probability distribution, derived from generating sequences by the Markov model [125].

It is also important that the data present on the use case follows some assumptions for the chi-square test results to be valid for taking conclusions. The data is randomly drawn from the test-suite, the sample size is sufficiently large and the variables are mutually exclusive (if a business concept is present in a test, it can not also be counted as not present). The formula for the chi-square statistic is given by:

$$\tilde{\chi}^2 = \sum_{k=1}^{n} \frac{(O_k - E_k)^2}{E_k}$$

Where $O$ stands for the observed frequency and $E$ stands for the expected frequency. The observed count (in this case, the probabilities of a business concept being triggered) is subtracted by expected count to find the difference between the two. The square of the difference is calculated in order to eliminate negative values and divided by the expected count to normalize the chi-square value (to avoid having bigger chi-square values simply because of the large size of the data). The values obtained for every business concept are summed up, reaching the final chi-square value.

The achieved results from the chi-square test show that the values of this statistic are very high initially for the low values of $N$, but arrive fast to 0 (where the expected and the observed frequencies are equal), opposite to the higher values of $N$, where this statistic

starts off lower than for small $N$ values, but travels slower to 0. These results can be seen on the graphic below:

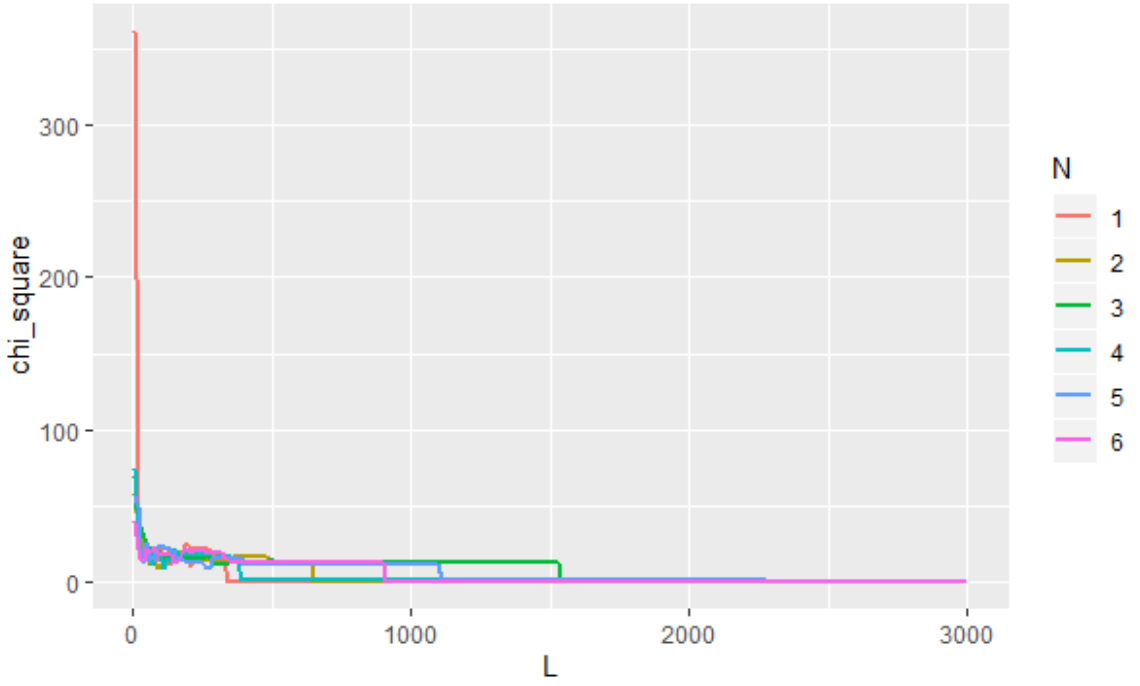



**Figure 6.3:** Evaluation results for the chi square metric

## 6.4 DDU

As referred over chapter 3, DDU [71], which stands for Density Diversity Uniqueness, is a metric used to assess a test suite effectiveness. It measures the density, diversity and uniqueness of a test suite and combines all of these three metrics to measure the probability of a bug being found in that test suite. It applies a normalized $\rho$' metric, derived from the $\rho$ metric, which captures the density of a system, the Gini-Simpson index to perceive diversity and a metric proposed by Baudry et. al [72] to identify the number of dynamic basic blocks in a system to estimate uniqueness. These three measurements are then combined through multiplication, providing a result between 0.0 and 1.0, being 1.0 the best result possible.

### 6.4.1 Density

The $\rho$ metric, used to measure the density of a matrix, can be used as a measurement to evaluate the efficiency of a test suite. Considering $A$ is a binary matrix, in which 1 represents a business concept used on a test and 0 represents the opposite. This matrix is composed by $N$ columns (where each one represents a test) and $M$ rows (where each one represents a business concept). The $\rho$ metric is given by the total sum of used business concepts dividing by the number of columns and rows of the matrix:

$$\rho = \frac{\sum_{i,j} A_{ij}}{N \times M}$$

In order to maximize the information gain from each failure, the optimal value for this metric is 0.5. To obtain a normalized value, in between the interval of 0.0 and 1.0, where 1.0 is the best possible value, the formula is the following:

$$\rho = 1 - |1 - 2 \times \rho|$$

Obtaining this way the $\rho$' metric which is used by DDU for the calculation of density.

