UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Késsia Nepomuceno

A Machine Learning Approach to Escaped Defect Analysis

Recife
2022

Késsia Nepomuceno

A Machine Learning Approach to Escaped Defect Analysis

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito para obtenção do título de Mestre em Ciência da Computação.

Área de Concentração: Inteligência Computacional
Orientador: Prof. Dr. Ricardo Bastos Cavalcante Prudêncio

Recife
2022

Catalogação na fonte
Bibliotecária Luiza Maria Pereira de Oliveira, CRB4-1316

N441m Nepomuceno, Késsia Thais Cavalcanti
A machine learning approach to escaped defect analysis /Késsia Thais Cavalcanti Nepomuceno. – 2022.
73 f.: il., fig., tab.

Orientador: Ricardo Bastos Cavalcante Prudêncio.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIN, Ciência da Computação, Recife, 2022

Inclui referências.

1. Análise de defeitos escapados. 2. Ranking. 3. Automação. 4. Aprendizagem de máquina I. Prudêncio, Ricardo Bastos Cavalcante (orientador). II. Título.

006.31 CDD (23. ed.) UFPE - CCEN 2022- 173

Késsia Nepomuceno

“A Machine Learning Approach to Escaped Defect Analysis”

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 11/08/2022

Orientador: Prof. Dr. Ricardo Bastos Cavalcante Prudêncio

BANCA EXAMINADORA

Prof. Dr. Alexandre Cabral Mota
CIn - UFPE

Prof. Dr. André Câmara Alves do Nascimento
DeInfo - UFRPE

“If I have seen further, it is by standing on the shoulders of giants.” [Newton, 1965, p. 353-384]

ABSTRACT

Defects in computer systems or applications directly impact the quality and performance of a final product, generating consequences for the user and the supplier. Therefore, identifying the escaped defect not detected by the tester at the proper stage, thus, incorporating it into the product, is one of the software industry's primary activities. To mitigate or eliminate the missing defects, companies usually have a sector responsible for analyzing and evaluating the lost bugs to understand the context in which they are inserted and correct the flaws. The aim is to avoid repetition and improve product quality and test performance. The analysis of escaped defects also measures the testing team's performance and the launch of new products and services. However, despite being a crucial activity, it requires resources such as time, equipment, training and others, making its consistent and precise application unfeasible. Because of this, in partnership with Motorola Mobility, we built a machine learning system to automate the analysis of escaped defects and optimize the manual process, reducing the resources invested in the stages of analysis. For this, the company provided us with information about the process, such as historical data regarding their latest analyzes performed manually by company employees. Thus, our model relies on real industry bug reports for historical data. From the Motorola Bug Report, we collected, processed and used as input to our model the data referring to the escaped and non-escaped defects, and applied Random Forest as the main classifier. As a result, we ranked the Bug Reports most likely to become an escaped defect. To measure the classifier's performance, we used the ROC Curve and a new metric that we proposed, the cost-benefit curve. In both metrics, we obtained significant and promising results. That said, our main contributions with this work were the escaped defect analysis system and the cost-benefit curve metric that we used to measure the performance of our system. Therefore, testers in the software industry will be able to focus and direct their efforts on those Bug Reports that are more or less likely to become an escaped defect, optimizing work operation resources.

Keywords: escaped defect analysis; ranking; automation; machine learning.

RESUMO

Defeitos em sistemas ou aplicações computacionais impactam diretamente a qualidade e performance de um produto final, gerando consequências para o usuário e o fornecedor. Portanto, identificar o defeito escapado não detectado pelo testador na devida etapa, e incorporado ao produto, torna-se uma das principais atividades na indústria de software. Com o objetivo de mitigar ou eliminar os defeitos escapados empresas costumam ter um setor responsável pela análise e avaliação dos bugs perdidos para entender o contexto em que eles estão inseridos e corrigir as falhas. Busca-se evitar a sua repetição e ter um ganho na qualidade do produto e performance dos testes. A análise de defeitos escapados também mede o desempenho da equipe de testes, bem como do lançamento de novos produtos e serviços. Entretanto, apesar de ser uma atividade crucial, ela exige recursos como tempo, equipamentos, treinamentos e outros, tornando-se inviável a sua aplicação consistente e precisa. Por isso, em parceria com a Motorola Mobility, construímos um sistema de aprendizagem de máquina para automatizar a análise de defeitos escapados e otimizar o processo manual, diminuindo os recursos investidos nas etapas da análise. A empresa forneceu-nos informações sobre o processo, tais como os dados históricos referentes às últimas análises feitas de forma manual por funcionários da empresa. Deste modo, nosso modelo conta com Bug Reports reais da indústria para dados históricos. Coletamos, tratamos e utilizamos como entrada para o nosso modelo os dados referentes aos defeitos escapados e não escapados do Bug Report da Motorola e empregamos o Random Forest como classificador principal, resultando no ranking dos Bug Reports com maior probabilidade de se serem um defeito escapado. Para medir o desempenho do classificador, utilizamos a Curva ROC e uma nova métrica que propusemos, a curva de custo-benefício. Em ambas as métricas, obtivemos resultados significativos e promissores. Dito isso, nossas principais contribuições com esse trabalho foram o sistema de análise de efeitos escapados e a métrica curva custo-benefício que utilizamos para medir o desempenho do nosso sistema. Logo, os testadores da indústria de software poderão concentrar e direcionar seus esforços nos Bug Reports com maior ou menor probabilidade de se tornarem um defeito escapado, otimizando recursos de operação de trabalho.

Palavras-chaves: análise de defeitos escapados; ranking; automação; aprendizagem de máquina.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AI	Artificial Intelligence
AUC	Area Under the Curve
CRISP-DM	Cross Industry Standard Process for Data Mining
DDS	Design Document Specification
EDA	Escaped Defect Analysis
EDCR	Escaped Defect Coverage Rate
FN	False Negative
FP	False Positive
FPR	False Positive Rate
GUI	Graphical User Interface
MLB	MultiLabel Binarizer
NN	Nearest Neighbor
PER	Productive Efficiency Rate
QA	Quality Assurance
RegEx	Regular Expression
ROC	Receiver Operating Characteristic Curve
SDLC	Software Development Life Cycle
SMOTE	Synthetic Minority Over-sampling Technique
SPC	Statistical Process Control
SRS	Software Requirements Specification
STLC	Software Testing Life Cycle
TF-IDF	Term Frequency Inverse Document Frequency
TN	True Negative
TNR	True Negative Rate
TP	True Positive
TPR	True Positive Rate

CONTENTS

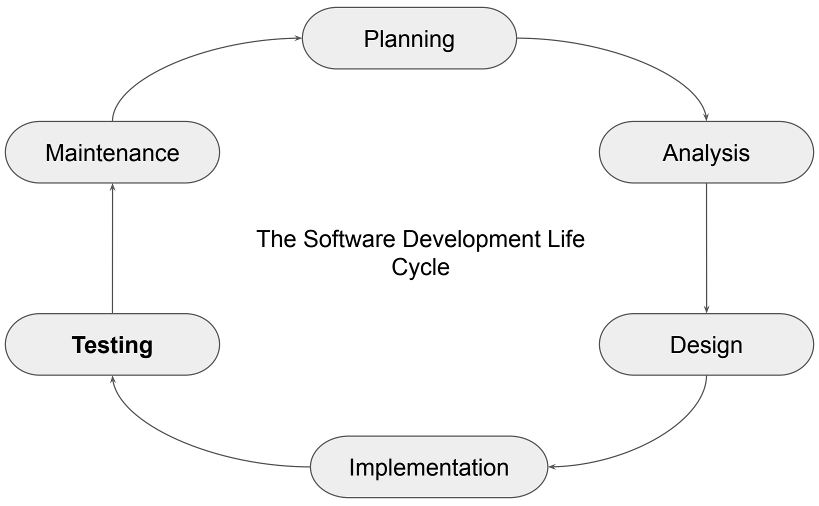
1 INTRODUCTION

1.1 MOTIVATION

The activity of finding bugs in software production is crucial in the Software Development Life Cycle (SDLC) [Davis, Bersoff and Comer 1988] [McCormick 2021]. SDLC indicates testing as one of the main activities. Figure 1 confirms it. We have six stages in total, and testing is one of them. There is a strategy focused on test planning, called the Software Testing Life Cycle (STLC) model, which we will cover better in the background chapter.

Thus, we can note the importance of testing once it has been inserted into the main systematic development process. There is also a systematic testing process to track, develop and update tests. All these things suggest the impact of testing on performance and product quality.

Figure 1 – Software Development Life Cycle.



Source: Authorial, based on: [Davis, Bersoff and Comer 1988] and [McCormick 2021].

Finding bugs is essential to improve the quality of a system. It keeps the product quality high and the consumer satisfied. The proper system functioning impacts the business directly, and most importantly, it affects clients' life and determines if they return or not. The price of not detecting a bug could be high [Herzig et al. 2015, Shull et al. 2002]. Therefore, Table 1 was created based on Raygun's studies about "Resolving bugs early and often reduces associated costs" [Urias and Rivas 2019, Spaven 2017] and presents cost-associated steps of SDLC for uncovered defects. We note that the sooner the defect is found, the lower the cost. So, in the ideal process, the bug has to be detected priorly.

Table 1 – Costs associated with bugs found in software production stages

Stage at which a bug is found	**Cost to fix a bug**
Conception	1x
Design	10x
Development	100x
Testing	1000x
Release	10.000x

The release stage is the most expensive phase for finding a bug. Therefore, finding a defect in this phase could generate serious problems. According to the latest Tricentis report [Tricentis 2016, Eguide 2018], the costs of software failures in 2017 generated US$ 1.7 trillion in financial losses, where 3.6 billion people were affected and a downtime loss of 268 years. This report covered only 606 failures from 314 companies, but we can still see significant numbers of losses. These alarming numbers also show how important it is to pay attention to finding bugs early.

In discovering defects, there is a critical scenario when the test team fails to detect the bug. Not detecting the bug when it should, characterizes an escaped defect [Eickelmann and Anant 2003]. In this work, the escaped defect is our main object for analysis. Therefore, we will consider each document labeled Bug Reports as escaped or not escaped defects.

Considering the purpose of the Quality Assurance (QA) team or the test team is to find defects. The rate of escaped defects is directly related to the team's performance. So, having these rates mapped is vital. Tracking how many defects escaped is a great way to constantly gauge the quality of the software releases and measure the team's work rate of success once this is clear evidence of the effectiveness of the inspection realized by the employee [Yaung and Raz 1994, Eickelmann and Anant 2003].

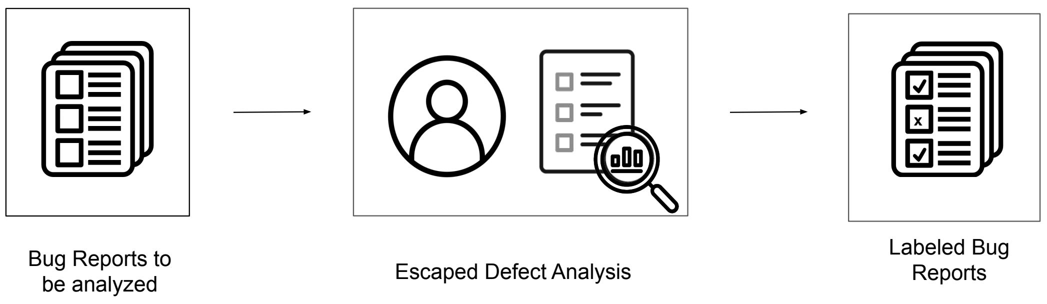
With the analysis of escaped defects, it is possible to establish success rate for the testing and development team, understanding the process's flaws and hits. Walking at a fast pace in software development is excellent; however, setting a threshold for walking is even better. This way, substantial losses are avoided for the business and the customers.

The discovery and analysis of escaped defects can lead us to information about tests not updated, not planned, or missing. Consequently, resulting in the activities performed by the testing team, how much effort to give, costs, and software maintenance.

Escaped defects analysis helps to ensure continuous improvement in testing and development processes as well as in the software product, making it possible to create a plan to avoid future escapes [Vandermark 2003]. As a result, we could mention a reduction in the number of defects found by the user, a decline in costs and an improvement in product quality, sales, and reputation [Vandermark 2003].

Despite the countless benefits associated with the analysis of escaped defects, this process is quite costly since it is necessary to allocate resources, people, and time to carry out the analysis. All of this influences the efficiency of the inspection process [Eickelmann and Anant 2003]. Therefore, although it is a fundamental process to guarantee the quality of the product, it is not always feasible. Usually, this analysis is done as follows: a person is allocated to analyze each parameter of the bug report and identify whether that document is an escaped defect or not (see Figure 2). In the process of manual labeling, usually, the tester takes a long time to explore each Bug Report parameter in detail in order that the bug report does not have a wrong manual classification result.

Figure 2 – Illustration of Escaped Defects Analysis Process. Manual format.



Source: Authorial.

1.2 OBJECTIVE

This work proposes an automated machine learning system that will make the analysis of escaped defects. In addition, it will return a probability ranking of a particular bug report as being an escaped defect or not to optimize the Escaped Defect Analysis (EDA) manual process and mitigate possible subjective results generated by employees.

Through this ranking, the tester will be directed to the bug reports more likely to be escaped defects, saving time in mapping and analysis. Also, the tester can choose, through the ranking, the bug reports preferable to work, according to their characteristics. So, the goal is to create an AI system to automatize one of the software engineering industry processes.

1.3 PROPOSED SOLUTION

In order to achieve the objective, we built a machine learning system to detect when a bug report is an escaped defect or not. We divide the system's development process into modules. Firstly, starting with the model's construction, we followed the usage phase after the model had been generated. Thus, we can summarize the proposed system in the steps below.

- Model building - Receiving labeled bug reports, Data treatment, Model building, and Model generation;
- Model usage - Input the unlabeled bug reports, Application of the model trained, Model evaluation, and Rank generated as the output.

Besides, to validate the results, this work presents two metrics: the Receiver Operating Characteristic Curve (ROC) and a new metric that we call the Cost-Benefit curve.

The ROC curve translates, in a graphic, the performance of a classification model at all classification thresholds [Fawcett 2006]. In this work, we had a ROC of 88% for Team A and 80% for Team B.

The cost-benefit curve translates the gain obtained through the classification model; it considers the escaped defects coverage rate and the productive efficiency rate. For both teams,we had considerable gains of about 70% to 60% of escaped defects coverage for Team A and 90% to 60% for Team B.

1.4 WORK ORGANIZATION

This work is arranged in the following order:

- Chapter 2 presents the central machine learning and software engineering topics necessary to build this work. We also present the main works found, as far as we know, about escaped defects. Finally, we explain important points raised by them and make a comparative analysis.
- Chapter 3 presents our proposed solution, each module necessary to construct and validate our machine learning system.
- Chapter 4 presents our experiments' dataset, testbed, and methodology.
- Chapter 5 presents and discusses the results obtained.
- Chapter 6 concludes our work with an overall review and implications of our studies.

2 BACKGROUND

This chapter will cover in detail essential topics to understand the work realized and the methods chosen. The concepts involve Machine Learning as well as Software Engineering.

2.1 MACHINE LEARNING CONCEPTS

Here we describe machine learning topics related to this work. For better comprehension, other topics will be explained and discussed in their respective chapters; for example, we describe the concepts about the ROC curve in the results; once this chapter presents the ROC curve results.

2.1.1 Confusion Matrix

The confusion matrix, or error matrix, is a key concept to understanding many other subjects, such as the ROC curves, Recall, Precision, Specificity, Accuracy, and others. Calculating a confusion matrix gives us a real understanding of the correctness and incorrectness of the model. In addition, it summarizes the performance measurement for machine learning classification and shows the rate of correct and incorrect predictions, indicating the type and amount of errors and hits.

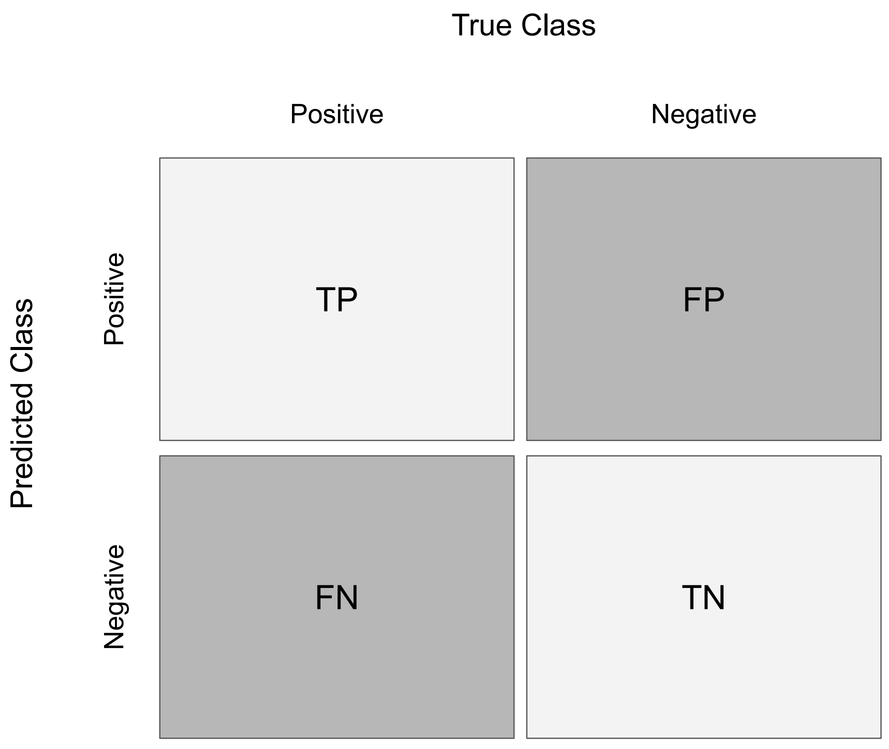
In Figure 3,we can note that there are four combinations considering the True classes and Predicted classes. The rows of the table correspond to the predicted classes. Furthermore, the columns of the table correspond to the true classes. Then, the number of correct and incorrect classifications is filled into the table [Narkhede 2018, Brownlee 2016]. The fundamentals that compose a confusion matrix are the number of True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN) detailed below.

- True Positive
  - Interpretation: The classifier predicts the values relate to the positive class, and the classifier's prediction is correct.
  - The model predicted that a Bug Report is an escaped defect, and it is.
- True Negative
  - Interpretation: The classifier predicts the negative class, and the prediction is correct.
  - The model predicted that a Bug Report is not an escaped defect, and it is not.
- False Positive (Type 1 Error)

  - Interpretation: The classifier predicts the positive class, but the prediction is wrong.
  - The model predicted that a Bug Report is an escaped defect, but it is not.

- False Negative (Type 2 Error)
  - Interpretation: The negative class is predicted by the classifier, but the prediction is wrong.
  - The model predicted that a Bug Report is not an escaped defect, but it is.

Figure 3 – Confusion Matrix representation.



Source: Authorial, based on Confusion Matrix representation.

2.1.2 SMOTE

When leading with real-world data, it is common to find imbalanced datasets. A dataset is imbalanced when the dataset's class classification has a different proportional representation. For example, if we have more 'Yes' classes than 'No' classes labeled. Or in our context, whenever we have more labeled bug reports as not escaped defects than escaped defects.

Table 2 shows three types of imbalanced classifications. The “mild” appears when the proportion of the minority class is between 20% and 40%. The “moderate” appears when we have a 1% to 20% proportion. Moreover, the “extreme” appears when the amount of minority classes is minor than 1% [Developers 2022].

Table 2 – Imbalanced data classification

Imbalanced Class	**Proportion of Minority Class**
Mild	20% - 40% of the data
Moderate	1% - 20% of the data
Extreme	<1% of the data

An imbalanced dataset could lead to an imbalanced classification. So, if the data is not treated, the classification results in a biased outcome with the majority class in advantage. On the other hand, we do not want the model to ignore the minority class. Thus, the Synthetic Minority Over-sampling Technique (SMOTE), or other oversample solutions, comes to resample the proportion of the minority class.