### 6.4.2 Diversity

The optimal target of the density metric is only valid assuming that all transactions in the activity matrix are distinct. However, this assumption is is not encoded in the metric itself, which means that a matrix with no diversity is able to reach the ideal value for the metric $\rho$'.

In order to fix this issue, it is necessary to add a check for test diversity to the $\rho$' metric. The gain from this measure is that, by having a great variety in the recorded transactions, it is easier to perceive which component has a bigger probability of being responsible of causing an error in the system. This leads to a more accurate representation of the state of the system [71].

The Gini-Simpson index to used to estimate the diversity $\mathcal{G}$ [126]. For the use case, the $\mathcal{G}$ metric computes the probability of two tests, selected at random, which utilize a determined business concept, having a different activity pattern:

$$\mathcal{G} = 1 - \frac{\sum n \times (n-1)}{N \times (N+1)}$$

where n is the number of tests that share the same activity. When $\mathcal{G}$=1, every test has a different activity pattern. When $\mathcal{G}$=0, all tests have equal activity.

### 6.4.3 Uniqueness

To evaluate uniqueness, a metric proposed by Baudry et al. was used [72]. It measures the number of dynamic basic blocks in a system (also referred to as ambiguity groups [127]), which are groups of components that exhibit the same usage pattern on the entire test-suite. For this use case, this measurement tracks the ratio of between the number of unique tests performed (tests that utilize different business) concepts and the total number of tests. Therefore, the formula to measure the uniqueness $\mathcal{U}$ of a system is given by:

$$\mathcal{U} = \frac{|G|}{M}$$

When $\mathcal{U}$ = 1/M, all the tests available on the test-suite perform the exact same user actions. On the contrary, if $\mathcal{U}$=1, all tests have a unique usage pattern in terms of triggered business concepts.

### 6.4.4 Results

Utilizing a test suite of generated sequences from a file encompassing 19662 user interactions, containing 279 business concepts, and with parameters of $N$=7 and $L$=2000, where the optimal value is reached, the DDU metrics present a total value of 0.004. This number, however, does not capture the effectiveness of the test suite, and the reason for it is the very low density value (0.03). This is caused by the large difference between used and unused business concepts for each test. Although the final value of DDU is compromised from this fact, diversity and uniqueness present important indicators over the test suite's performance. The diversity of tests value is very high (0.97), which means that if an error occurs, it is easy to understand which business concept caused it. Despite

the result on diversity, the measured value of uniqueness on the test suite is very low (0.125). This means that the generated sequences contain only 12.5% of unique tests. Although it is expected that the random sampling generates repeated tests, a higher value of uniqueness would represent a broader exploration of the application' paths. In order to address this issue, improvements to the process of generating sequences were made. These modifications will be presented over the following section.

## 6.5 Sequences' generation improvements

To improve the results of the uniqueness of metric, the approach taken was to make changes on the method for generating test sequences were made. Considering the Markov model generates test cases using a proportional sampling criterion, it is possible that future state choices with low probabilities might not be fully explored. This subject can be tackled by changing the probabilities according to the choices performed by the random sampling. Whenever a state is chosen, its probability of being chosen is reduced by dividing the original probability by the number of possible state choices and subtracting this difference from the original probability. This difference is divided for the number of not chosen states and the result is added to their probabilities, keeping them balanced, compelling the model to explore all possible states. These modifications take more execution time compared to the original method (45 seconds of the original method versus 70 seconds of the modified method).

In practice, and using the same previous parameters to generate the test suite, this modification lowered both the diversity (to 0.96) and the uniqueness values (to 0.105) from the tests. This happens because of the fact that states with less probabilities of being chosen presented further states fewer future choices, which diminishes the uniqueness and the diversity from the obtained tests.

Testing both the original and modified methods to generate sequences, and utilizing a file encompassing 3018 user interactions, containing 500 business concepts, and with the same previous parameters of $N$and $L$, the obtained results show a different reality. The uniqueness rises from 0.165 to 0.203 and the diversity slightly increases from 0.973 to

0.979. The larger number of business concepts creates a bigger diversity of paths, which are more vastly explored by the modified method to generate sequences.

Although PAC presents slightly bigger coverage for most values of $N$, using the original method, the obtained value for this metric, under the usage of the modified version of the generating sequences' algorithm, is significantly bigger for $N$=6. This explains the previous obtained results.

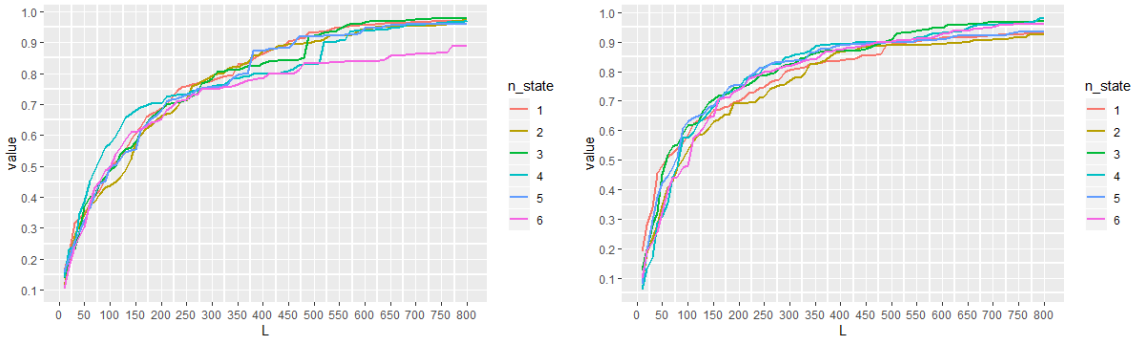



**Figure 6.4:** Histograms of PAC for the original and modified methods

| N | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| PAC (Original) | 0.970 | 0.972 | 0.978 | 0.972 | 0.958 | 0.888 |
| PAC (Modified) | 0.928 | 0.924 | 0.970 | 0.980 | 0.936 | 0.962 |

**Table 6.3:** Comparison of PAC for the original and modified methods

Both the chi-square and the Kullback-Leibler's divergence show similar results. For the modified version, these measures need a little bit less of 3000 test sequences to converge to the optimal value of 0, where the difference between the generated test sequences and the original user interactions is minimal. This does not occur for the non modified version of the algorithm, where this convergence occurs only for smaller values of $N$. In practice, this means that, for almost all values of $N$, the original version of the algorithm needs to generate more tests to obtain an accurate representation of the performed user interactions in the system.

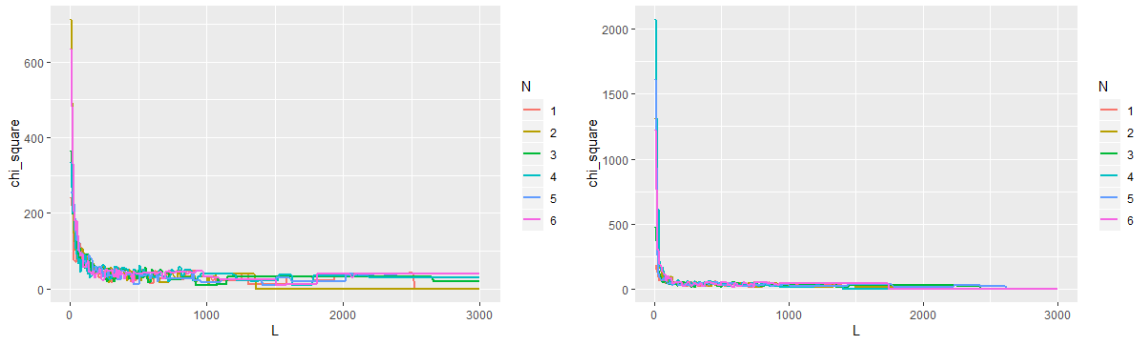



**Figure 6.5:** Histograms of chi-square for the original and modified methods

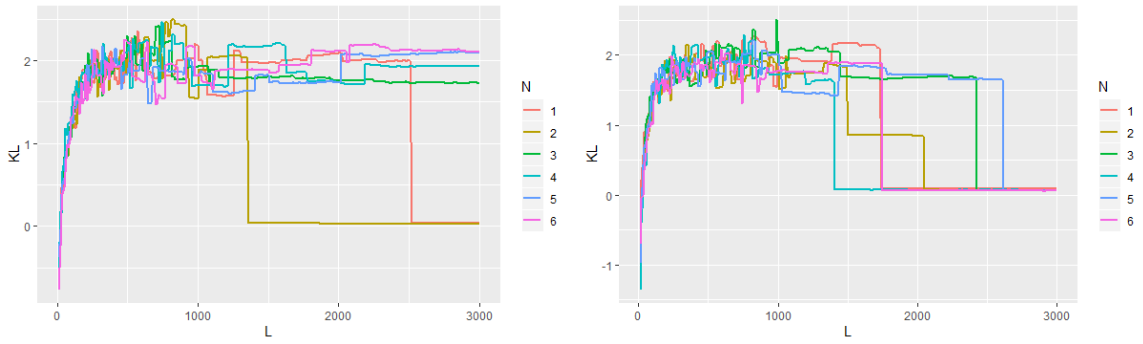



**Figure 6.6:** Histograms of KL for the original and modified methods

The final conclusions of this work will be addressed in its final chapter.

# Chapter 7

# Conclusions

This work presents a tool that automatically generates software tests based on GUI event logs. This approach potentially highly reduces the time, costs and efforts from developing software tests and, therefore, finding failures more efficiently. This framework can be easily adapted for other Web-based GUI software applications, requiring only a fair amount of fixed users in order to easily collect a high amount of interactions. The deployment of the tool provides an easy way to distribute it and to use it for any kind of user, even without any knowledge of its internal way of functioning. Due to operational restrictions in the company, the tool has not been yet tested. The plan is to objectively measure the impact of using the test tool in operation.