Some oversampling techniques duplicate the data from the minority class, but this does not add meaning to the dataset. The SMOTE oversamples the minority class synthesizing the data. Unlike other approaches ([Japkowicz 2000] and [Ling and Li 1998] as cited by [Chawla et al. 2002]), the SMOTE creates an informative data sample and uses the k-nearest neighbor approach to select the samples that are close in space [Chawla et al. 2002].

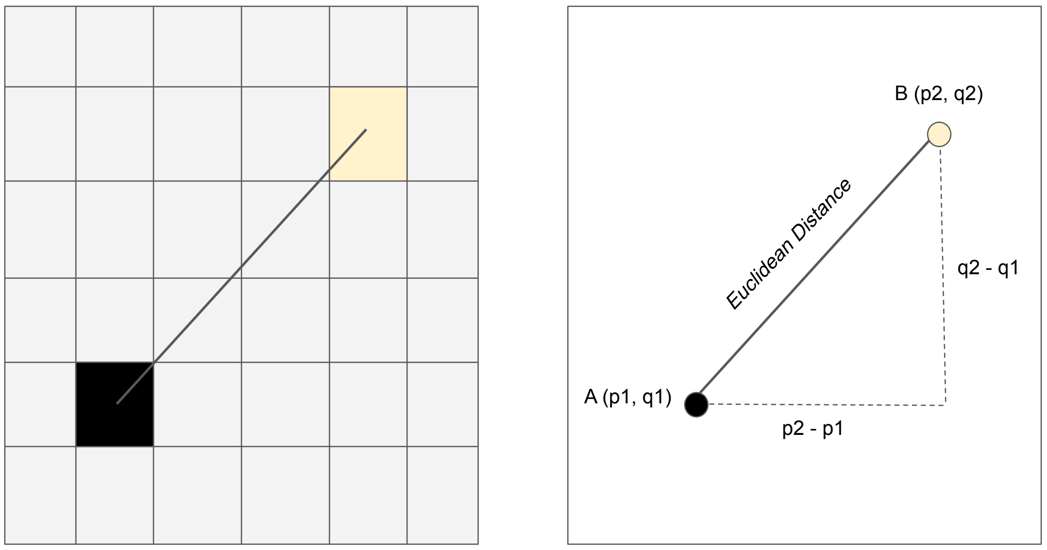
The Nearest Neighbor (NN) problem is described as follows: given a joint of data points and a query point in a metric space, find the data point closest to the query point [Beyer et al. 1999]. In the k-nearest neighbor, the k closest points is taken into consideration to create a synthetic data point.

The steps of k-nearest neighbor used inside the SMOTE approach are:

- Receipt of joint data. Considering there are a minority class and a majority class;
- Selection of a data point in joint data;
- Measurement the distance (Euclidean, Manhattan, Minkowski, Weight) between two points;
- Obtaining the shortest k-distance to the chosen point.

The Euclidean distance is one of the most widely used to calculate the k-nearest neighbor distance. It calculates a straight line distance between two points in space. The equation at the heart of this distance is the Pythagorean theorem, as we illustrate in Figure 4 [Clarkson 1999, Cover and Hart 1967].

Figure 4 – Euclidean Distance representation.



Source: Authorial, based on Euclidean distance representation.

There are different ways to describe the Euclidean distance equation. For example, considering the orthogonal triangle in Figure 4, we can describe the distance by the equation below. The equation computes the distance for points given by polar coordinates, where the polar coordinates of p are (r, θ) and the polar coordinates of q are (s, ψ). So, the law of cosines can give their distance.

$$d\left(p,q\right) = \sqrt{r^2 + s^2 - 2rs\cos(\theta - \psi)} \tag{2.1}$$

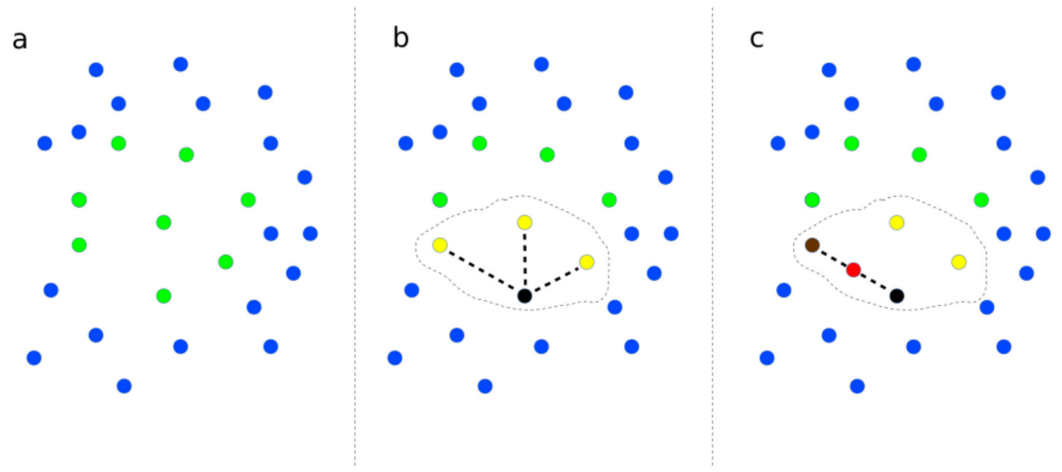
$$d\left(p,q\right) = \sqrt{\sum_{i=1}^{n}\left(q_i - p_i\right)^2} \tag{2.2}$$

Equation 2.2 represents a general format of the Euclidean distance between two points. For example, given a set S of points in space M and a query point $A \in M$, the distance represents the closest point in S to A.

In Figure 5a, we see a set of samples where the blue points represent the majority class, and the green point represents the minority class. Next, the SMOTE chooses a random sample from the minority class, and then the k-nearest neighbors are set, as shown in

Figure 5b. The black point is the selected random sample, and the three yellow points represent the 3-nearest neighbor. Finally, a synthetic sample is created between the first sample chosen (the black point) and one of its k-nearest neighbors (the brown point). In Figure 5c representation, the synthetic sample is the red point created between the black and brown points [Schubach et al. 2017].

Figure 5 – Graphical representation of the SMOTE algorithm.



Source: [Schubach et al. 2017].

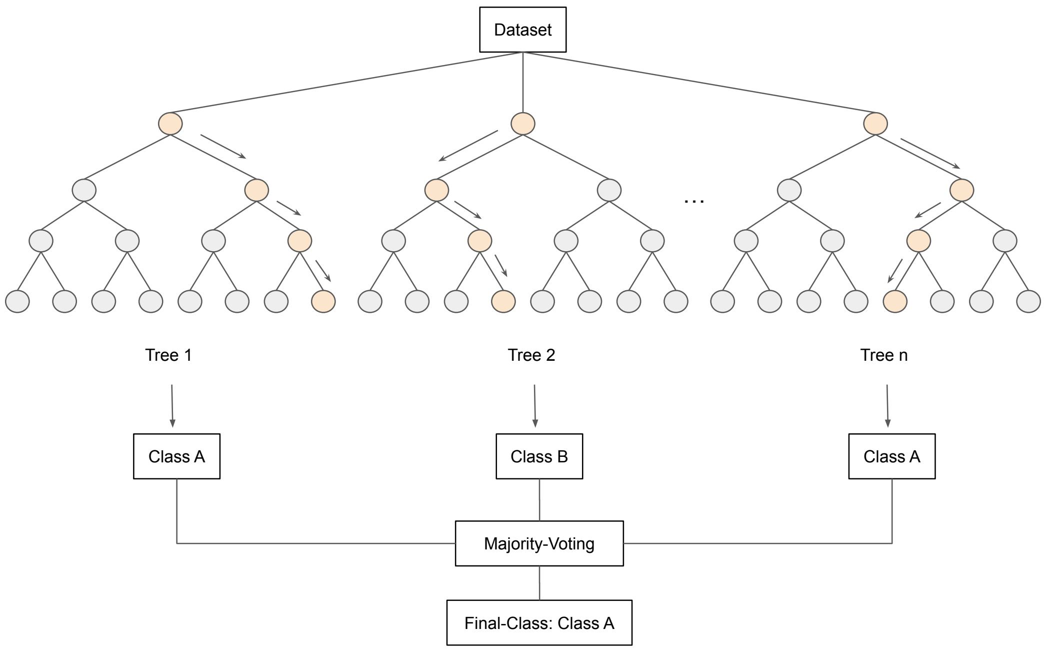
Finally, the procedure is repeated until the minority class has the same proportion as the majority class. Once we have the minority and majority classes with a similar number of samples, we can perform a more precisely classification model without concerns with the possible bias.

2.1.3 Random Forest

The Random Forest Algorithm is a Supervised Machine Learning Algorithm proposed by L. Breiman in 2001 and has been highly successful as a general-purpose classification and regression method. The approach produces and combines randomized decision trees using a randomly selected subset of training samples and variables. The classifier aggregates their predictions by majority vote for classification and average in case of regression [Cutler, Cutler and Stevens 2012]. The Random Forest has shown excellent performance in settings where the number of variables is much larger than the number of observations. Moreover, it is versatile enough to be applied to large-scale problems, is easily adapted to various ad hoc learning tasks, and returns measures of variable importance [Cutler, Cutler and Stevens 2012]. One of the most important features of the Random Forest Algorithm is that it can handle the data set containing categorical variables, as in the case of classification. Therefore, it performs better results for classification problems [Belgiu and Drăguţ 2016] [Biau and Scornet 2016].

Summarizing the idea, the Random Forest is an ensemble of decision trees; it combines several decision trees to produce better predictive performance than a single decision tree [Zhang and Ma 2012], as we can note in Figure 6. Moreover, we keep the same random order of the sample when executing the classifier another time. With that, even if we run the Random Forest more times, the result will not change slightly, considering the same sample.

Figure 6 – Simplified Random Forest Model.



Source: Authorial, based on Random Forest representation.

2.1.4 The ROC Curve

In Machine Learning, performance measurement is an essential task. The ROC curve is one of the most important evaluation metrics for checking any model's performance classification.

A receiver operating characteristic curve, or ROC curve, was originally developed for operators of military radar receivers starting in 1941, which led to its name. The method is a graphical plot illustrating the diagnostic ability used with a binary classifier system as its discrimination threshold is varies. In other words, ROC is a curve of probability distributions. It is very similar to the precision/recall curve, but the ROC curve plots the True Positive Rate (TPR) against the False Positive Rate (FPR) instead of plotting precision versus recall. The TPR is the positive instances correctly classified as positive, also called recall or sensitivity. The FPR is the ratio of negative instances incorrectly classified as positive [Narkhede 2018] [Géron 2017]. We can see the equations of these statements below.

$$Sensitivity = TruePositiveRate(TPR) = \frac{TP}{TP + FN} \tag{2.3}$$

$$Specificity = TrueNegativeRate(TNR) \tag{2.4}$$

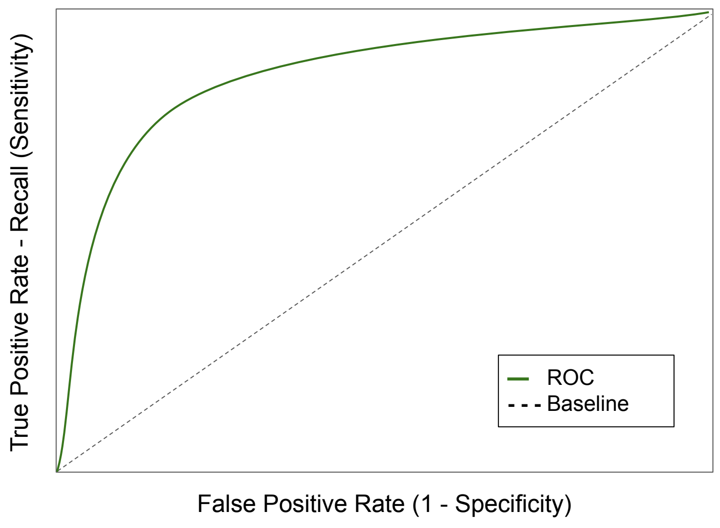
$$FalsePositiveRate(FPR) = 1 - Specificity = \frac{FP}{FP + TN} \tag{2.5}$$

The FPR equals 1 – TNR (True Negative Rate (TNR)). TNR is the ratio of negative instances correctly classified as negative. The TNR is also called specificity. Hence, the ROC curve plots sensitivity (recall) versus 1 – specificity, as we can visualize in Figure 7.

The ROC Curve approach is much used to:

- Compare classifiers, measuring the Area Under the Curve (AUC). The curves of different models can be compared directly or for different thresholds.
- Summary of the model skill using the AUC.

Figure 7 – General representation of an ROC Curve.

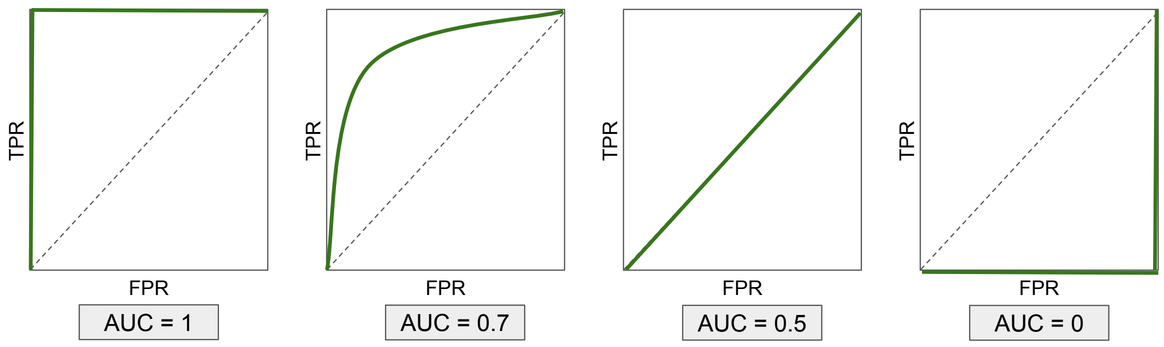


Source: Authorial, based on ROC curve representation.

ROC curves typically feature the True Positive Rate on the Y-axis and the False Positive Rate on the X-axis, meaning that the top left corner of the plot is the perfect point, as we can note in the first plot in Figure 8. An ideal classifier will have a ROC AUC equal to 1, whereas a purely random classifier will have a ROC AUC equal to 0.5.

It is desirable to maximize the True Positive Rate while minimizing the False Positive Rate [Narkhede 2018] [Géron 2017]. We can see this relation in Figure 8. The first plot represents the perfect ROC curve with the maximum performance possible. In the second plot, we have a representative value of 0.7 AUC. The third plot shows us a ROC curve of 0.5 AUC, which means a random classifier. Lastly, the fourth plot presents a ROC curve of 0 where all the probabilities are wrong.

Figure 8 – General representation of an ROC Curve.

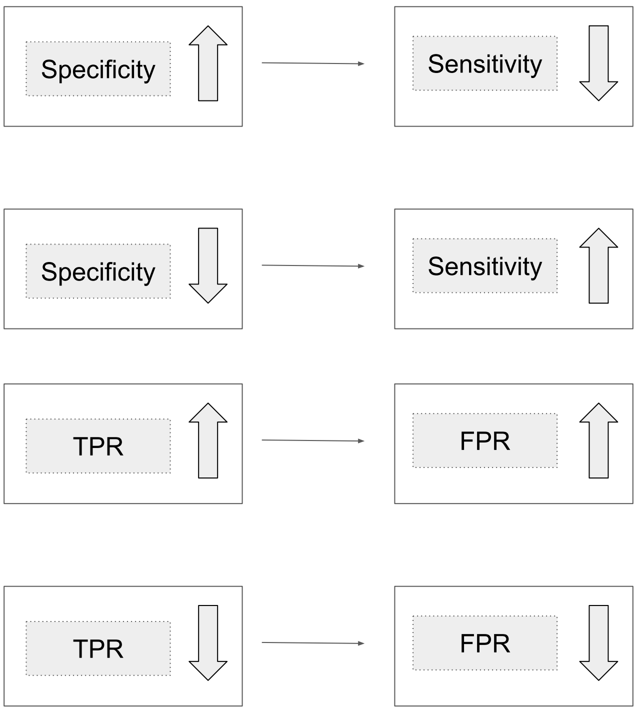


Source: Authorial, based on ROC curve representation.

2.1.4.1 The relation between Sensitivity and Specificity

Sensitivity and specificity are inversely proportional to each other. So when we increase Sensitivity, Specificity decreases, and vice versa. As FPR is equal to 1 - specificity, when we increase TPR, FPR also increases and vice versa. We can note this relation in Figure 9.

Figure 9 – Relationship measures.



Source: Authorial.

2.2 SOFTWARE ENGINEERING CONCEPTS

In this subsection, we describe the software engineering concepts necessary to understand the development of this work.

2.2.1 SDLC and STLC

Although they are similar acronyms, SDLC and STLC describe different strategies to help build softwares efficiently. STLC is focused on tests, enforcing a systematic method to meet quality standards. It can be performed as a series of steps within the SDLC cycle or alongside SDLC phases. In comparison, the SDLC considers all development stages. It is a process of building or maintaining software systems and includes various stages, from preliminary development to post-development software [Kale, Bandal and Chaudhari 2019, Franklin 2019].

A software application is intended to perform a set of tasks to provide results, involving computation and processing. It is necessary to manage the entire development process to ensure that the end product comprehends a high degree of integrity, robustness, and user acceptance. Therefore, a model that can emphasize the scope and complexity of the complete development process is vital to achieving a successful system. Thus, development teams often use the SDLC to build software systems in the software industry and development [Kale, Bandal and Chaudhari 2019, Franklin 2019].

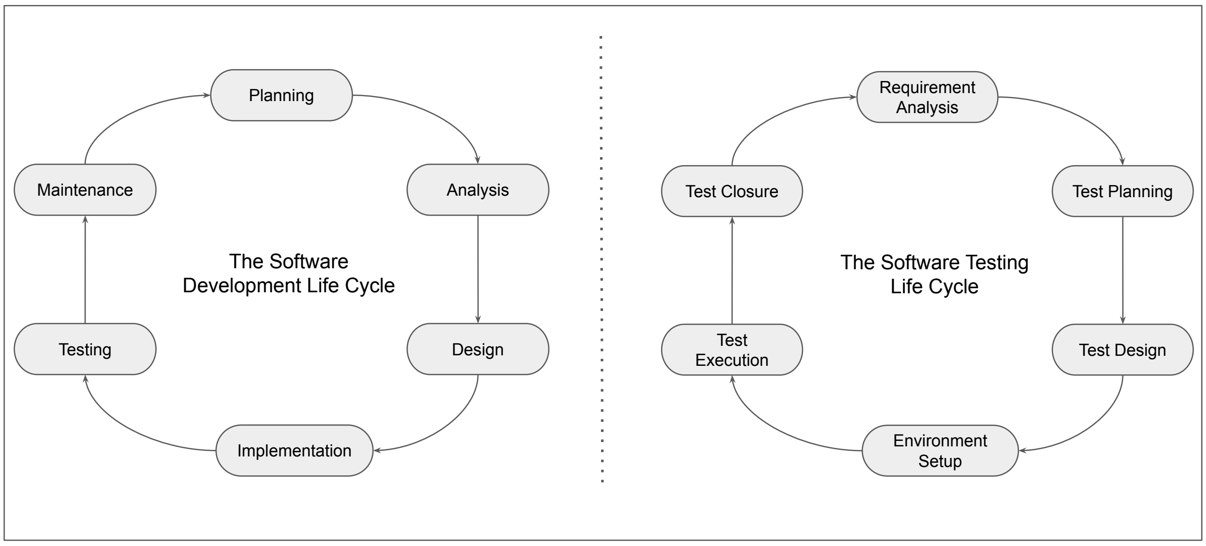
As we pointed out before, testing is one of the main activities in the Software Development Life Cycle. The SDLC details the critical phases in software development, as shown in Figure 10 (left side). In it, we have the following six phases [Tuteja, Dubey et al. 2012]:

- Planning: the first SDLC stage starts by defining the objective and scope of the system and the problems the future system needs to solve;
- Analysis: defines the Software Requirements Specification (SRS) document to create a technical solution with minimum risks as well as the quality assurance methodology;
- Design: this stage is responsible for creating the architecture of a software system and its elements. At the end of this stage, we have the Design Document Specification (DDS);
- Implementation: this phase is about building the software application using different tools and following a coding guideline defined by the organization;
- Testing: evaluates the quality of the software with the aim of finding and fixing defects;
- Maintenance: is responsible for updating and supporting the software after it has been delivered.