Applying sequence mining to the logs and building a Markov model for new tests' generation provides a good overview and understanding of the system and how it is used. Through the usage of evaluation metrics, it is possible to observe the performance of this method and tune its input parameters. Not only the amount of covered business concepts is viewed, but also the difference between the probability distributions from the original user interactions and the test sequences generated from the Markov model, providing safe values for the number of software tests ($L$) and the number of previous items ($N$) to take into account. The approach of balancing the probabilities according to the choices performed by the random sampling showed better results than simply maintaining the probabilities obtained from the Markov model when the system with a fair number of business concepts and sequence paths, compelling the model to explore all possible states. The test suite obtained from this

process is also assessed in terms of its effectiveness for finding failures, both in terms of uniqueness and diversity from the tests.

## 7.1 Future Work

For future work, it is necessary to perform the evaluation of the test tool in a real environment. Also, directly measuring plain code coverage can prove to be relevant, considering that, if some portions of code from the system are never involved in real sessions, they are not tested. A dashboard tool containing performance indicators originated by the evaluation metrics would also be an useful feature to develop, for easy access of both obtained values for the metrics and secure $L$ and $N$ values. Different models may also present improvements on the generation of tests, such as using long sequence term memory (LSTM) network (building a binary matrix of business concepts, where 1 means there is a path transition between two concepts and generating test cases from these transitions) instead of a Markov model. Lastly, the integration of this framework with the e-learning component poses various advantages for the user, by providing visual tips to guide the user through the multiple actions of the application.

# References


[1] A. Ajouli and K. Henchiri. Modem: an uml profile for modeling and predicting software maintenance before implementation. In *2019 International Conference on Computer and Information Sciences (ICCIS)*, pages 1–5, 2019.

[2] M. Choetkiertikul, H. K. Dam, T. Tran, and A. Ghose. Predicting delays in software projects using networked classification (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 353–364, November 2015.

[3] R. Florea and V. Stray. The skills that employers look for in software testers. *Software Quality Journal*, 2019.

[4] Mark Last, Menahem Friedman, and Abraham Kandel. Using data mining for automated software testing. *International Journal of Software Engineering and Knowledge Engineering*, 14(4):369–393, 2004.

[5] Chunhui Wang, Fabrizio Pastore, Arda Goknil, Lionel Briand, and Zohaib Iqbal. Automatic generation of system test cases from use case specifications. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 385–396, New York, NY, USA, 2015. Association for Computing Machinery.

[6] Lazaro Clapp, Osbert Bastani, Saswat Anand, and Alex Aiken. Minimizing GUI event traces. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 422–434. ACM, 2016.

[7] P. Silva, A. C. R. Paiva, A. Restivo, and J. E. Garcia. Automatic test case generation from usage information. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 268–271, September 2018.

[8] Alberto Oliveira, Ricardo Freitas, Alípio Jorge, Vítor Amorim, Nuno Moniz, Ana C. R. Paiva, and Paulo J. Azevedo. Sequence Mining for Automatic Generation of Software Tests from GUI Event Traces. In Cesar Analide, Paulo Novais, David Camacho, and Hujun Yin, editors, *Intelligent Data Engineering and Automated Learning – IDEAL 2020*, pages 516–523, Cham, 2020. Springer International Publishing.

[9] Alberto Oliveira, Ricardo Freitas, Alípio Jorge, Vítor Amorim, Nuno Moniz, Ana C. R. Paiva, and Paulo J. Azevedo. Demo: Sequence Mining for Automatic Generation of Software Tests from GUI Event Traces. In *INTUITESTBEDS*, October 2020.

[10] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2004.

[11] Maria Halkidi, Diomidis Spinellis, George Tsatsaronis, and Michalis Vazirgiannis. Data mining in software engineering. *Intelligent Data Analysis Journal (IDA)*, 15:413–441, January 2011.

[12] Paolo Palmerini. *On performance of data mining: from algorithms to management systems for data exploration*. Doctoral dissertation, PhD. Thesis: TD-2004-2, Università Ca'Foscari di Venezia, March 2004.

[13] Lisa Grossman. Metric math mistake muffed mars meteorology mission. `https://www.wired.com/2010/11/1110mars-climate-observer-report/`, November 2010. Accessed: 2019-11-19.

[14] Leo Kelion. Fatal a400m crash linked to data-wipe mistake. `"https://www.bbc.com/news/technology-33078767"`, June 2015. Accessed: 2019-11-19.

[15] Muhammad Ali Babar, Alan W. Brown, and Ivan Mistrik. *Agile Software Architecture: Aligning Agile Processes and Software Architectures*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[16] Brian Anderson. Best automation testing tools for 2019 (top 10 reviews). `https://medium.com/@briananderson2209/best-automation-testing-tools-for-2018-top-10-reviews-8a4a19f664d2`, October 2017. Accessed: 2019-12-14.