The STLC is also composed of six stages, but differently from the SDLC model, the STLC pursues all the activities involving the testing processes, going through all the testing steps. According to [Jamil et al. 2016], they are:

- Requirement Analysis: the phase that studies and analyzes the requirements from a testing perspective;
- Test Planning: determines the test plan strategy; it includes the cost estimates and efforts for the project;
- Test Design: is responsible for preparing test cases, test scripts (in the case of automated tests), and test data;
- Environment Setup: this setup is done based on the hardware and software requirements;
- Test Execution: starts executing the test cases based on the planned test cases. The test cases are updated with Pass or Fail along with the execution;
- Test Closure: it is the final stage where we prepare the test closure report and the test metrics.

Figure 10 – SDLC and STLC process.



Source: Authorial, based on [McCormick 2021] and [Jamil et al. 2016].

Therefore, looking at the SDLC and STLC steps, we can note different concerns in each approach. Both aim to reach the standards of quality but through different perspectives.

Testing is a significant part of the development process that allows us to reach quality standards. It is a process of verifying the aspects of the software to identify flaws and bugs and fix them. Therefore, testing is also an imperative step in the development process. In companies, profits depend on the customer to get the best system version which requires running exhaustive tests. However, thoroughly going through every potential risk and covering it with test cases can stand for a high cost and a long time. So, finding the right balance between time, quality, amount of tests, and a release date is important.

There are two ways of executing the tests, manually and automated, as described below:

- Manual Software Testing: In this type of testing, the Quality Assurance (QA) manually tests software solutions to find bugs. The QA team follows a plan to perform the manual tests; the plan details the test scenario and the interactions necessary to achieve the result. They record their findings and report to the team so they can fix all the possible issues [Rehkopf 2021];

- Automated Software Testing: Consists on the application of software tools to automate manual software testing. It ensures the quality of the system using a different kind of software that controls the execution of tests and compares the results [Rehkopf 2021].

Inside these two kinds of tests, many factors could be considered to create a strategy to build tests. So, various strategies exist to supply the system's operational activities.

In summary, tests consist in reviewing and validating a software product, and they can take anywhere from 20% to 40% of the total development timeline [Turpitka 2016]. The testing will take 20% of the development timeline for a small, single-component application. For more complicated systems with a Graphical User Interface (GUI), testing takes up to 40% of the total development timeline [Turpitka 2016]. Therefore, we need a systematic development and testing process to avoid extra costs while respecting the time and budget constraints of the project.

Once we have presented the types, methods, and testing strategies in the following subsection, we will detail aspects of escaped defects.

2.2.2 EDA

The EDA process is part of companies' activities. As we know, the software industry usually has a sector responsible for testing its software and services. In our context, there were different teams in charge of inspecting different kinds of defects. For example, Team A was responsible for examining Bug 1, while Team B inspected Bug 2.

After these inspections, report documents were generated. These documents are called bug reports and are created just when the defect is found. Through these documents, it is possible to analyze team and product aspects. When the bug is found for a different team – e.g. if Team A found Bug 2 or Team B found Bug 1 – it is called an escaped defect because the bug was found for a different team and we understand it was missed or escaped. Therefore, it is necessary to allocate an employee to analyze all the bug reports and label them as escaped defects or not.

Figure 11 describes the process of EDA analysis performed by the tester. We can understand the process in the following way:

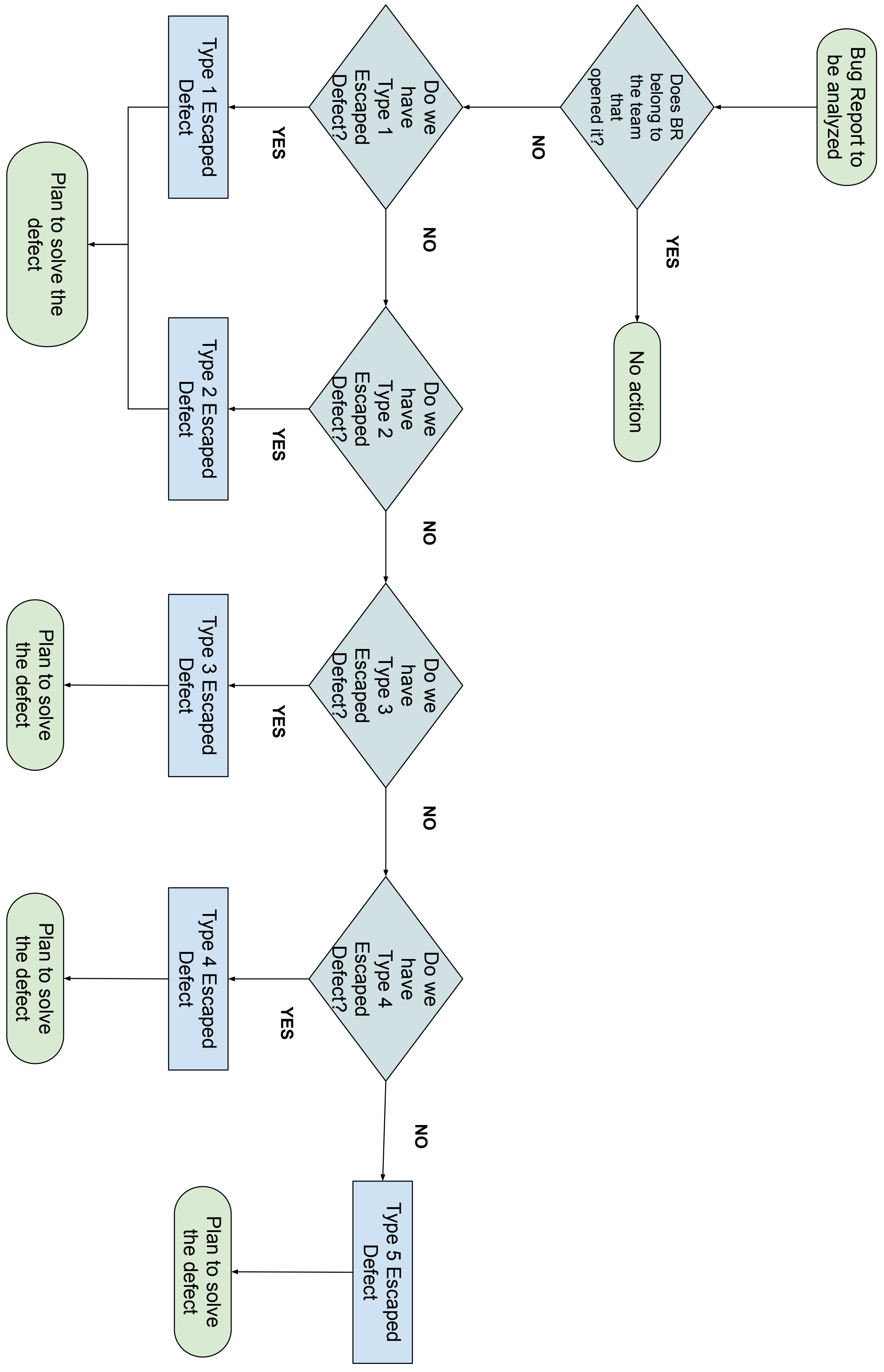
- First, we can see the flowchart starts with the bug reports to be analyzed;
- Then, the employee responsible for the escaped defect analysis checks if the same team who found the defect owns that bug report. If true, no action is taken, meaning the bug report is not an escaped defect and will be labeled as a not escaped defect. If false, we proceed to the next step;
- We check to which category the bug belongs. If the bug's characteristics contemplate type 1 escaped defect, it means "Type 1" is the cause of the escape. Therefore, we can say that test was missed by "type 1 escaped defect". The action to solve that bug can be related to the test case, test plan, or another component of the test process. Thus, the plan to solve the defect will be linked to one of them.

- Sometimes, the test pass for the first verification but gets stuck at the second. If this is the case, we can say the defect is missed by "Type 2". So, we have the "Type 2 escaped defect". The solution for this defect, as well the "Type 1", is related to some test step process. It can be to reorganize, update, or create a new test.
- If the tests pass through the first and second verification but fail at the third, the bug is characterized as missed by "Type 3", resulting in a "Type 3 escaped defect". And then, once the type is identified, the plan to solve the defect can be practiced.
- When the verification process fails in question 4, "Do we have Type 4 Escaped Defect?", we say the bug was missed by "Type 4". And then, we follow with the solution.
- Lastly, if we pass successfully through all verification, we conclude the test was missed by the only type remaining, "Type 5". Likewise the other, we start the process of defect resolution.

These are the step-by-step procedures to discover if the bug report is an escaped defect or not, the category the defect belongs to and also the action to take to solve the defect. When this action updates or creates a test case or plan, it usually takes about 10% to 15% of the overall time [Turpitka 2016]. So, it is important to have a precise result in the EDA analysis.

This work will identify and classify whether the bug is an escaped defect.

Figure 11 – Flowchart of Escaped Defects Analysis Process. Manual format.



Source: Authorial.

2.3 RELATED WORK

This section analyzes some works that have contributed to the escaped defect perspective, presenting a more detailed discussion. Besides that, we analyzed papers that have discussed the automation process in the bug report classification; for these, we present a brief explanation instead, they are not directly related to escaped defects; have discussed comparative research. As far as we know, there are not many papers directly related to escaped defect analysis. Follow below the works we have found and their connection with our work.

1 - [Herzig et al. 2015] - This work brought a solution to improve production costs and development agility. They proposed Theo, a generic test selection strategy. Theo builds a test selection based on the costs, and measurements, associated with each test. With it, test executions are reduced by 50%. To compute the cost, Theo considers costs associated with an escaped defect that involves people affected and the duration of time the defects remain undetected. Theo's approach is not directly related to escaped defects but uses them to do the cost calculations. The concern with this approach is that it skips tests that are not meant to be skipped.

   Connection: The solution of this work is focuses on creating a test plan so that the escaped defects do not occur. The proposed solution differs from ours because their idea is to select tests that already contemplate the error scenarios, thus, avoiding escaped defects. In comparison, we do not focus on generating a test selection but on analyzing bug reports and ranking them based on escaped defects.

2 - [Yaung and Raz 1994] - This work deals with the effectiveness of inspections based on escaped defects. They analyze escaped and non-escaped defects based on the source of the defect, the cost of inspection, and other information. With this, a linear discriminant function evaluates the effectiveness of each inspection. The result of this evaluation had findings on the inspection's effect, the time of preparation for inspection, and the artifact's size in the occurrence of escaped defects. Based on these results, the authors propose some interesting recommendations for the test team.

   Connection: This work differs from ours since its solution is focuses on analyzing of the inspection of escaped defects and their effects, bringing insights into the testing process in a conceptual approach. In contrast, our solution focuses on the automation of escaped defect analysis.

3 - [Eickelmann and Anant 2003] - This work discusses the Statistical Process Control (SPC), developed by Walter Shewhart in 1924, and how SPC should be applied to the software industry. It also raises points where that application fails and how to fix them. Furthermore, the work tells about the occurrence of escaped defects

when workers are given complex artifacts to inspect and have little time to prepare for inspection. Moreover, it mentions the importance of measuring these defects in the inspection process. Finally, this work focuses on applying SPC in the software industry.

Connection: This work presents relevant issues about the occurrence of the escaped defects and considers them in the testing process and measuring. The analysis of the escaped defects and the discussion raised connect with our work, but the focus of the work is not directly related to the escaped defect.

4 - [Vandermark 2003] - Lastly, this work aims to present the analysis of escaped defects and show the types of analysis, their metrics, and their benefits. It is a demonstrative work; no analysis or comparative study is done.

Connection: The connection with our work is limited to the escaped defects concepts, understanding, and definition.

In Figure 12, we summarize the main aspects of each work and compare them with ours. The last line of the table is related to this work, whilst the others are about the works mentioned above. We can observe that our work brings a tool to automate the escaped analysis process. The second work mentions the development of a tool, but it does not detail that, and this was not the focus of the work. Therefore, all the other works mentioned here are limited to the conceptual approach of escaped defect analysis, except the first work, which presents a tool to select tests.

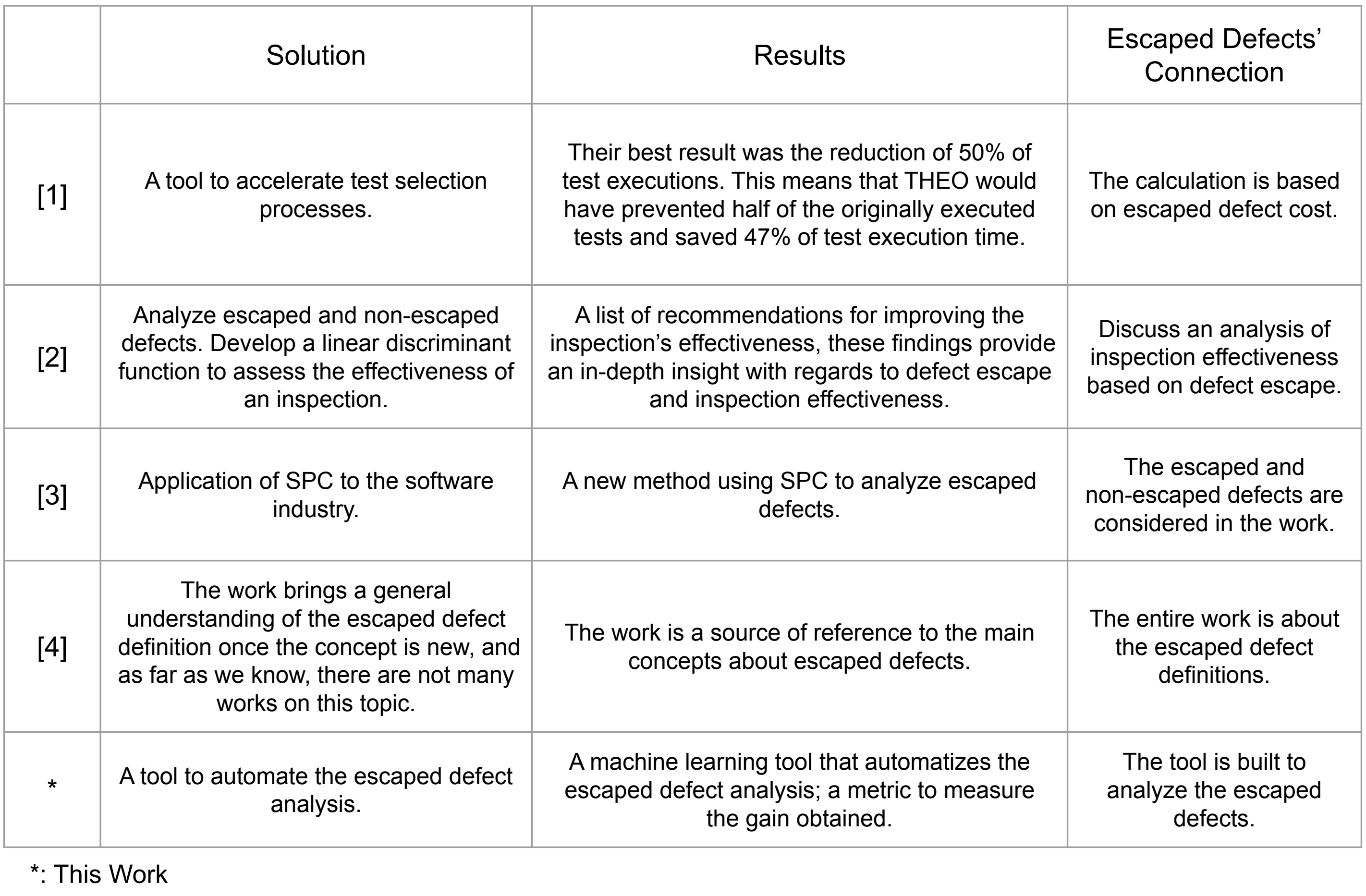
In addition to these approaches that directly involve the escaped defect issues, there are others. For example, there are works on classification and prediction in Bug Reports that keep up with state of the art. The related point with our work is in leading with the Bug Report. Processing, manipulating and treating them to improve software engineering processes in software companies. Here we will discuss some of these works.

In work [Otoom, Al-jdaeh and Hammad 2019], an SVM classifier is built to predict Bug Reports in two classes: the first is the corrective report (defect fixing), and the second is the perfective report (major maintenance). This predictor helps to quickly understand the Bug Report and allocate resources to each category. As a result, it had an average accuracy of 93.1% on three open-source projects.

In the works [Sabor, Hamdaqa and Hamou-Lhadj 2016, Tian et al. 2015, Kanwal and Maqbool 2012], the subject of a Bug Report's severity level, or priority level, is addressed. A classifier is built to identify and recommend the severity level automatically. This severity level is usually set manually. So, automating this process can prevent human error and make it faster.

Stack trace and categorical features are used in [Sabor, Hamdaqa and Hamou-Lhadj 2016]. The results show that when the stack trace and categorical features are combined, the KNN classifier's accuracy increases. The work shows that the accuracy of the severity

Figure 12 – Comparison table of related work with present work.



	Solution	Results	Escaped Defects' Connection
[1]	A tool to accelerate test selection processes.	Their best result was the reduction of 50% of test executions. This means that THEO would have prevented half of the originally executed tests and saved 47% of test execution time.	The calculation is based on escaped defect cost.
[2]	Analyze escaped and non-escaped defects. Develop a linear discriminant function to assess the effectiveness of an inspection.	A list of recommendations for improving the inspection's effectiveness, these findings provide an in-depth insight with regards to defect escape and inspection effectiveness.	Discuss an analysis of inspection effectiveness based on defect escape.
[3]	Application of SPC to the software industry.	A new method using SPC to analyze escaped defects.	The escaped and non-escaped defects are considered in the work.
[4]	The work brings a general understanding of the escaped defect definition once the concept is new, and as far as we know, there are not many works on this topic.	The work is a source of reference to the main concepts about escaped defects.	The entire work is about the escaped defect definitions.
*	A tool to automate the escaped defect analysis.	A machine learning tool that automatizes the escaped defect analysis; a metric to measure the gain obtained.	The tool is built to analyze the escaped defects.

*: This Work

Source: Authorial

prediction approach can be improved from 5% to 20% by considering categorical features. In [Tian et al. 2015], it outperformed baseline approaches in terms of average F-measure by a relative improvement of up to 209%. The classifier used Naive Bayes and SVM in [Kanwal and Maqbool 2012]. [Kanwal and Maqbool 2012] also analyzed the features that best define the degree of severity of a bug. The work used precision, recall, and two others metrics created by the authors to validate the results.

As mentioned earlier, none of these works directly addresses classification with an escaped defect. Still, they address the Bug Report in other ways and create models to solve different failures involving the Bug Report. In the case of the latest three works presented, they address the severity level of regular Bug Reports, which helps the tester to choose a Bug Report more carefully; however, they do not focus on the escaped bug, which is the objective of this work. In other words, we can say they score the bug reports centered on a general approach, and we score the escaped bugs.

2.4 FINAL CONSIDERATIONS

In this chapter, we presented fundamental concepts to understand this work's development, such as Confusion Matrix, Smote, Random Forest, ROC Curve, SDLC, STLC and EDA. These concepts are related to machine learning and software engineering fields, and all of them were important to building this work. Many machine learning concepts were omitted here; however, they were presented in the other chapter, along with the working flow and discussion. We also introduced the related work, explaining and comparing relevant points raised by them. The main difference between the works is that our work presents a practical approach to the escaped defect analysis issue while the others bring a conceptual approach. In addition, we developed a tool to automatize the escaped analysis process; this tool provides a ROC Curve of 88% and a gain of 90% to 60% of escaped defects in our best result.

3 PROPOSED SOLUTION

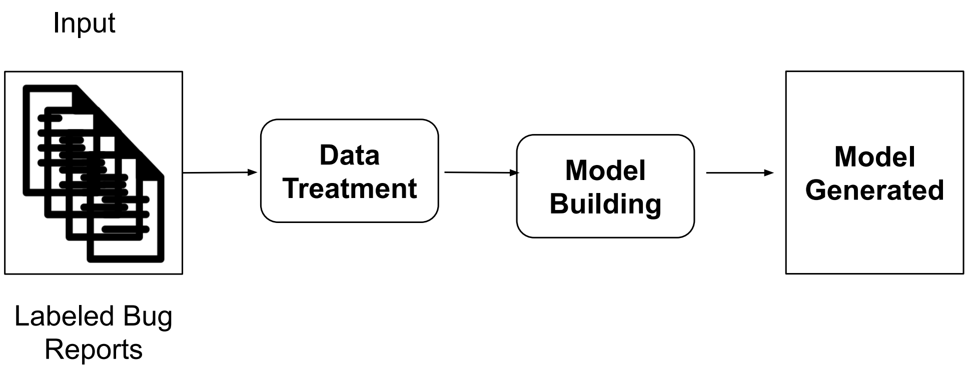
This work considered real data in the software test industry in partnership with Motorola Mobility. The company provided the dataset resulting from escaped defects analysis. The tester analyzed each Bug Report and labeled them. With that, we had our historical data. In our context, the historical data is the labeled dataset with manual EDA analysis previously executed for the company. This dataset was used as input to our system to recognize the pattern and make the model learn.

In this work, we propose a system based on Machine Learning to find escaped defects automatically. This system consists of an Artificial Intelligence (AI) model that uses Random Forest to predict the probability of instances being an escaped defect or not. Each instance represents a Bug Report. So, the model considers some Bug Report documents as instances and the bug report fields as attributes. In the end, we intend the model to build a ranking with the probability of a Bug Report being an escaped defect or not. The idea is that the tester uses this ranking to predict the class of a Bug Report and gain time and performance in the EDA analysis.

3.1 PROPOSED ARCHITECTURE

This section will go through each flowchart module to explain how we proceeded with the proposed architecture. Figure 13 and 14 present our solution in a flowchart in more detail. Figure 13 shows us the process of building the model, while Figure 14 presents the components of the model usage process. Both flowchart components represent a module we implemented to build our machine learning system.

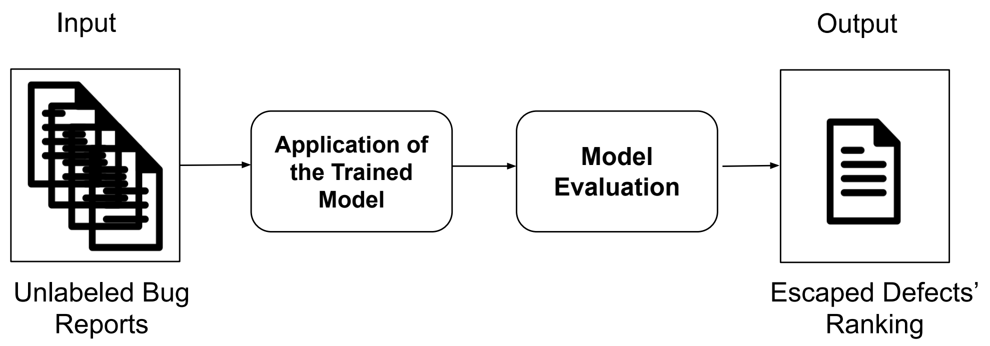
Figure 13 – Flowchart of main components to build the solution.



Source: Authorial.

The labeled Bug Reports were the input of our system. They included all the labeled instances. In our context, the instances were Bug Reports. The labeled bug reports with

Figure 14 – Flowchart of main components of model usage.



Source: Authorial.

1 represented the escaped defects; label 0 represented not escaped defects. Table 3 shows a representation of the input spreadsheet.

Table 3 – Representation of the input data.

	Attribute 1	**Attribute 2**	**Label**
Bug Report 1	...	...	0
Bug Report 2	...	...	1

Data Treatment was the second step and part of the pre-processing; it is mainly about data formatting. This process is responsible for eliminating the noise in the dataset and preparing the attributes to be used in the model's input. Once we had treated the dataset, we divided it between training and testing, and the data was ready for the training phase.

The next step was Model Building, where we chose the model, set the parameters, and then trained the classifier with the dataset. At the end of this process, we obtained the learned model. Thus, once the model is trained, it is possible to evaluate it, and this step (the Model Evaluation process) can predict the new instances of a Bug Report based on the learning.

Lastly, the system returns the output corresponding to the ranking of escaped defects. The instances with a significant probability of escaping defects will be found at the top of the results. More details will be presented in the results chapter.

As we said before, the objective of this work is to automate the process of EDA analysis. For that, we received the dataset related to two company teams. Here, we will call them Team A and Team B. The process consisted of Team A and Team B employees, with a tester function at the company, analyzing all escaped defects manually. So they took the Bug Report documents, and based on the information presented, the Bug Report was classified as an escaped defect or not, as shown in Figure 2. Then, the escaped defect figures were computed in the product and team performance.

The Bug Report was created at the previous step while the tester was performing the test on the mobile device. The tester documented any bug as a Bug Report whenever it was found. So, the Figure 2 previous process will go through each step in Figures 13 and 14 to be automatized and return the ranking. In the following section, we will explain in detail how these modules work and how they were implemented.

3.2 IMPLEMENTATION

This subsection describes the implementations of the modules seen in subsection 3.1 required to construct our machine learning system. The following subsections are divided into the topics: Exploratory Data Analysis, Receiving Labeled Bug Reports, Data Treatment, Model Building, Model Evaluation, and the generated rank in the system's output.

3.2.1 Exploratory Data Analysis

Before implementing the modules, we investigate and analyzed the features, as will be detailed in Chapter 5. The items below describe each of the techniques used.

- Missing data search – visualize the features with missed information.
- Histogram/dispersion of the main attributes – distribution of the features.
- Heat matrix – relationships among the features.
- Word cloud – frequency of the main features and categories in a visual format.
- Relation: Resolution x Label – distribution of the Resolution feature according to the labels' figures.
- Relation: Team Found x Label – distribution of the Team Found feature according to the labels' figures.
- Relation: Product Affected x Label – distribution of the Product Affected feature according to the labels' figures.
- Relation: Components x Label – distribution of the Components feature according to the labels' figures.

3.2.2 Receiving Labeled Bug Reports

Initially, the company provided the raw data, with the information of a lot of labeled Bug Reports. Then, employees manually analyzed these bugs to be used internally (this process is better described in Chapter 2). Therefore, the model was trained to automate this task. It is important to note that one of the steps of this process was meant to transform the raw data into a spreadsheet (CSV format), allowing for better manipulation.

3.2.3 Data Treatment

Before proceeding to the model training, it is necessary to understand, clean, and format the data. The format is fundamental for the classifier to recognize the data. So, one crucial part is converting the textual, categorical, and multi-categorical data to the numerical pattern, allowing the classifier to operate appropriately. Thus, this subsection will describe converting the data to a readable classifier format with a binary pattern.

First, we separated the data from Team A and Team B in two different spreadsheets, and then, we made a filter to extract only the instances (bug reports) with the marked label. The instances representing the escaped bug were marked as one of the following types, as explained in Chapter 2:

- Type 1 Escaped Defect
- Type 2 Escaped Defect
- Type 3 Escaped Defect
- Type 4 Escaped Defect
- Type 5 Escaped Defect

These are the types of escaped defects. Next, we assigned 1 to the escaped defects with one of the types above. We assigned 0 to the label referring to the Bug Report considered a not escaped defect, without the types above. Finally, we eliminated the Bug Report with no labels.

After that, we removed noises from the new dataset, so we had to deal with blank fields, missing values, invalid attributes, special characters, and other noises. We did it using tools in the python libraries and the Regular Expression (RegEx) technique, which is proper for creating patterns to manage text [Câmpeanu, Salomaa and Yu 2002].

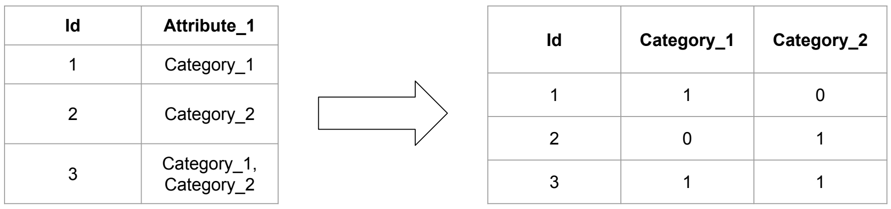
Next, it was necessary to deal with the nature of the attributes, which are inherently categorical, multi-categorical, and textual. For this, we used the approaches described below.

3.2.3.1 Categorical and Multi-Categorical Attributes

The dataset used is composed of, among others, multi-categorical and categorical/nominal attributes (more details about what these attributes are in Chapter 4). In Figure 15, specifically in the table on the left, we can see examples of each attribute type. They represent discrete values that belong to a set of categories or classes, but there is no intrinsic ordering to the categories [Leeper 2000].

For categorical and multi-categorical attributes, we used the MultiLabel Binarizer (MLB) method from Sklearn to encode the features [Pedregosa et al. 2011]. It is responsible for transforming each category into a new column in the dataset. Figure 15 shows an example of how this transformation occurs, where the category_1 and category_2 (left figure side) were transformed into two new attributes filled with 0's and 1's to represent when that category is present or not (on the right side of the figure).

Figure 15 – Dataset before and after MLB technique application.



Id	Attribute_1
1	Category_1
2	Category_2
3	Category_1, Category_2

Id	Category_1	Category_2
1	1	0
2	0	1
3	1	1

Source: Authorial.

Figure 15 also represents a multi-categorical spreadsheet (Id 3). There, we have two different categories in the same instance, so we can say that attribute_1 is multi-categorical. On the other hand, when the attribute is categorical, each instance has just one category (Id 1 and 2). At the end, we had a new dataset with the categorical and multi-categorical attributes replaced by binary patterns, as shown in Figure 15 (right side). Note that there are other approaches like MLB. In our case, MLB fits due to leading with more than one category.

3.2.3.2 Textual Attributes

As mentioned before, an important point was converting the textual attributes to a numerical pattern (In our context, the 'Summary' is our text feature). For textual attributes, after separating them from the other features, the first step was to apply tokenization to the sentence. Tokenization means dividing up the input text, which to a computer is just one long string of characters, into subunits called tokens. The technique identifies each atomic unit and represents the first operation performed in document processing [Grefenstette 1999, Habert et al. 1998].

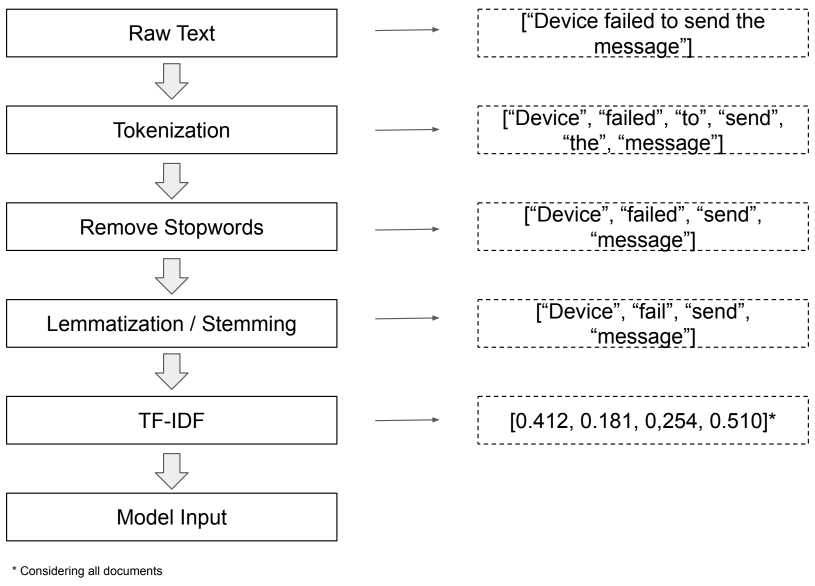
After that, we eliminated the English stop words existing in each instance. Also known as noise words, they contain little information, which is not usually required [Kaur and Buttar 2018].

Next, we normalized the sentences with lemmatization and stemming techniques. Stemming is a procedure to reduce all words with the same stem to a standard form, whereas lemmatization removes inflectional endings and returns the base or dictionary form of a word; that is, it finds the normalized form of a word [Plisson et al. 2004, Balakrishnan and Lloyd-Yemoh 2014]. Both techniques are responsible for reducing the word to its respective root word [Balakrishnan and Lloyd-Yemoh 2014].

Lastly, we applied the Term Frequency Inverse Document Frequency (TF-IDF) Vectorizer to transform the text into a vector. It allows transforming text to feature vectors and analyzes how relevant a term is in a document. All these words are stored in a vocabulary of known words. Then, it calculates how often a word appears and the inverse document frequencies, penalizing the words that are very similar and encoding the documents [Pedregosa et al. 2011]. Finally, we used the generated vector as one of the model's inputs (with the other attributes).

With that, we had our text attribute converted to a classifier's readable format and could use it, along with the other features, as the model input. Below, we can visualize the data treatment pipeline in Figure 16. Starting with the raw text, we went through all the steps mentioned in this section until we obtained the text in the appropriate format. Finally, we finished the data cleaning and treatment step and loaded the dataset into another script responsible for building the model.

Figure 16 – Text Cleaning Pipeline.


Raw Text
["Device failed to send the message"]
Tokenization
["Device", "failed", "to", "send", "the", "message"]
Remove Stopwords
["Device", "failed", "send", "message"]
Lemmatization / Stemming
["Device", "fail", "send", "message"]
TF-IDF
[0.412, 0.181, 0,254, 0.510]*
Model Input
* Considering all documents

Source: Authorial.

3.2.4 Model Building

Now, our dataset is ready to be used as input for the model. However, a relevant issue identified was that data had a significant degree of imbalance between the distribution of the two classes. Accordingly, before dividing the data into training and testing, we split its instances into three sets, each with different balances, as detailed in the following paragraphs. The objective was to know how much a possible data imbalance influenced the model's performance.

Many real-world classification problems have an imbalanced class distribution, such as risk management, anomaly detection, and fraud detection [He and Garcia 2009]. Not different, our original dataset had many more not escaped defects (label 0) than escaped (label 1) labeled instances. Therefore, its condition could bring a bias in the model's answer. To avoid misunderstandings in the data representation, we chose to balance the data.

In the division of the three sets, the first set applied the undersampling technique; the second used the oversampling technique; the third used all the data. Undersampling and oversampling are popular methods in dealing with class-imbalance problems to reduce the skew in class distributions. Undersampling uses only a subset of the majority class, while oversampling uses only a subset of the minority class [Liu, Wu and Zhou 2008, Pozzolo et al. 2015, Mullick, Datta and Das 2019].

Once we had tested and chosen the sampling approach, we followed the classifier to run the model. We used the Random Forest as our principal classifier and implemented its solution to make our model train with the dataset. The choice of classifier took into account preliminary tests carried out on small samples that demonstrated that Random Forest is a good option to be considered in our context.

3.2.5 Model Evaluation

The last implementation step consisted in the model validation. The following section will discuss the details of the experiments in this step.

In Table 4, we visualize the distribution of test data to both teams. Similar to the training distribution, the data test had more instances of not escaping defect (label 0) than escaped (label 1) for both teams. Besides, Team A had more instances and Bug Reports compared with Team B.

Table 4 – Distribution of testing data.

	Team A	**Team B**
Label 0	1864	74
Label 1	134	42

After predicting new instances of the Bug Report, we implemented the metrics to measure the model's output and understand its behavior. We used the language tools and libraries to capture the following metrics:

- False Positive (FP) Rate;
- True Positive (TP) Rate;
- False Negative (FN) Rate;
- True Negative (TN) Rate;
- Receiver Operating Characteristic Curve (ROC);

- Cost-Benefit Curve.

Among the metrics mentioned above, the last one is a new metric proposed in this work (Cost-Benefit Curve). Its purpose is to bring a different perspective to the analysis of the classifier's performance. More details on how it works will be discussed in Section 3.3.

3.2.6 System's Output

In the end, we obtained the generated rank. This step did not require any implementation. We captured the results and analyzed them as described in Chapter 5.

3.3 THE COST-BENEFIT CURVE

The Cost-Benefit Curve presents the benefits while using the ranking generated to realize the Escaped Defect Analysis. It shows us in the Y-axis the Escaped Defect Coverage Rate (EDCR); in an overall context, it is the positive instance correctly classified as positive or, in other words, the True Positive Rate. On the other hand, the X-axis presents the Productive Efficiency Rate (PER); in the general context, it is the rate of negative instances: the True Negative plus the false-negative instances divided by all instances.

From the EDA's perspective, the tester's analysis will not consider the instances classified as negative (the True Negative and False Negative). In this case, the idea is to focus on the escaped defects. So, the EDA analysis time will not consider the negative instances that represent the labeled Bug Reports as not escaped defects. Therefore, the negative rate instances represent the productive efficiency rate; when we decrease the negative rate instances, the productive efficiency rate decreases. According to the model, the PER indicates the Bug Reports that will not be necessary for analysis because they already are not escaped defects.

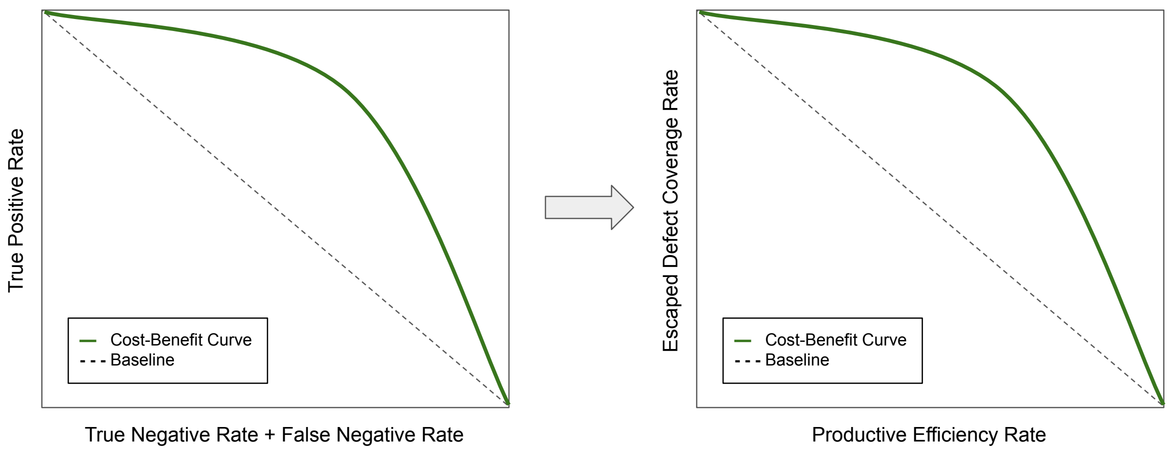
In Figure 17, we see the graphic of the Cost-Benefit Curve. On the left side, we have the graphic with the equations for the Y-axis and X-axis, while on the right side, we have the graph that presents the evaluation results.

Below are the equations that model the behavior of the escaped defect rate and productive efficiency rate. The first equation presents the escaped defect coverage rate metric. The second equation shows the productive efficiency rate metric.

$$EDCR = \frac{TP}{TP + FN} \tag{3.1}$$

$$PER = \frac{FN + TN}{TN + FP + FN + TP} \tag{3.2}$$

Figure 17 – Cost-Benefit graphic representation.