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[18] Pedro Domingos. *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World*. Basic Books, Inc., USA, 2018.

[19] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. Data management challenges in production machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1723–1726, New York, NY, USA, May 2017. Association for Computing Machinery.

[20] Tom M. Mitchell. *Machine learning, International Edition*. McGraw-Hill Series in Computer Science. McGraw-Hill, 1997.

[21] Du Zhang and J.J.P. Tsai. Machine learning and software engineering. In *Software Quality Journal - SQJ*, volume 11, pages 22 – 29, February 2002.

[22] Kevin P. Murphy. *Machine learning : a probabilistic perspective*. The MIT Press, 2012.

[23] T. Bayes. An essay towards solving a problem in the doctrine of chances. *Phil. Trans. of the Royal Soc. of London*, 53:370–418, 1763.

[24] Pat Langley, Wayne Iba, and, and Kevin Thompson. An analysis of bayesian classifiers. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, AAAI'92, page 223–228. AAAI Press, 1992.

[25] R. Zhu, Y. Dai, T. Li, Z. Ma, M. Zheng, Y. Tang, J. Yuan, and Y. Huang. Automatic real-time mining software process activities from svn logs using a naive bayes classifier. *IEEE Access*, 7:146403–146415, 2019.

[26] N. Dhamayanthi and B. Lavanya. Software defect prediction using principal component analysis and naïve bayes algorithm. In Nabendu Chaki, Nagaraju Devarakonda,

Anirban Sarkar, and Narayan C. Debnath, editors, *Proceedings of International Conference on Computational Intelligence and Data Engineering*, pages 241–248, Singapore, 2019. Springer Singapore.

[27] Jifeng Xuan, Jiang He, Yan Hu, Zhilei Ren, Weiqin Zou, Zhongxuan Luo, and Xindong Wu. Towards effective bug triage with software data reduction techniques. *Knowledge and Data Engineering, IEEE Transactions on*, 27:264–280, January 2015.

[28] Lior Rokach and Oded Maimon. *Decision Trees*, volume 6, pages 165–192. January 2005.

[29] Max Bramer. *Principles of Data Mining*, pages 39–48. Springer London, 2013.

[30] Mrunmayee Khare and Rajvardhan Oak. *Real-Time Distributed Denial-of-Service (DDoS) Attack Detection Using Decision Trees for Server Performance Maintenance*, pages 1–9. Springer Singapore, Singapore, 2020.

[31] Mohammad Y. Mhawish and Manjari Gupta. Software metrics and tree-based machine learning algorithms for distinguishing and detecting similar structure design patterns. *SN Applied Sciences*, 2(1):11, December 2019.

[32] Jaswinder Singh, Kanwalvir Singh, and Jaiteg Singh. Reengineering framework for open source software using decision tree approach. *International Journal of Electrical and Computer Engineering (IJECE)*, 9:2041, June 2019.

[33] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, October 2001.

[34] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, August 1996.

[35] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, page 161–168, New York, NY, USA, 2006. Association for Computing Machinery.

[36] Yunhu Jin, Y. Shen, G. Zhang, and Hua Zhi. The model of network security situation assessment based on random forest. In *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 977–980, August 2016.

[37] Abdelali Zakrani, Hain Mustapha, and Namir Abdelwahed. Investigating the use of random forest in software effort estimation. *Procedia Computer Science*, 148:343–352, January 2019.

[38] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. Detecting code smells using machine learning techniques: Are we there yet? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 612–621, 2018.

[39] Egor Bogomolov, Vladimir Kovalenko, Alberto Bacchelli, and Timofey Bryksin. Authorship attribution of source code: A language-agnostic approach and applicability in software engineering. *CoRR*, volume abs/2001.11593, 2020.

[40] Jerome Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, volume 29, November 2000.

[41] Warren Sturgis McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 52:99–115, 1990.

[42] Shahab Mohaghegh. Virtual-intelligence applications in petroleum engineering: Part 1 - artificial neural networks. *Journal of Petroleum Technology - J PETROL TECHNOL*, 52:64–73, September 2000.

[43] Enzo Grossi and Massimo Buscema. Introduction to artificial neural networks. *European journal of gastroenterology & hepatology*, 19:1046–54, January 2008.

[44] Yulei Pang, Xiaozhen Xue, and Huaying Wang. Predicting vulnerable software components through deep neural network. In *Proceedings of the 2017 International Conference on Deep Learning Technologies*, ICDLT '17, page 6–10, New York, NY, USA, June 2017. Association for Computing Machinery.

[45] D. Amudhavalli, Dr. S. Rajalakshmi, and Dr. M. Marikannan. An efficient software effort estimation by combining neural network and optimization technique. *International Journal of Applied Engineering Research*, 13(6):3890–3897, 2018.

[46] Poonam Rijwani and Sonal Jain. Enhanced software effort estimation using multi layered feed forward artificial neural network technique. *Procedia Computer Science*, 89:307–312, December 2016.