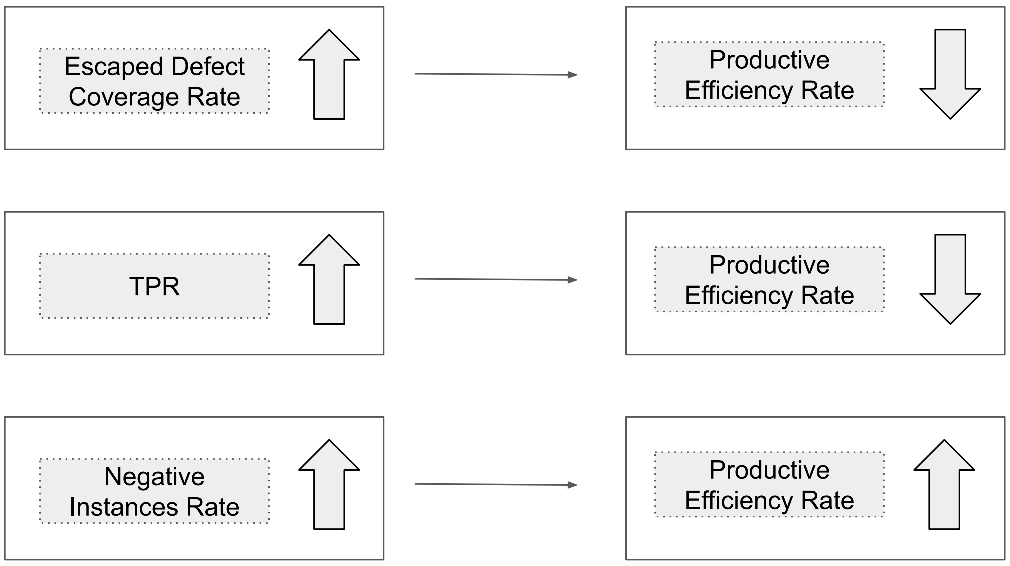
Source: Authorial.

3.3.1 The relation between the Escaped Defect Coverage Rate and the Productive Efficiency Rate

The escaped defect coverage rate is inversely proportional to the productive efficiency rate; when we increase the productive efficiency rate, the escaped defect coverage rate decreases. As the escaped defect rate is equivalent to the True Positive Rate, the TPR is also inversely proportional to the productive efficiency rate. Moreover, we can note through the productive efficiency rate formula that if we increase the negative instances rate, we also increase the productive efficiency rate. We see these relationships in Figure 18.

In order to have a higher productive efficiency rate, we should take a point in the graph that contemplates a considerable value to the escaped defect rate. Then, we could balance both rates.

Figure 18 – Cost-Benefit Curve Relations.


Escaped Defect Coverage Rate
Productive Efficiency Rate
TPR
Productive Efficiency Rate
Negative Instances Rate
Productive Efficiency Rate

Source: Authorial.

3.4 FINAL CONSIDERATIONS

This chapter described all the steps and approaches we used to implement the machine learning system. These tools and processes were chosen due to the data's nature and conditions, flexibility, and other characteristics. They were the results of investigation and learning. After applying all the processes described here, we had our model ready to be used in the proposed context.

4 EXPERIMENTS

This chapter describes the setup and scenario of the executed experiments and the specificities of each experiment. It is divided into:

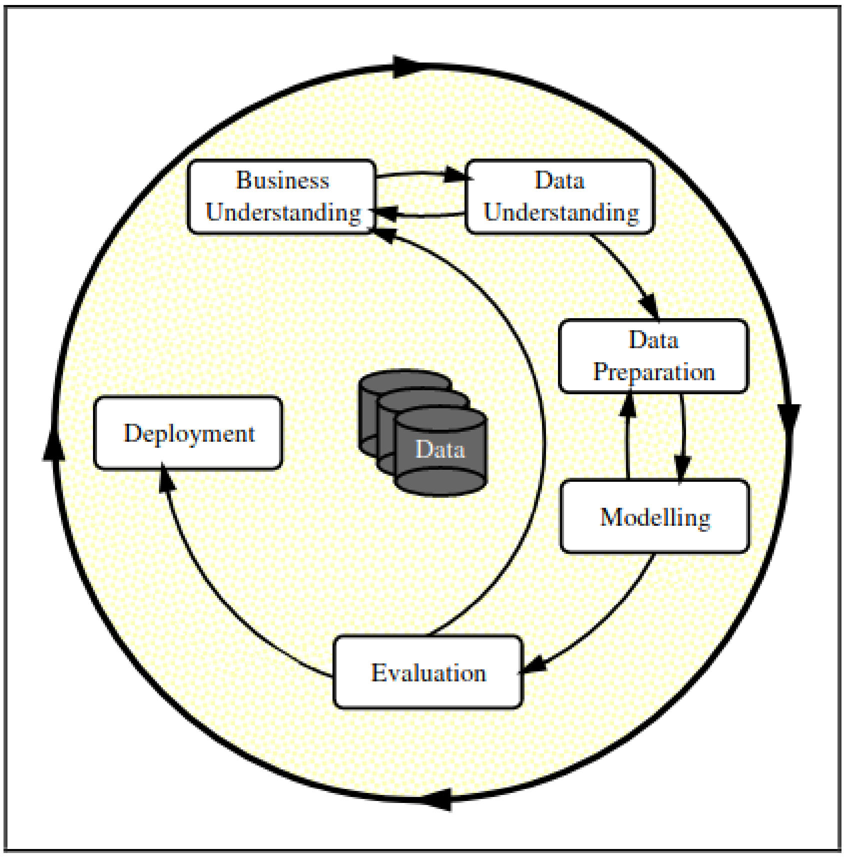
- Dataset - Where we explain the dataset, its structure and characteristics;
- Methodology of Experiments - Where we show the construction of the executed experiments;
- Experiment Team A - Experiments using Team A's dataset and its use in the experiment's setup;
- Experiment Team B - Experiments using the dataset of Team B and how it was used in the experiment setup;
- Final Considerations - The chapter's overview and goals.

We present the experiments and results following the Cross Industry Standard Process for Data Mining (CRISP-DM) scientific method. This method address the data science process. It brings a methodology focused on translating business problems into data mining tasks [Wirth and Hipp 2000]. Its steps consist in:

- Business understanding - Understanding the project's objectives and requirements;
- Data understanding - Investigating and getting familiar with the data;
- Data Preparation - Modelling the dataset to construct the data to be used in the system;
- Modelling - Selecting and applying the modeling technique;
- Evaluation - Evaluating the model to be sure it properly achieves the business' objective;
- Deployment - Releasing and reporting the system.

The CRISP-DM method provides an objective, standardized approach to conducting experiments and improves results using a data science perspective. Figure 19 shows the steps described above.

Figure 19 – Phases of the CRISP-DM Process.



Source: [Wirth and Hipp 2000].

4.1 DATASET

In this work, we conducted experiments with real industry datasets. We used two joints of data. The first dataset referred to Team A, and the second dataset was about Team B. It means that the first dataset was related to the defects that Team A should found but escaped from the team. The second dataset was about the escaped defects of Team B. Both data joints were collected under the same EDA manual analysis process, but the instance of Team A and Team B differed. More specifically, Team A's data was four times larger than Team B's. It means that Team A had more Bug Reports documented and, consequently, the input of Team A's experiments was larger compared with Team B's experiments. However, the datasets contained the same attributes.

Table 5 shows the attributes present in the dataset as well as their characteristics. Attributes with no aggregate value were removed and all of them are the initial features before applying any treatment and the encoding technique, which increases the number of the features, as explained in Chapter 3.

The attributes shared the characteristic of being categorical, multi-categorical, and textual. Below we present a description of each one according to the company context:

Table 5 – Attributes of dataset.

Attributes	**Attribute Characteristic**
Team Found	Categorical
Components	Categorical
Resolution	Categorical
Tags	Multi-categorical
Product Affected	Categorical
Summary	Textual

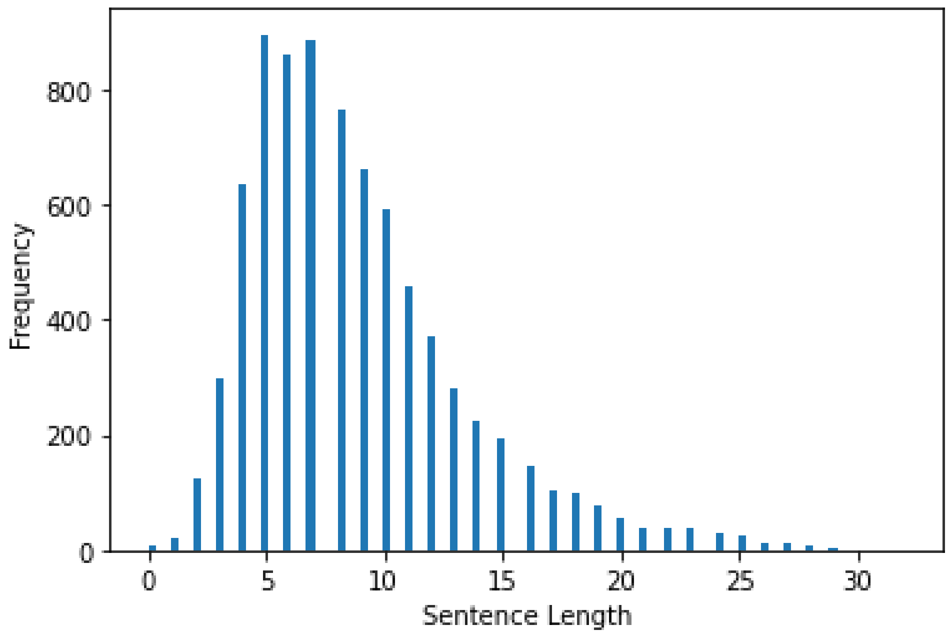
- Team Found - The name of the team that found the bug.
- Components - All the device's components are involved in the failure. It could be any additional hardware equipment;
- Resolution - The current Bug Report status. It could be "Fixed", "Requirement Updates", or other;
- Tags - All tags related to the failure. It could be any subject or object involved with the defect;
- Product Affected - The code to the device model. It helps to identify the issue straight away;
- Summary - The title of the Bug Report. The title is the first thing to be analyzed in the document and summarizes the issue.

These attributes are information related to the Bug Report. Therefore, they were considered along with the manual analysis and the automated model. Besides, the company provided the historical dataset, collected manually in the traditional process of escaped defect analysis.

In the beginning, we had a dataset, as shown in Table 5. After applying the transformation and encoding process to deal with the categorical, multi-categorical, and textual features, the dataset became binary to the categorical and multi-categorical values (see Figure 15). And about the textual treatment, we got the vectorized format for the textual feature, as seen in the last phase of Figure 16.

Looking straightforward at the "Summary" textual feature in Figure 20, we see the sentence length histogram, which is the number of tokens by sentence. The major part of the Bug Report titles presents between five and seven tokens. Figure 21 shows the frequency these tokens appear, considering the whole document. The "Summary" field is an important attribute. After its analysis and treatment, it is turned into a vector we can use inside the model.

Figure 20 – Sentence length histogram for Team A.



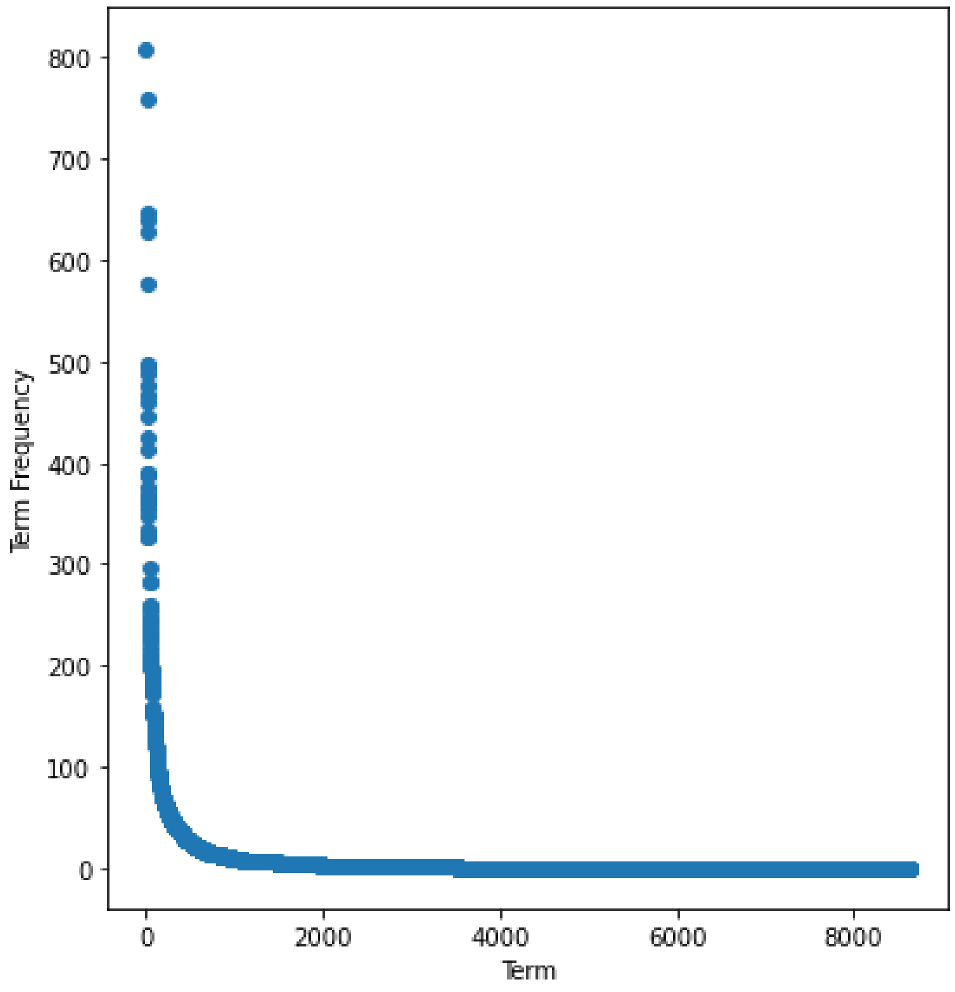
Source: Authorial.

Our dataset changes as long as we apply a new transformation to it. Ultimately, we have a dataset with a different dimension resulting from the methods applied. This last version of the data treated works as the input to our system, and then we can start the experiments. More characteristics of the dataset can be seen in Table 6.

Table 6 – Dataset characteristics.

Dataset Characteristics	Univariate
Attribute Characteristics	Categorical, Multi-Categorical, and Textual
Associated Tasks	Classification / Ranking
Number of Attributes	6 (Initially)
Missing Values	Yes

Figure 21 – Graphic of term frequency.



Source: Authorial.

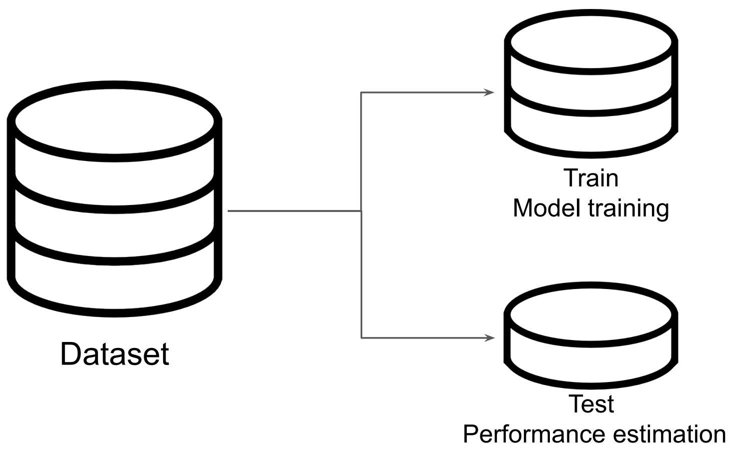
4.2 METHODOLOGY OF EXPERIMENTS

The methodology for Team A's and Team B's experiments were similar but not precisely the same due to the variety of the datasets. Both datasets contained equivalent features; however, they were structured differently. So we created different scripts to deal with them. This section describes the similarity in the approaches to the experiments. Subsequent sections will present more specific aspects of each team.

To understand the data, we searched for strategies to lead with them. First, we explored works leading with the same data type. And then, we applied different techniques until one implementation functioned, as described in the chapter on the Proposed Solution. As we had a large data joint, we used the Holdout approach [Lorena et al. 2021]. We divided the dataset into a proportion of p to train and $(1-p)$ to test [Michie, Spiegelhalter and Taylor 1994]], as shown in Figure 22. One part of the data was directed to the model training, and the other to the performance estimation. The division percentage varies depending on the experiments, but usually, the major part is to train and the remainder to test. In the following section, we specify the holdout division percentage values for Team A and Team B.

The Holdout approach could sub-estimate the hit rate once a predictor produced on all objects generally has a higher hit rate than that generated from only some of them [Baranauskas and Monard 2000]. However, as the authors present in [Lorena et al. 2021] and [Michie, Spiegelhalter and Taylor 1994], we do not have to worry about it when leading with a large data joint.

Figure 22 – Holdout division.



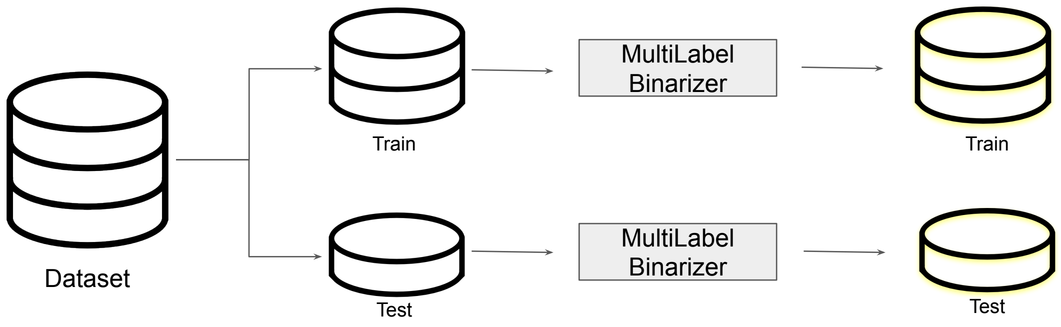
Source: Authorial.

After this division, we applied the categorical and multi-categorical techniques separately (MLB), as we shown in Figure 23. First, we divided the dataset into training and testing parts using the holdout technique, and then we encoded the data by applying MLB to obtain the treated data.

We built it this way not to merge the training and testing data as long as we led with the categories. In the real industry data, the dataset arriving to be analyzed could not contain some category present in the trained model. For instance, we had a model already trained with the training data, and the dataset just-arrived to be analyzed. The train data had the attribute "x" with a total of three categories, but the testing dataset had the attribute "x" with two categories, and the third category was not present. So, we had to be prepared to lead with different dimensions in this situation. In this case, we did not consider the third category present in the arrived dataset until the subsequent training phase.

It is important to evaluate the model to cover the scenario where we do not know all the categories present in the testing dataset. Considering this scenario, the treatment of categorical and multi-categorical values limits the transformation based on the training data. It means that the encoding process will be applied to both, training and testing data, but if any value appears in the testing and not in the training data, we do not consider these values until the following training phase. With that, we avoid the problem of different columns in the training and testing data or dimension errors and contemplate the real industry scenario. So, we built our experiments, as shown in Figure 23.

Figure 23 – MLB procedure to training and testing dataset.



Source: Authorial.

After the MLB process, we evaluated the sampling with three versions of different divisions. As mentioned in the previous chapter, the idea with these divisions was to understand how dataset imbalance could affect model's performance. In the first and second versions, we used a technique known as undersampling and oversampling, respectively, to balance the dataset. The third version correspond to the original data, with a significant imbalance.

More specifically, in the undersampling technique, we randomly decreased the instances of not escaped defects (labeled as 0). Then, we dropped these instances until we had a balanced dataset. On the other hand, in the oversampling, we applied the SMOTE. This method recreates new synthetic data based on existing data [Chawla et al. 2002]. In our context, we applied the SMOTE to increase the instances of escaped defects (labeled as 1). The random state was set to 42 to guarantee randomness in the subsequent model executions.

In Table 7 and 8, we can visualize the data distribution before and after the technique, as mentioned earlier (undersampling and oversampling) and the data as they are. In the undersampling approach, the number of instances in class 0 reduced from 7457 to 534, according to the amount of the minority class: escaped defect (label 1). In the oversampling process, we noted the rise of the instances with class 1, from 534 to 7457. The new instances were created according to the amount of the majority class: not escaped defects (label 0).

We can better visualize the class distribution through Figure 24. The graph presents the data distribution using the PCA technique. The number of principal components used was two, as we wanted to plot it on a 2D graph. Looking at the graph, we can see the

Table 7 – Balanced training data for Team A.

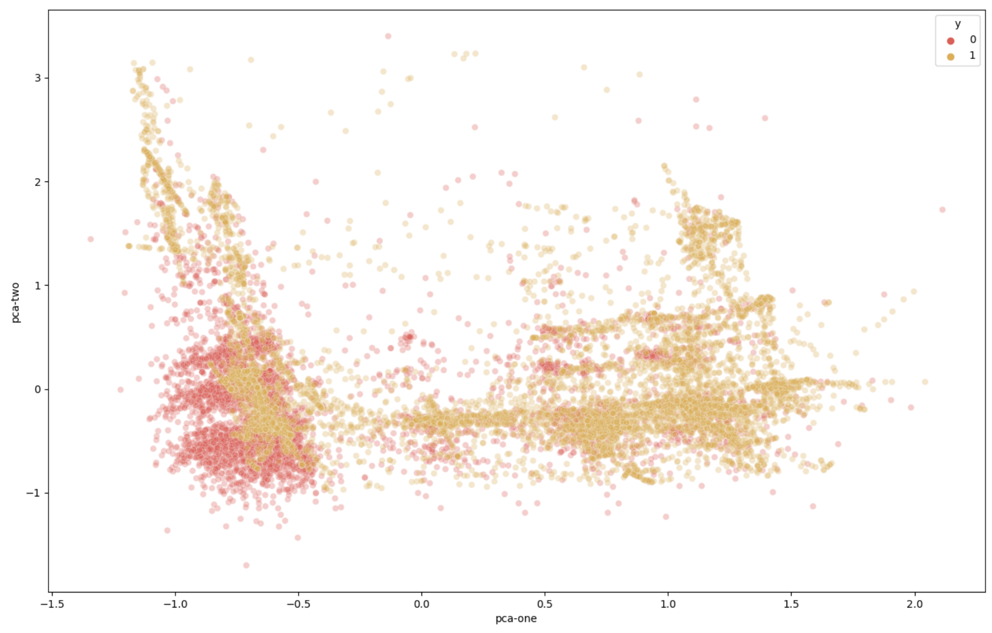
	Original Data	**Undersampling**	**Oversampling**	**Dist. Test Data**
Label 0	7457	534	7457	1864
Label 1	534	534	7457	134

Table 8 – Balanced training data for Team B.

	Original Data	**Undersampling**	**Oversampling**	**Dist. Test Data**
Label 0	1720	403	1720	74
Label 1	403	403	1720	42

dispersion of data in relation to our class of escaped defects (class 1) and non-escaped defects (class 0).

Figure 24 – Visualization of high-dimensional data using PCA.



Source: Authorial.

After completing the divisions, we started the classification step for building the ranking using the Random Forest classifier. After the training step, we applied the testing data as input to the model. As a result, we obtained the ranking with the probability of each classified instance. To calculate the quality of the generated ranking, we used the ROC and the cost-benefit curve, both explained in detail in the results chapter.

The Random Forest classifier received the dataset as input to each decision tree. They obtained the input values and, based on the patterns presented in the data, indicated the class (1 or 0) considering the defined threshold. These labels represented an escaped defect in the EDA context, where the class with more votes was chosen. Besides, in this work, we were interested in the classification ranking. So we collected the ranking that represented the probability of instances being an escaped defect or not, returning this ranking to the user.

4.3 EXPERIMENT TEAM A

In the previous section, we presented the general approach chosen. However, there were singularity settings for each one of the experiments. One of them was about data division. For example, in the case of Team A, due to data structure and organization, we split the dataset into 20% for testing and 80% for training.

The data of Team A lacked the information about cataloged months, so the division was based on percentage, differently from Team B. We also applied a shuffle into the data before the division to randomize with a fixed order. These were the main differences between Team A's and Team B's experiment protocols.

The experiment protocol of Team A followed the subsequent order: 1) load the whole data; 2) perform the division of training and testing; 3) set the parameters to randomize the instances; 4) clean the dataset; 5) apply balance techniques (oversampling and undersampling); 6) manipulate categorical and multi-categorical attributes (MLB); 7) handle textual attribute (like TF-IDF and others); 8) collect the results; 9) generate the EDA ranking; and 10) analyze the performance of the model.

4.4 EXPERIMENT TEAM B

For Team B, data from May/2020 to February/2021 were used for training and data from March/2021 were used for testing. This configuration was based on a machine learning system constantly receiving the latest month's data from the production line to analyze escaped defects. We used this configuration for Team B due to dataset's structure. During the protocol of experiment B, we cleaned the data and reset the index to help the manipulation instances.

The protocol of experiment B started with the training and testing data loading, which had been previously divided according to the months. After that, we proceeded with the previously mentioned approaches: SMOTE, TF-IDF, and MLB. Lastly, we collected and analyzed the results as we had done in Team A's experiment.

4.5 FINAL CONSIDERATIONS

In this chapter, we presented the methodology and protocols to run the experiments on two different datasets: the first of Team A and the second of Team B. We presented the datasets' structures and characteristics and the sampling approach and the Random

Forest classifier used. We also showed the difference between the experiments of the teams. After detailing the process, we can move forward to explain the results.

5 RESULTS

This chapter presents the results obtained of this work, discussing relevant points from different perspectives. For this purpose, we used two metrics to evaluate the model's performance, behavior, the reason for using each, and relations with relevant features.

This chapter will be divided into:

- Sampling - The impact of sampling approaches (oversampling and undersampling) on the data;
- Metrics Analysis - We present the ROC and Cost-Benefit curve;
- Data Discussion - We present the relation between the 'Product Affected', 'Resolution' and 'Team Found' with the label. We also present the analysis of different types of escaped defects with 'Components'. Lastly, we present the influence of different escaped defects types;
- Final Considerations - We conclude the chapter with an overview and the final thoughts.

5.1 SAMPLING

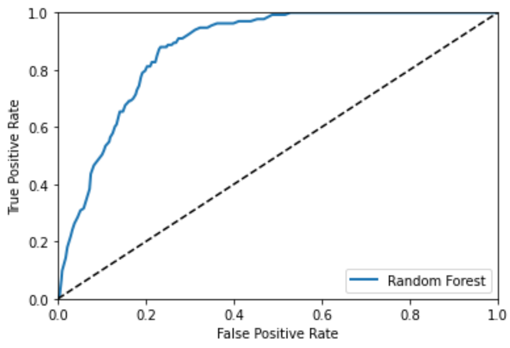
As we explained before, we identified imbalanced characteristics inside the dataset. So, we applied different sampling techniques to deal with inequality. The first was the undersampling; the second was the oversampling; in the last set, we did not change the data setup. In Figure 25 and 26, we can see the ROC curve of the different balance settings.

Analyzing Figure 27, we can see the confusion matrix in relation to the three sampling divisions used (undersample, oversample, and unbalanced). Although the values of the ROC curve are similar, it is possible to notice significant differences when looking at the confusion matrix.

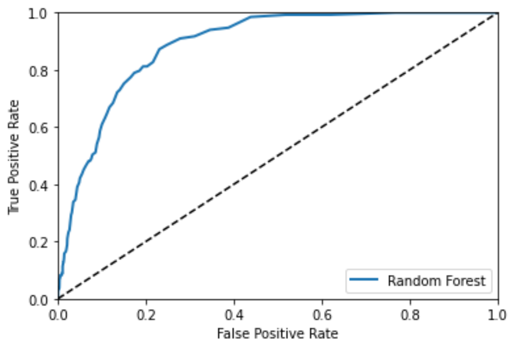
The unbalanced approach presented several hits and errors close to the oversample approach, shown in Figure 27. However, we understand there were more hits by the majority class due to the imbalance of the data, which emphasizes class 0. However, the oversample presented better prediction values of class 1 (Bug Reports marked as an escaped defect), with 111 hits, while the unbalanced approach had only 7.

The undersample removed instances of the majority class (class with label 0). Removing these instances impairs the classifier's generalization, as there is reduced confidence in predicting when the defect is not escaped (class 0). This aspect can be verified by comparing the hits of the model using undersample with the others: 1461 instances of the negative class (undersample) against 1823 (oversample) and 1854 (unbalanced). In

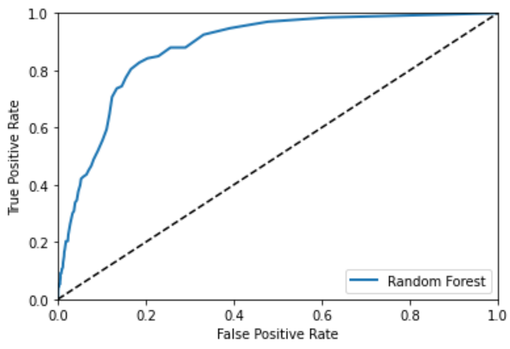
Figure 25 – ROC Curve with different approaches to Team A.



(a) Undersampling

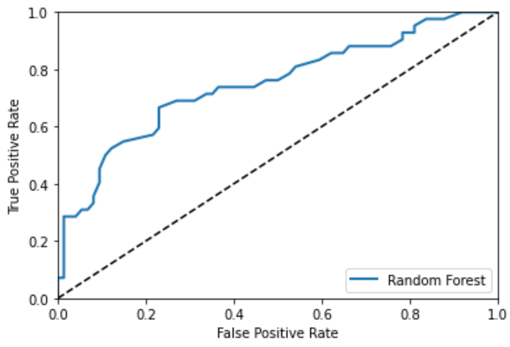


(b) Oversampling

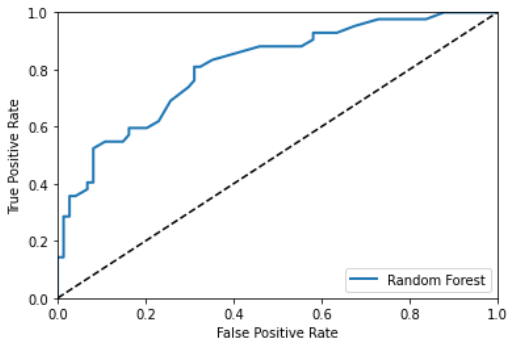


(c) Not Balanced

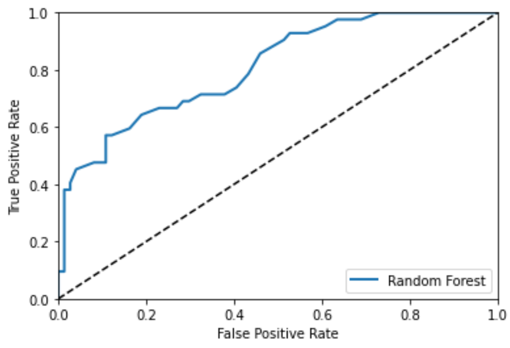
Figure 26 – ROC Curve with different approaches to Team B.



(a) Undersampling



(b) Oversampling

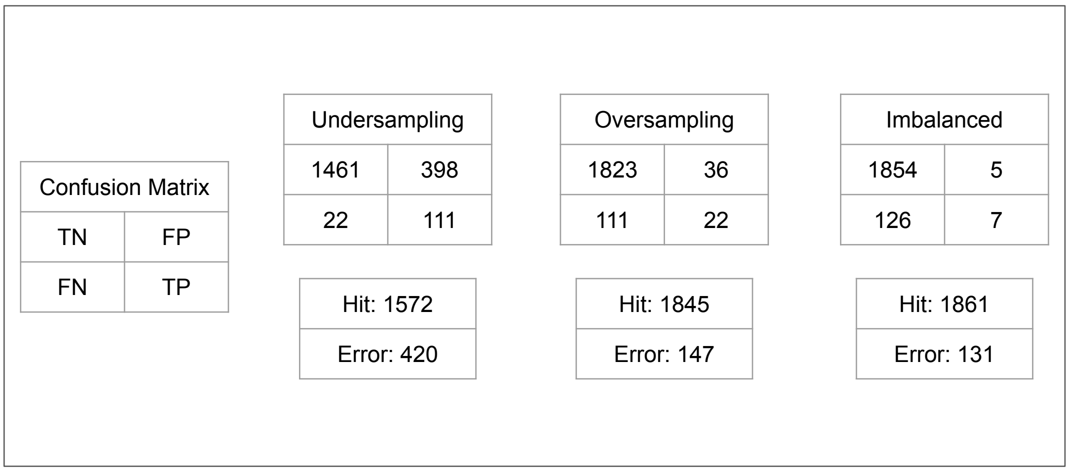


(c) Not Balanced

addition, more errors were computed: 420 for undersample against 147 and 131 for the oversample and unbalanced, respectively.

Considering the points presented in the different types of sampling, we can understand that an imbalance between the classes represents a problem. An imbalance tends to bias the model to the majority class. Also, in the case of undersampling, removing most instances of class 0 deprives the model of relevant data for generalization. For these reasons, we discarded both methods and adopted the oversample model to proceed with the experiments.

Figure 27 – Confusion matrix of the different sampling approaches to the Team A dataset.



Confusion Matrix	
TN	FP
FN	TP

Undersampling	
1461	398
22	111

Hit: 1572
Error: 420

Oversampling	
1823	36
111	22

Hit: 1845
Error: 147

Imbalanced	
1854	5
126	7

Hit: 1861
Error: 131

Source: Authorial.

5.2 METRICS ANALYSIS

This section discusses the ROC curve and the Cost-Benefit curve metrics. Along with the discussion, we explain the results from the perspective of the generated ranking and the relevant features.

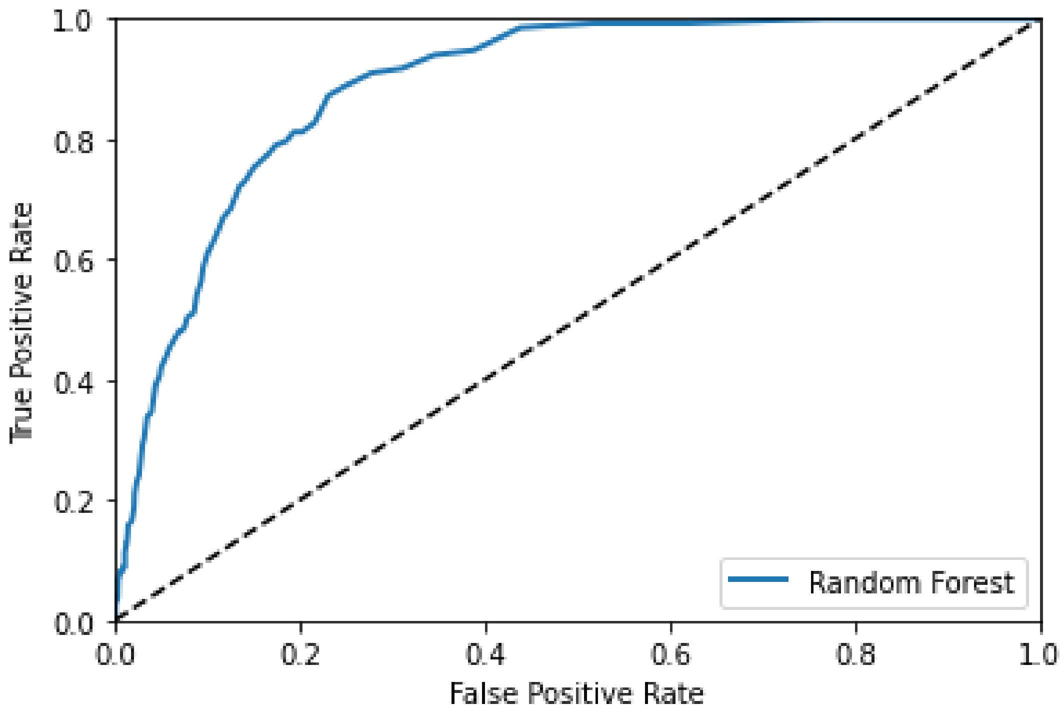
5.2.1 ROC Curve

The ROC curve was the first metric we used to measure the model's performance and the generated ranking. The goal was to communicate the relationship between the correctly and incorrectly classified escaped defect Bug Report instances.

As discussed earlier, to plot the ROC curve graphic, we calculated the TPR, the rate of instances correctly classified as an escaped defect, and the FPR, the Bug Report instances that were negative but had been incorrectly classified as positive. And then, with these values, we plotted the graphics for both teams to measure the classification results.

Figure 28 is the ROC curve for Team A. We can see that the classifier achieved a good performance by predicting bug reports, which were escaped defects. In general, there were many True Positives. It means that the Random Forest correctly classified the Bug Reports that were escaped defects. However, we also note that a few Bug Reports were classified in the wrong way as False Positives. It means that few bug reports were classified as escaped defects when in fact, they were not.

Figure 28 – ROC Curve for Team A.



Source: Authorial.

The performance of Team A was the one with the best result, pursuing a ROC AUC of 88% (Table 9), which indicates that the ranking generated for this team brought a considerable resolution of the bug reports that are and are not escaped defects.

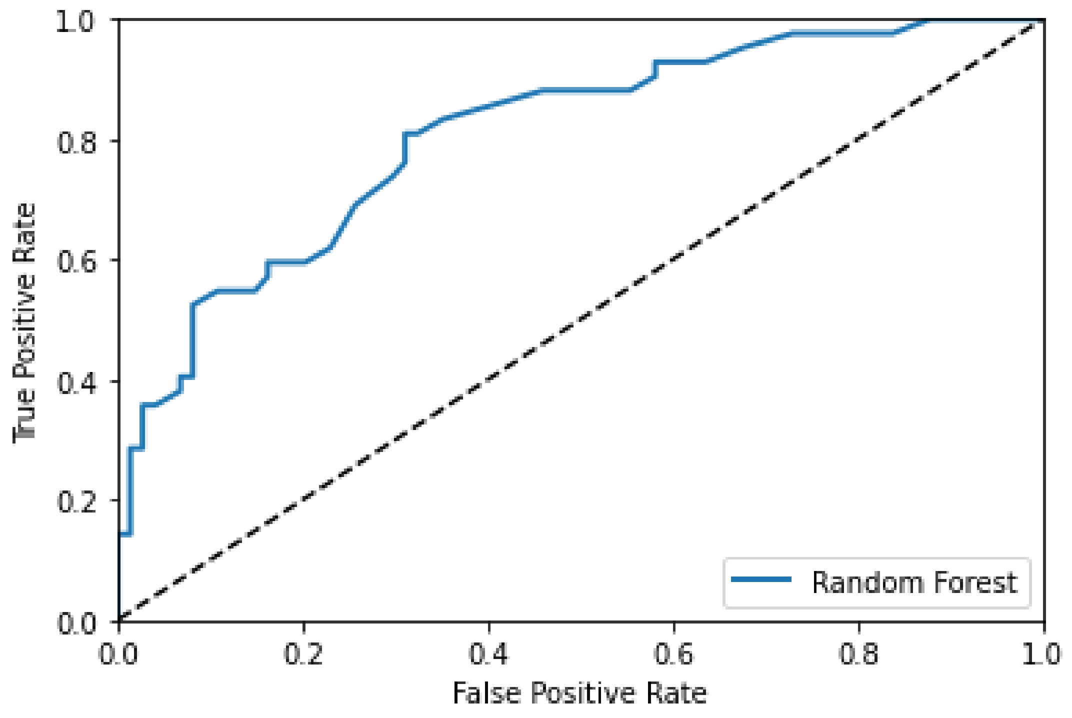
Table 9 – AUC results for Team A and Team B

Teams	Area Under the ROC Curve
Team A	88%
Team B	80%

Figure 29 represents the ROC curve of Team B. However, the curve presented some instability and lower result than Team A, but still with a ROC AUC value of 80% (Table 9). Therefore, it represented a value above the threshold dictated by the curve and can be considered a positive result.

We can better understand the ROC curve results when we analyze the generated ranking, as seen in the tables below. Tables 10 and 11 raise the ten Bug Reports with

Figure 29 – ROC Curve for Team B.



Source: Authorial.

more probability of being an escaped defect for Team A and B. These top ten Bug Reports presented some relevant features found when we analyzed the features' importance.