[47] V. Gupta. *Crowdsourcing and Probabilistic Decision-Making in Software Engineering: Emerging Research and Opportunities: Emerging Research and Opportunities*. Advances in Systems Analysis, Software Engineering, and High Performance Computing. IGI Global, 2019.

[48] Claude Sammut and Geoffrey I. Webb. *Encyclopedia of Machine Learning*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[49] Patrick Kwaku Kudjo and Jinfu Chen. A cost-effective strategy for software vulnerability prediction based on bellwether analysis. In Dongmei Zhang and Anders Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 424–427. ACM, 2019.

[50] Jianglin Huang, Jacky Wai Keung, Federica Sarro, Yan-Fu Li, Yuen-Tak Yu, W. K. Chan, and Hongyi Sun. Cross-validation based K nearest neighbor imputation for software quality datasets: An empirical study. *Journal of Systems and Software*, 132:226–252, 2017.

[51] Mohsen Hasanluo and Farhad Soleimanian Gharehchopogh. Software cost estimation by a new hybrid model of particle swarm optimization and k-nearest neighbor algorithms. *Journal of Electrical and Computer Engineering Innovations*, 4(1):49–55, 2016.

[52] Christoph Molnar. *Interpretable Machine Learning*. Lulu, 1st edition, March 24, 2019; eBook (GitHub, 2019-06-19), 2019.

[53] Viv Bewick, Liz Cheek, and Jonathan Ball. Statistics review 14: Logistic regression. *Critical care (London, England)*, 9:112–118, March 2005.

[54] Md Saidur Rahman, Emilio Rivera, Foutse Khomh, Yann-Gaël Guéhéneuc, and Bernd Lehnert. Machine learning software engineering in practice: An industrial case study. *CoRR*, volume abs/1906.07154, 2019.

[55] Yasunari Takagi, Osamu Mizuno, and Tohru Kikuno. An empirical approach to characterizing risky software projects based on logistic regression analysis. *Empirical Software Engineering*, 10(4):495–515, 2005.

[56] Thitima Christiansen, Pongpisit Wuttidittachotti, Somchai Prakancharoen, and Sakda Arjong Vallipakorn. Prediction of risk factors of software development project by using multiple logistic regression. *ARPN Journal of Engineering and Applied Sciences*, 10(3):1324–1331, 2015.

[57] Venkata Subba Reddy and Dr. B. Raveendra Babu. Logistic regression approach to software reliability engineering with failure prediction. *International Journal of Software Engineering & Applications (IJSEA)*, volume 4(1), January 2013.

[58] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. *Proc. 20th Int. Conf. Very Large Data Bases VLDB*, volume 1215, August 2000.

[59] Muhammad Asif, Ahmad Jamil, and Abdul Hannan. Software risk factors: A survey and software risk mitigation intelligent decision network using rule based technique. In *2018 International Conference on Advancements in Computational Sciences (ICACS)*, 2018.

[60] S. AlZu'bi, B. Hawashin, M. EIBes, and M. Al-Ayyoub. A novel recommender system based on apriori algorithm for requirements engineering. In *2018 Fifth International Conference on Social Networks Analysis, Management and Security (SNAMS)*, pages 323–327, October 2018.

[61] R. Anand and M. Dinakaran. Handling stakeholder conflict by agile requirement prioritization using apriori technique. *Computers & Electrical Engineering*, 61:126–136, July 2017.

[62] K. Yoon, O. Kwon, and D. Bae. An approach to outlier detection of software measurement data using the k-means clustering method. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 443–445, October 2007.

[63] Xiaozhou Li. Research on software project developer behaviors with k-means clustering analysis. In *SSSME 2019 : Joint Proceedings of the Summer School on Software Maintenance and Evolution*, pages 54–61, September 2019.

[64] Shi Zhong, Taghi Khoshgoftaar, and Naeem Seliya. Analyzing software measurement data with clustering techniques. *Intelligent Systems, IEEE*, 19:20– 27, April 2004.

[65] Jinfu Chen, Minmin Zhou, T. H. Tse, Tsong Yueh Chen, Yuchi Guo, Rubing Huang, and Chengying Mao. Toward a k-means clustering approach to adaptive random testing for object-oriented software. *Science China Information Sciences*, 62(11):219105, September 2019.

[66] K. Meinke and A. Bennaceur. Machine learning for software engineering: Models, methods, and applications. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 548–549, 2018.

[67] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, USA, 1 edition, 2008.

[68] Quadri S.M.K and Sheikh Umar Farooq. Software testing – goals, principles, and limitations. *International Journal of Computer Applications*, volume 6, September 2010.

[69] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. Code coverage at google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 955–963, New York, NY, USA, 2019. Association for Computing Machinery.

[70] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 435–445, New York, NY, USA, 2014. Association for Computing Machinery.

[71] A. Perez, R. Abreu, and A. van Deursen. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 654–664, May 2017.

[72] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. volume 2006, pages 82–91, January 2006.

[73] Eric Steegmans, Pieter Bekaert, F. Devos, Geert Delanote, N. Smeets, Marko Dooren, and Jeroen Boydens. Black & white testing: Bridging black box testing and white box testing. January 2004.