Table 10 – Ranking of Escaped Defects - Team A

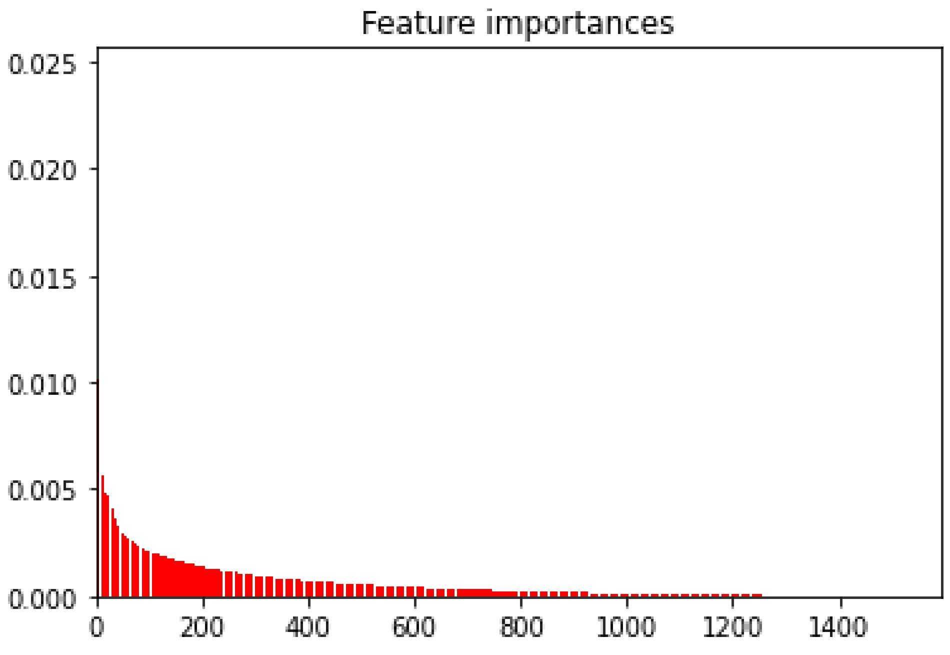
	Bug Report ID	**Classification**
1	IXTYU-24058	0.85
2	IXTYU-9835	0.79
3	IXTYU-17754	0.77
4	IXTYU-7762	0.72
5	IXTYU-50165	0.71
6	IXTYU-42165	0.71
7	IXTYU-1457	0.7
8	IXTYU-23859	0.69
9	IXHYOI-573	0.67
10	IXTP-8927	0.67

The feature importance, tells us what the features the model considered more relevant to make the classification. Figure 30 presents the features' importance for Team A, assigning a score to each feature. The sum of all scores totalizes 1 and features with a higher score are considered the most important features for the model classification. We note in the figure the proportion of the features considered relevant to our context in the figure.

Table 11 – Ranking of Escaped Defects - Team B

	Bug Report ID	**Classification**
1	IXRYG-14785	0.86
2	IXRYG-96371	0.79
3	IXRYG-22497	0.72
4	IXRYG-45789	0.71
5	IXRYG-27934	0.7
6	IXRYG-98720	0.66
7	IXRYG-03257	0.65
8	IXRYG-48453	0.64
9	IXRYG-775402	0.62
10	IXRYG-452100	0.61

Figure 30 – Feature importance.



Source: Authorial.

We listed these features in Table 12 to Team A and Table 13 to Team B, where we have the feature's name and its respective score. With these tables, we can perceive the main attributes that Random Forest considered to build the model. For example, there are important features related to just one of the classes, escaped defect or not escaped defect, but there are also relevant features linked with both classes. This behavior occurs for both teams. The ranking reflects the characteristics of these features' importance, and the top ten Bug Reports probably contain bugs with relevant characteristics. Therefore, the ROC curve measured the performance of the generated ranking based on the relevant features for each team.

Table 12 – Feature importance - Team A

Attribute	**Score**
Team Found	0.046
Tag	0.027
Tag	0.026
Team Found	0.025
Summary	0.025
Tag	0.012
Summary	0.010
Tag	0.009
Resolution	0.007
Components	0.007

Table 13 – Feature importance - Team B

Attribute	**Score**
Tag	0.020
Tag	0.011
Product Affected	0.010
Components	0.010
Tag	0.009
Product Affected	0.009
Tag	0.008
Team Found	0.008
Team Found	0.007
Tag	0.007

5.2.2 Cost-Benefit Curve

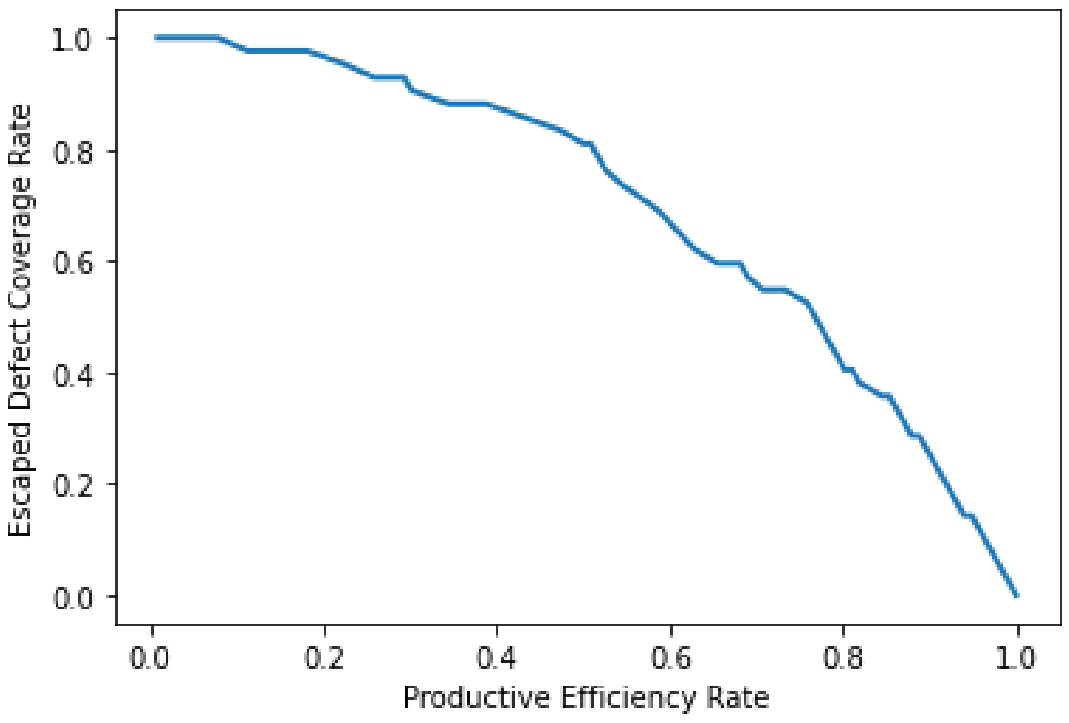
The cost-benefit curve was the second metric we used to measure our ranking performance. It measures the gain generated by the ranking when analyzing escaped defects—bearing in mind that the tester will need to examine only those Bug Reports more likely to be an escaped defect.

As we said before, the cost-benefit curve will measure the relationship between the escaped defect rate and the productive efficiency rate. The productive efficiency rate has an important role here because once the raking is generated, the tester will exclude those instances indicated as having not escaped defects. Thus,working time will be reduced, consequently improving the productive efficiency rate.

So, to plot the Cost-Benefit graphic, we need two metrics: (1) The Escaped Defect Coverage Rate for the y-axis; and (2) the Productive Efficiency Rate for the x-axis. The first step was calculating the y_score, which means the predicted label. Once we had the y_score calculated, we generated the Confusion Matrix to y_score several times. It was generated for a threshold that starts from 0 until 1. Then, we calculated the two metrics cited above (1 and 2) for each confusion matrix generated. After calculating the metric, we stored the value. So, in the end, we had a list of Escaped Defect Coverage Rate values and Productive Efficiency Rate values, which helped plot the graphic.

As a result of the cost-benefit curve, Figure 31 shows that, for Team B, the lower the Escaped Defect Coverage Rate, the higher the Productive Efficiency Rate, almost proportionally. Therefore, through the graphic, we note that the ranking generated will insert gain in work done by the tester, removing the bug reports classified as not escaped with a considerable correctness rate.

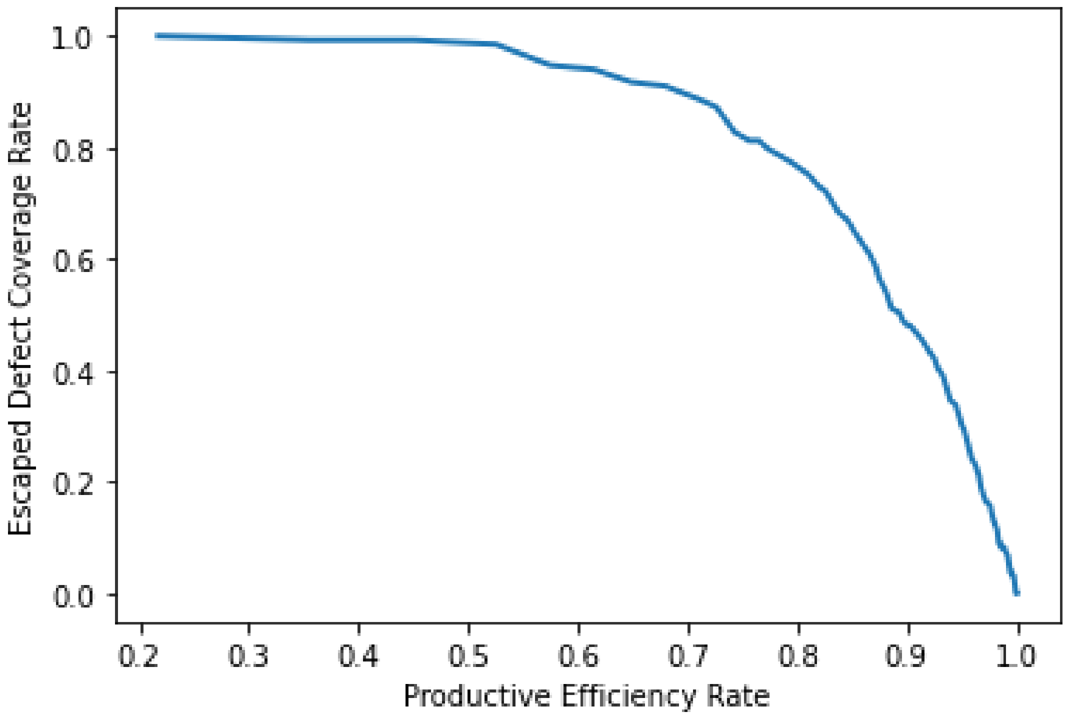
Figure 31 – Cost-Benefit Curve for Team B.



Source: Authorial.

In Figure 32, we have the graphic of the cost-benefit curve for Team A. It is possible to notice that Team A has a better relationship between the coverage rate of escaped defects and the production efficiency rate, while for Team B, 50% coverage of escaped defects represented about 80% of productive efficiency. For team A, 50% coverage of escaped defects represents about 90% of productive efficiency. Therefore, we can note performance gained by using the ranking for both teams.

Figure 32 – Cost-Benefit Curve for Team.



Source: Authorial.

5.3 DATA DISCUSSION

Before modeling the classifier, an exploratory analysis was performed on the data to identify patterns manually and better understand the problem domain. After automating the process, many of these patterns were confirmed, endorsing the quality of the model. This section discusses each of these points, relating manual to automated patterns.

5.3.1 Relation between Features and Label

Before starting the model development, we applied an exploratory data analysis to understand the features' behavior. Therefore, in the following paragraphs, we present some expressive relations between the attributes and the labels in the sampling used.

In Figure 33, we have a bar graph where the x-axis contains the name of the Product Affected, and the y-axis is the appearance percentage of occurrence from 0 to 1. This graph shows the number of times a specific product appeared in the Bug Report and was considered escaped or not. The blue bar shows the amount of occurrences of escaped defects while the green bar stands for the Product Affected that is not an escaped defect.

We can see the relation between the Product Affected feature and the label, with ‘yes’ y_label representing the escaped defects while ‘no’ is the not escaped defect. Furthermore, we can note the predominance of the ‘Device 2’ ‘Device 18’ and ‘Device 26’ with the label representing the escaped defect. So, we can point to these Products Affected as an expressive feature of an escaped defect, confirmed through the feature importance performed. Therefore, from the industry tester’s point of view, it is possible to redirect the efforts towards these products to avoid future escaped defects or update the testing plan.

A potential interpretation for these labeled products as escaped defects may be due to their release date. For example, they could have been released recently, and there might not have been enough testing cases to cover the new functionalities. On the other hand, another interpretation we can raise is about recent updates in the software of these products. Alternatively, yet, they were included in the test recently.

On the other hand, we can see ‘Device 4’, ‘Device 9’, ‘Device 13’, ‘Device 15’, ‘Device 16’, ‘Device 20’, ‘Device 24’, ‘Device 27’ and ‘Device 29’ as not escaped defects. So again, the tester can use this information to redirect the team’s efforts and gain work performance. So the first point we can understand from these products is that there is no need to be concerned about them once the work is directed to validate and fix the escaped defects. One possible interpretation for them not being escaped defects could be that they already have been tested a lot, so all the test cases are well documented, covering all scenarios, and all potential defects have been mitigated.

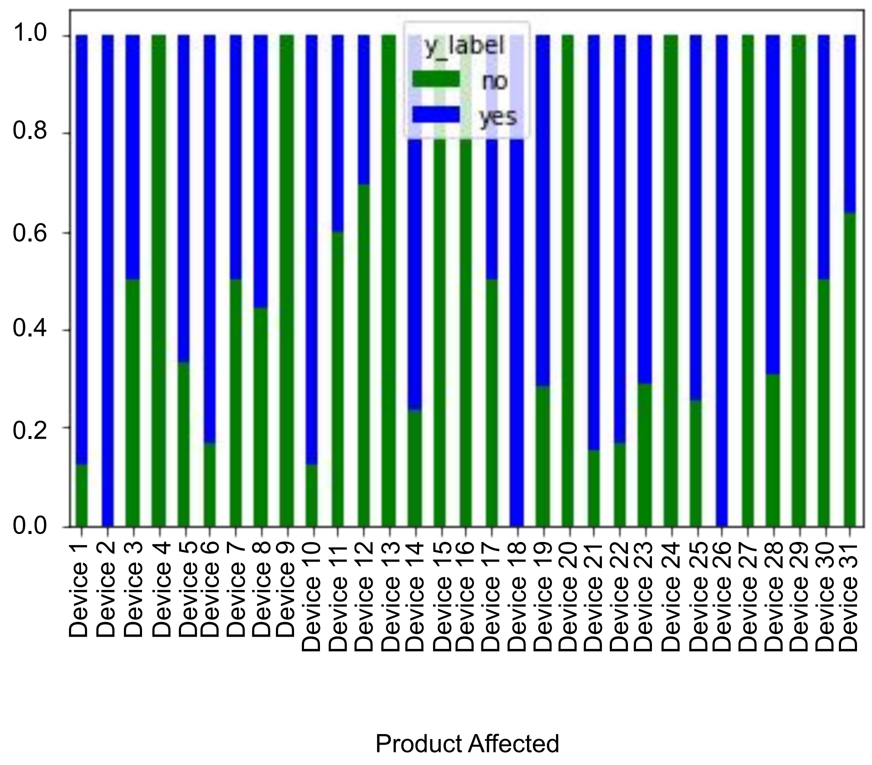
These Products Affected presented powerful graph visualization with totality labels zero or one. However, other relations could be considered to redirect efforts to that product with the major part labeled ‘yes’ or ‘no’.

The second relationship we can establish is between the Resolution feature and the label. Figure 34 shows the influence of the Resolution on the labels. We can note the ‘Resolution 1’ and ‘Resolution 7’ Resolution features contain no escaped defects, and most parts of the escaped defects are ‘Resolution 4’. When detailing these Resolutions, we can understand that some escaped defects occur for different reasons, like Android updates or releases. So, in these cases, the responsibility for the escaped defect is not necessarily of the testing team.

Another relation we can see is between the Team Found feature and the label. The team who found the bug is substantial to determine if the defect is escaped or not. We can note that most parts of escaped defects are detected by the ‘Team 3’ team (Figure 35).

Still on the Team Found attribute, if the team is ‘Team 1’, this is a very relevant factor to determine that it is not an escaped defect (see Figure 35). In terms of classification,

Figure 33 – Relation Product Affected x Label.



Source: Authorial.

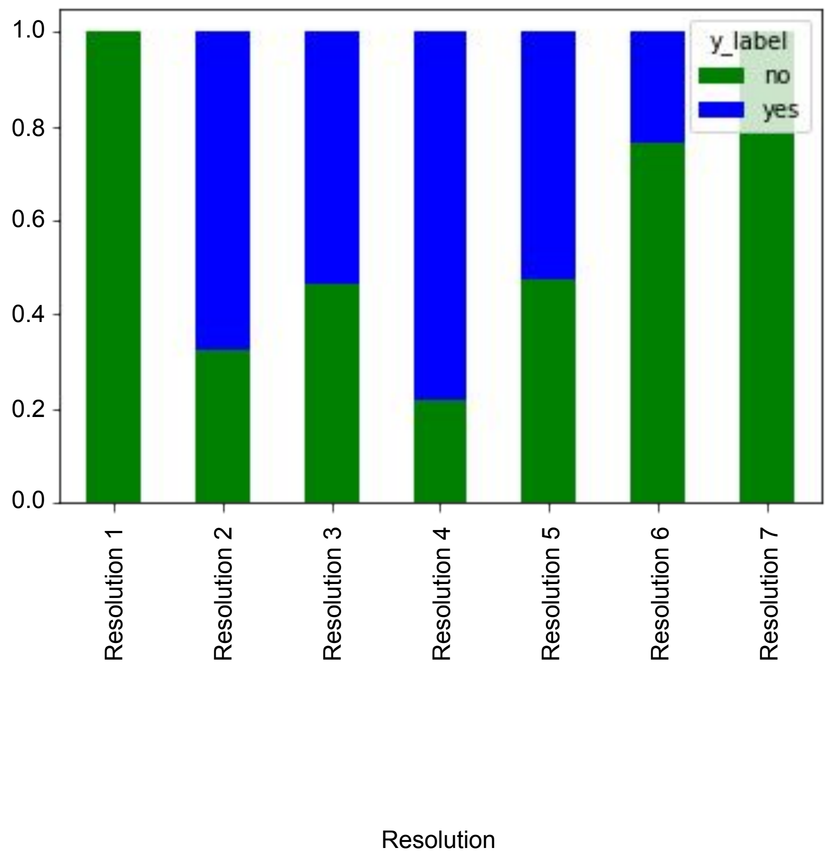
this information is confirmed by analyzing the first most important features defined by Random Forest, which includes different categories of Team Found. It was the fourth most important feature for Team A, while for Team B, it was the fifteenth. It is important to highlight that the total number of features analyzed was 13446 (Team A) and 7052 (Team B).

5.3.1.1 Analysis of Different Types of Escaped Defect

As we said before, there are five types of classes defining a Bug Report as an escaped defect: Type 1 Escaped Defect, Type 2 Escaped Defect, Type 3 Escaped Defect, Type 4 Escaped Defect, and Type 5 Escaped Defect.

Table 14 presents the relevance of the number of categories in each attribute, just considering the different types of escaped defects, just the bugs marked as escaped. The columns represent each type of escaped defect, while the lines are the different attributes we used in our dataset. That table brings the amount of each attribute considering the different types of escaped defects. Consequently, it informs the importance of the attributes given their presence in the dataset. For example, in the first line, we see the presence of two values of "Resolution" in the "Type 4 Escaped Defect". It tells us that among all the resolutions categories, just two occurrences in the "Type 4 Escaped Defect". We could say that an attribute is very important for an escaped defect type if all the categories

Figure 34 – Relation Resolution x Label.