[74] Philip Samuel and Rajib Mall. A novel test case design technique using dynamic slicing of uml sequence diagrams. *e-Informatica*, 2:71–92, 2008.

[75] Debasish Kundu and Debasis Samanta. A novel approach to generate test cases from uml activity diagrams. *Journal of Object Technology*, 8:65–83, May 2009.

[76] Santosh Kumar Swain, Durga Prasad Mohapatra, and Rajib Mall. Test case generation based on state and activity models. *Journal of Object Technology*, 9:1–27, 2010.

[77] Itti Hooda and Rajender Singh Chhillar. A review: Study of test case generation techniques. *International Journal of Computer Applications*, 107:33–37, 2014.

[78] George Candea and Patrice Godefroid. *Automated Software Test Generation: Some Challenges, Solutions, and Recent Advances*, pages 505–531. January 2019.

[79] Thomas Wetzlmaier and Rudolf Ramler. Hybrid monkey testing: Enhancing automated gui tests with random test generation. In *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing*, A-TEST 2017, page 5–10, New York, NY, USA, September 2017. Association for Computing Machinery.

[80] R. Ramler, D. Winkler, and M. Schmidt. Random test case generation and manual unit testing: Substitute or complement in retrofitting tests for legacy code? In *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 286–293, 2012.

[81] T. Y. Chen. Adaptive random testing. In *2008 The Eighth International Conference on Quality Software*, pages 443–443, 2008.

[82] Walter J. Gutjahr. Software dependability evaluation based on markov usage models. *Perform. Eval.*, 40(4):199–222, 2000.

[83] Kuanjiu Zhou, Xiaolong Wang, Gang Hou, Jie Wang, and Shanbin Ai. Software reliability test based on markov usage model. *JSW*, 7(9):2061–2068, 2012.

[84] Sebastian Siegl, Winfried Dulz, Reinhard German, and Gerhard Kiffe. Model-driven testing based on markov chain usage models in the automotive domain. In *12th European Workshop on Dependable Computing (EWDC 2009)*, May 2009.

[85] Deepak Kumar and Manu Phogat. Genetic algorithm approach for test case generation randomly: A review. *International Journal of Computer Trends and Technology*, 49:213–216, July 2017.

[86] Rizwan Khan and Mohd Amjad. Automatic generation of test cases for data flow test paths using k-means clustering and generic algorithm. *International Journal of Applied Engineering Research*, 11:473–478, February 2016.

[87] Achmad Arwan and Denny Rusdianto. Optimization of genetic algorithm performance using naïve bayes for basis path generation. *KINETIK*, volume 2, September 2017.

[88] Ute Zeppetzauer and Peter M. Kruse. Automating test case design within the classification tree editor. *2014 Federated Conference on Computer Science and Information Systems*, pages 1585–1590, 2014.

[89] Matthias Grochtmann, Joachim Wegener, and Klaus Grimm. Test case design using classification trees and the classification-tree editor cte. *Proceedings of the 8th International Software Quality Week*, January 1995.

[90] Prachi Saraph, Mark Last, and Abraham Kandel. Test set generation and reduction with artificial neural networks. *Artificial Intelligence Methods in Software Testing*, pages 101–132, June 2004.

[91] Ye Mao, Feng Boqin, Zhu Li, and Lin Yao. Neural networks based automated test oracle for software testing. In *Proceedings of the 13th International Conference on Neural Information Processing - Volume Part III*, ICONIP'06, page 498–507, Berlin, Heidelberg, 2006. Springer-Verlag.

[92] Marco Cunha, Ana CR Paiva, Hugo Sereno Ferreira, and Rui Abreu. Pettool: a pattern-based gui testing tool. In *2010 2nd International Conference on Software Technology and Engineering*, volume 1, pages V1–202. IEEE, 2010.

[93] Lee J. White. Regression testing of gui event interactions. In *Proceedings of the 1996 International Conference on Software Maintenance*, ICSM '96, page 350–358, USA, 1996. IEEE Computer Society.

[94] A. Isabella and Emi Retna. Study paper on test case generation for GUI based testing. *CoRR*, volume abs/1202.4527, 2012.

[95] Rudolf Ramler, Georg Buchgeher, and Claus Klammer. Adapting automated test generation to GUI testing of industry applications. *Information & Software Technology*, 93:248–263, 2018.

[96] Kanglin Li and Mengqi Wu. Effective gui testing automation: Developing an automated gui testing tool. Sybex Publications, November 2004.

[97] T. Wetzlmaier, R. Ramler, and W. Putschögl. A framework for monkey gui testing. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 416–423, 2016.

[98] Christoph Csallner and Yannis Smaragdakis. Jcrasher: An automatic robustness tester for java. *Softw., Pract. Exper.*, 34:1025–1050, September 2004.

[99] Catherine Oriat. Jartege: A tool for random generation of unit tests for java classes. In Ralf Reussner, Johannes Mayer, Judith A. Stafford, Sven Overhage, Steffen Becker, and Patrick J. Schroeder, editors, *Quality of Software Architectures and Software Quality*, pages 242–256, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[100] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, page 815–816, New York, NY, USA, 2007. Association for Computing Machinery.

[101] Stephan Arlt, Simon Pahl, Cristiano Bertolini, and Martin Schäf. Trends in model-based gui testing. *Advances in Computers*, 86:183–222, July 2012.

[102] Miguel Pinto, Marcelo Gonçalves, Paolo Masci, and José Campos. Tom: A model-based gui testing framework. In *Formal Aspects of Component Software: 14th International Conference, (FACS 2017)*, pages 155–161, October 2017.

[103] Laryssa Lima Muniz, Ubiratan S. C. Netto, and Paulo Henrique M. Maia. Tcg - a model-based testing tool for functional and statistical testing. In *17th International Conference on Enterprise Information Systems (ICEIS)*, 2015.

[104] Bao Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. Guitar: An innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, volume 21, March 2014.

[105] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*, pages 39–76. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[106] M. Bharati and Bharati Ramageri. Data mining techniques and applications. *Indian Journal of Computer Science and Engineering*, volume 1, December 2010.

[107] Carl Mooney and John Roddick. Sequential pattern mining: Approaches and algorithms. *ACM Computing Surveys*, volume 45, June 2013.

[108] Wensheng Gan, Jerry Chun-Wei Lin, Philippe Fournier-Viger, Han-Chieh Chao, and Philip S. Yu. A survey of parallel sequential pattern mining. *ACM Trans. Knowl. Discov. Data*, 13(3), June 2019.

[109] Thabet Slimani and Amor Lazzez. Sequential mining: Patterns and algorithms analysis. *ArXiv*, volume abs/1311.0350, 2013.

[110] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 3–14, 1995.

[111] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In Peter Apers, Mokrane Bouzeghoub, and Georges Gardarin, editors, *Advances in Database Technology — EDBT '96*, pages 1–17, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[112] F. Masseglia, F. Cathala, and P. Poncelet. The psp approach for mining sequential patterns. In Jan M. Żytkow and Mohamed Quafafou, editors, *Principles of Data Mining and Knowledge Discovery*, pages 176–184, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[113] Ben Kao, Minghua Zhang, Chi Lap Yip, David W. Cheung, and Usama M. Fayyad. Efficient algorithms for mining and incremental update of maximal frequent sequences. *Data Min. Knowl. Discov.*, 10(2):87–116, 2005.

[114] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12, May 2000.

[115] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Freespan: Frequent pattern-projected sequential pattern mining. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, page 355–359, New York, NY, USA, 2000. Association for Computing Machinery.

[116] Jian Pei, Jiawei Han, B. Mortazavi-Asl, H. Pinto, Qiming Chen, U. Dayal, and Mei-Chun Hsu. Prefixspan,: mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings 17th International Conference on Data Engineering*, pages 215–224, 2001.

[117] Philippe Fournier-Viger, Antonio Gomariz, Manuel Campos, and Rincy Thomas. Fast vertical mining of sequential patterns using co-occurrence information. In Vincent S. Tseng, Tu Bao Ho, Zhi-Hua Zhou, Arbee L. P. Chen, and Hung-Yu Kao, editors, *Advances in Knowledge Discovery and Data Mining*, pages 40–52, Cham, 2014. Springer International Publishing.

[118] Jiong Yang, Wei Wang, Philip Yu, and Jiawei Han. Mining long sequential patterns in a noisy environment. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, October 2002.

[119] Mohammed Zaki. Spade: An efficient algorithm for mining frequent sequences. machine learning. *Machine Learning*, 42:31–60, January 2001.

[120] Jay Ayres, Jason Flannick, Johannes Gehrke, and Tomi Yiu. Sequential pattern mining using a bitmap representation. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, page 429–435, New York, NY, USA, 2002. Association for Computing Machinery.

[121] Martin Fowler and Matthew Foemmel. Continuous integration. `https://martinfowler.com/articles/continuousIntegration.html`, May 2006. Accessed: 2019-12-14.

[122] Babak Bashari Rad, Harrison Bhatti, and Mohammad Ahmadi. An introduction to docker and analysis of its performance. *IJCSNS International Journal of Computer Science and Network Security*, 173:8, March 2017.

[123] Carl Boettiger and Dirk Eddelbuettel. An introduction to rocker: Docker containers for r. *R Journal*, volume 9, October 2017.

[124] Solomon Kullback. *Information Theory and Statistics*. Wiley, New York, 1959.

[125] Rakesh. Rana and Richa. Singhal. Chi-square test and its application in hypothesis testing. *Journal of the Practice of Cardiovascular Sciences*, 1(1):69–71, 2015.

[126] Lou Jost. Entropy and diversity. *Oikos*, 113(2):363–375, 2006.

[127] Alberto Gonzalez-Sanchez, Rui Abreu, Hans-Gerhard Gross, and Arjan J. C. van Gemund. Prioritizing tests for fault localization through ambiguity group reduction. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, page 83–92, USA, 2011. IEEE Computer Society.