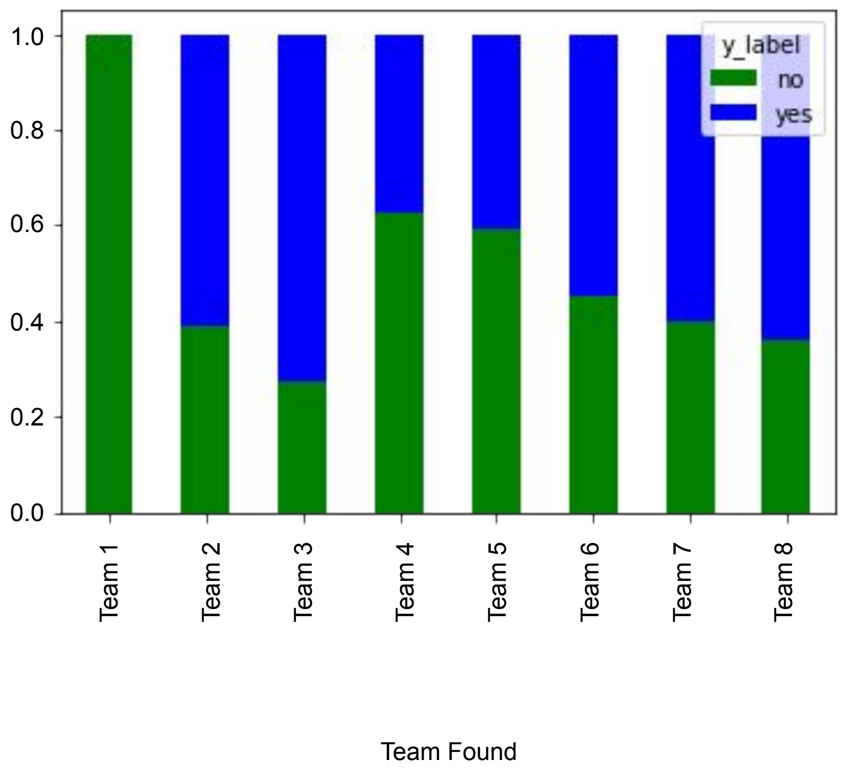
Source: Authorial.

of an attribute appear in that type. If zero appears, we can say that attribute is slightly relevant. So, we can note the absence of Resolution and Team Found categories during the Bug Reports Missed by Type 3. We can understand the team who detected the defect, and the Bug Report status was not relevant information to identify a bug report Type 3 Escaped Defect.

On the other hand, the components, product affected, and tags had an expressive relation in the categorization. The other feature that had a relevant presence in the other type of escaped defects was ‘Tags’ with the most expressive presence. The escaped defect type, with a significant number of categories taken into account to be classified, was "Type 1 Escaped Defect". It could mean more diversity inside the defects Missed by Type 1.

Continuing the observations into the different types of escaped defects, Figure 36 presents a distribution of escaped defect types in the presence of the Components' categories. We can note the solid presence of the Components' distribution in Type 5 and Type 3 Escaped Defects. The same behavior is not observed in Table 14. We can see that the feature Component has a weak presence, being four for Missed by Type 3 and thirteen for Missed by Type 5. Therefore, we can infer that, although it is pretty frequent, it is

Figure 35 – Relation Team Found x Label.



Source: Authorial.

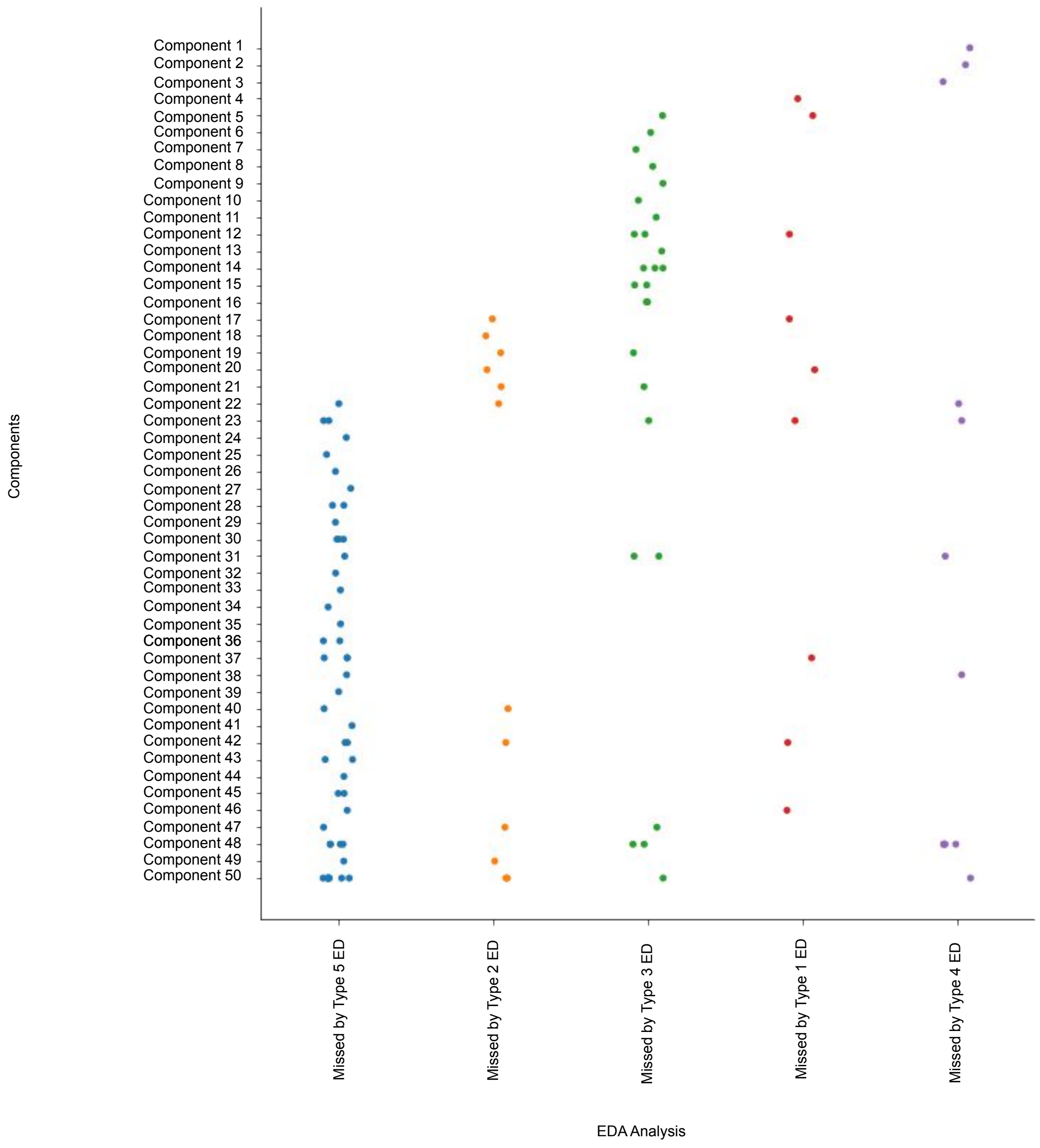
Table 14 – Relevant feature by classes

	Type 1 ED	**Type 2 ED**	**Type 3 ED**	**Type 4 ED**	**Type 5 ED**
Resolution	2	2	0	2	1
Team Found	5	0	0	1	2
Components	18	16	4	12	13
Product Affected	11	8	6	13	6
Tags	164	173	191	182	178

not relevant due to the weak presence noted in the table.

As mentioned, we can note the points raised in this subsection present in the results model, which show us a descriptive behavior in the relation between the attributes and the results. This relationship is notable in the features characteristic in the ranking.

Figure 36 – Relation Components and Escaped Defect Types.



Source: Authorial.

5.4 FINAL CONSIDERATIONS

This chapter presented the relationship between features, label, and the metrics we used to evaluate our results. Furthermore, we also presented our results. We evaluated our model for both Team A and B. In both, we obtained significant performance according to the ROC curve. We also evaluated the rank generated. The rank is the main product for the user/tester. The tester will receive the rank, and based on it, the tester can choose the best approach to analyze the escaped defects. Decreasing the time and consequently increasing productivity. With the cost-benefit curve of both teams, we can note productivity improvements.

6 CONCLUSION

We proposed a machine learning system to prioritize bug reports with escaped defects in this work. During the work development, we built different artifacts that we can define as contributions to the final solution, from the final result to the approach we created to build the project.

We can enumerate our results in the following order:

- A ranking of the Bug Reports documents - The user/tester company will utilize this ranking to do the automated EDA analysis. From the user's point of view, the ranking is the main product generated through the project. As we said before, the idea is that this ranking helps the EDA analysis;
- A model to classifier the Bug Reports - The model uses the Random Forest classifier to predict the class of each bug report instance;
- A new metric we called the cost-benefit curve - We used this metric to analyze our results as we did with the ROC curve. The Cost-Benefit translates the gain obtained using our approach;
- The model evaluation - We pointed out the evaluation results. We could analyze the model's performance through the ROC curve and cost-benefit curve;
- New EDA Analysis process - A new approach to optimize the EDA analysis and improve team performance. The method uses the ranking inside the tester activity instead of the manual EDA analysis.

In summary, we consider attributes and characteristics extracted from Bug Reports. This information was processed and passed to a classification model built by Random Forest. We measured our approach with the ROC and Cost-benefit curve. The result for Team A was 88%, and for Team B, 80%. Both teams presented significant gains in Escaped Defect Analysis (EDA) performance. The tester can use our technique as a ranking system to prioritize escaped defect bugs. Therefore, it is possible to save time and improve performance.

6.1 FUTURE WORK

In the future, we plan to compare our technique with more recent and advanced machine learning approaches, like neural networks. Also, we intend to add more bug report features, like the comments field. In addition, we plan to evolve the tool to discover the type of escaped defect the Bug Report belongs to. As a consequence, improve the process of testing in the software industry.

REFERENCES

BALAKRISHNAN, V.; LLOYD-YEMOH, E. Stemming and lemmatization: a comparison of retrieval performances. 2014.

BALAKRISHNAN, V.; LLOYD-YEMOH, E. Stemming and lemmatization: a comparison of retrieval performances. 2014.

BARANAUSKAS, J. A.; MONARD, M. C. Reviewing some machine learning concepts and methods. 2000.

BELGIU, M.; DRĂGUŢ, L. Random forest in remote sensing: A review of applications and future directions. *ISPRS journal of photogrammetry and remote sensing*, Elsevier, v. 114, p. 24–31, 2016.

BEYER, K.; GOLDSTEIN, J.; RAMAKRISHNAN, R.; SHAFT, U. When is “nearest neighbor” meaningful? In: SPRINGER. *International conference on database theory.* [S.l.], 1999. p. 217–235.

BIAU, G.; SCORNET, E. A random forest guided tour. *Test*, Springer, v. 25, n. 2, p. 197–227, 2016.

BROWNLEE, J. *Machine learning algorithms from scratch with python.* [S.l.]: Machine Learning Mastery, 2016.

CÂMPEANU, C.; SALOMAA, K.; YU, S. Regex and extended regex. In: SPRINGER. *International Conference on Implementation and Application of Automata.* [S.l.], 2002. p. 77–84.

CHAWLA, N. V.; BOWYER, K. W.; HALL, L. O.; KEGELMEYER, W. P. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, v. 16, p. 321–357, 2002.

CHAWLA, N. V.; BOWYER, K. W.; HALL, L. O.; KEGELMEYER, W. P. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, v. 16, p. 321–357, 2002.

CLARKSON, K. L. Nearest neighbor queries in metric spaces. *Discrete & Computational Geometry*, Springer, v. 22, n. 1, p. 63–93, 1999.

COVER, T.; HART, P. Nearest neighbor pattern classification. *IEEE transactions on information theory*, IEEE, v. 13, n. 1, p. 21–27, 1967.

CUTLER, A.; CUTLER, D. R.; STEVENS, J. R. Random forests. In: *Ensemble machine learning.* [S.l.]: Springer, 2012. p. 157–175.

DAVIS, A. M.; BERSOFF, E. H.; COMER, E. R. A strategy for comparing alternative software development life cycle models. *IEEE Transactions on software Engineering*, IEEE, v. 14, n. 10, p. 1453–1461, 1988.

DEVELOPERS, G. *Imbalanced Data.* 2022. Available at: <https://developers.google.com/machine-learning/data-prep/construct/sampling-splitting/imbalanced-data>.

EGUIDE, T. *The Impact of Software Failure-And How Automated Testing Reduces Risks. Tricentis (2017)*. 2018.

EICKELMANN, N.; ANANT, A. Statistical process control: what you don't measure can hurt you! *IEEE Software*, IEEE, v. 20, n. 2, p. 49–51, 2003.

FAWCETT, T. An introduction to roc analysis. *Pattern recognition letters*, Elsevier, v. 27, n. 8, p. 861–874, 2006.

FRANKLIN, A. *Software Testing Life Cycle – A Complete Guide For 2022*. 2019. Available at: <https://www.goodcore.co.uk/blog/software-testing-life-cycle/>.

GÉRON, A. Hands-on machine learning with scikit-learn and tensorflow: Concepts. *Tools, and Techniques to build intelligent systems*, 2017.

GREFENSTETTE, G. Tokenization. In: *Syntactic Wordclass Tagging*. [S.l.]: Springer, 1999. p. 117–133.

HABERT, B.; ADDA, G.; ADDA-DECKER, M.; MARËUIL, P. B. de; FERRARI, S.; FERRET, O.; ILLOUZ, G.; PAROUBEK, P. Towards tokenization evaluation. In: *Proceedings of LREC*. [S.l.: s.n.], 1998. v. 98, p. 427–431.

HE, H.; GARCIA, E. A. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, Ieee, v. 21, n. 9, p. 1263–1284, 2009.

HERZIG, K.; GREILER, M.; CZERWONKA, J.; MURPHY, B. The art of testing less without sacrificing quality. In: IEEE. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. [S.l.], 2015. v. 1, p. 483–493.

JAMIL, M. A.; ARIF, M.; ABUBAKAR, N. S. A.; AHMAD, A. Software testing techniques: A literature review. In: IEEE. *2016 6th international conference on information and communication technology for the Muslim world (ICT4M)*. [S.l.], 2016. p. 177–182.

JAPKOWICZ, N. The class imbalance problem: Significance and strategies. In: CITESEER. *Proc. of the Int'l Conf. on Artificial Intelligence*. [S.l.], 2000. v. 56, p. 111–117.

KALE, A. M.; BANDAL, V. V.; CHAUDHARI, K. A review paper on software testing. *International Research Journal of Engineering and Technology (IRJET)*, v. 6, n. 1, p. 1268–1273, 2019.

KANWAL, J.; MAQBOOL, O. Bug prioritization to facilitate bug report triage. *Journal of Computer Science and Technology*, Springer, v. 27, n. 2, p. 397–412, 2012.

KAUR, J.; BUTTAR, P. K. A systematic review on stopword removal algorithms. *International Journal on Future Revolution in Computer Science & Communication Engineering*, v. 4, n. 4, p. 207–210, 2018.

LEEPER, J. What is the difference between categorical, ordinal and interval variables. *Choosing the correct statistic*, 2000.

LING, C. X.; LI, C. Data mining for direct marketing: Problems and solutions. In: *Kdd*. [S.l.: s.n.], 1998. v. 98, p. 73–79.

LIU, X.-Y.; WU, J.; ZHOU, Z.-H. Exploratory undersampling for class-imbalance learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, IEEE, v. 39, n. 2, p. 539–550, 2008.

LORENA, A.; FACELI, K.; ALMEIDA, T.; CARVALHO, A. de; GAMA, J. *Inteligência Artificial: uma abordagem de Aprendizado de Máquina (2a edição)*. [S.l.: s.n.], 2021. ISBN 9788521637493.

MCCORMICK, M. Software development life cycle. In: *The Agile Codex*. [S.l.]: Springer, 2021. p. 79–85.

MICHIE, D.; SPIEGELHALTER, D. J.; TAYLOR, C. C. Machine learning, neural and statistical classification. Citeseer, 1994.

MULLICK, S. S.; DATTA, S.; DAS, S. Generative adversarial minority oversampling. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. [S.l.: s.n.], 2019. p. 1695–1704.

NARKHEDE, S. Understanding auc-roc curve. *Towards Data Science*, v. 26, n. 1, p. 220–227, 2018.

OTOOM, A. F.; AL-JDAEH, S.; HAMMAD, M. Automated classification of software bug reports. In: *Proceedings of the 9th International Conference on Information Communication and Management*. [S.l.: s.n.], 2019. p. 17–21.

PEDREGOSA, F.; VAROQUAUX, G.; GRAMFORT, A.; MICHEL, V.; THIRION, B.; GRISEL, O.; BLONDEL, M.; PRETTENHOFER, P.; WEISS, R.; DUBOURG, V. et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, JMLR. org, v. 12, p. 2825–2830, 2011.

PLISSON, J.; LAVRAC, N.; MLADENIC, D. et al. A rule based approach to word lemmatization. In: *Proceedings of IS*. [S.l.: s.n.], 2004. v. 3, p. 83–86.

POZZOLO, A. D.; CAELEN, O.; JOHNSON, R. A.; BONTEMPI, G. Calibrating probability with undersampling for unbalanced classification. In: IEEE. *2015 IEEE Symposium Series on Computational Intelligence*. [S.l.], 2015. p. 159–166.

REHKOPF, M. *Automated software testing*. 2021. Available at: <https://www.atlassian.com/continuous-delivery/software-testing/automated-testing>.

SABOR, K. K.; HAMDAQA, M.; HAMOU-LHADJ, A. Automatic prediction of the severity of bugs using stack traces. In: *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*. [S.l.: s.n.], 2016. p. 96–105.

SCHUBACH, M.; RE, M.; ROBINSON, P. N.; VALENTINI, G. Imbalance-aware machine learning for predicting rare and common disease-associated non-coding variants. *Scientific reports*, Nature Publishing Group, v. 7, n. 1, p. 1–12, 2017.

SHULL, F.; BASILI, V.; BOEHM, B.; BROWN, A. W.; COSTA, P.; LINDVALL, M.; PORT, D.; RUS, I.; TESORIERO, R.; ZELKOWITZ, M. What we have learned about fighting defects. In: IEEE. *Proceedings eighth IEEE symposium on software metrics*. [S.l.], 2002. p. 249–258.

SPAVEN, F. *How much could software errors be costing your company?* 2017. [Online]. Available: https://raygun.com/blog/cost-of-software-errors/.

TIAN, Y.; LO, D.; XIA, X.; SUN, C. Automated prediction of bug report priority using multi-factor analysis. *Empirical Software Engineering*, Springer, v. 20, n. 5, p. 1354–1383, 2015.

TRICENTIS. *Exploratory Testing: The Heart of All Things Testing.* 2016. [Online; accessed 11-Dec-2017]. Available at: <https://www.tricentis.com/wpcontent/uploads/2016/11/201610_Exploratory-Testing_Whitepaper.pdf>.

TURPITKA, D. *Time estimation for software testing.* 2016. Available at: <https://jaxenter.com/time-estimation-for-software-testing-128078.html>.

TUTEJA, M.; DUBEY, G. et al. A research study on importance of testing and quality assurance in software development life cycle (sdlc) models. *International Journal of Soft Computing and Engineering (IJSCE)*, v. 2, n. 3, p. 251–257, 2012.

URIAS, V.; RIVAS, A. *Improving the software delivery life cycle through automation and analytics.* [S.l.], 2019.

VANDERMARK, M. A. Defect escape analysis: Test process improvement definition of escape. *STAREAST 2003 - SOFTWARE TESTING CONFERENCE*, January 2003.

WIRTH, R.; HIPP, J. Crisp-dm: Towards a standard process model for data mining. In: MANCHESTER. *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining.* [S.l.], 2000. v. 1, p. 29–39.

YAUNG, A. T.; RAZ, T. A predictive model for identifying inspections with escaped defects. In: IEEE. *Proceedings Eighteenth Annual International Computer Software and Applications Conference (COMPSAC 94).* [S.l.], 1994. p. 287–292.

ZHANG, C.; MA, Y. *Ensemble machine learning: methods and applications.* [S.l.]: Springer, 2012.