Università
degli Studi
di Padova

Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA “TULLIO LEVI-CIVITA”

Corso di Laurea Magistrale in Matematica

Embedding Explainable AI for Quality Code Prediction in Devops

Relatore:
Prof. Roberto Confalonieri

Luca Vinti
Matricola: 2022143

Anno Accademico 2023/2024

16/04/2024

Abstract

Motivation In software development it is very common to deal with bugs during the code-writing process and not every bug is detected through the usual validation work, and consequently, post-release bugs remain present. These bugs can impact the final product and they can be fixed by subsequent software releases, but this process of bug fixing can be expensive and time-consuming.

Problem statement This research introduces a novel approach called predictive bug modeling. Using data analytics and machine learning, it aims to predict and resolve software bugs at the initial stages of coding. Instead of relying on traditional bug-fixing methods that address issues after they occur, this approach tries to prevent them and it tries to find them more efficiently.

Approach To solve the problem we used some machine learning models to study the presence of bugs in the project on a file level, we confronted different models and at the end we embedded the code in the pipeline to make it work the model every time that a file is modified. In particular we extracted software metrics, but since they were not enough, we increased their size using data augmentation techniques. Then, we split the dataset into two parts: files with bugs and bug-free files. We trained various models and tested them to obtain the results. Finally, we used Explainable AI techniques to understand how the various models worked.

Results The result of most models in terms of precision is good. We integrated the predictive bud modeling into a devops pipeline. Explainable AI techniques helped us to comprehend why a file is defective so that we can intervene on that file in order to change it and to make it better and fixing bugs.

Conclusions This is only a first step. The models could be improved, for example, one could look for a way for the models to work primarily not on static metrics, but rather on dynamic metrics to show the possibility that a file may have a bug between two commits, assuming that it was bug-free before the previous commit. It is also possible to work on a non-deterministic prediction, but rather to divide the whole thing into more nuances (definitely bug-free, low probability of bug in the file, medium probability of bug in the file, high probability of bug in the file, definitely has a bug in the file) to have a more complete view. or finally, one could work on comparing these models between projects and not between files to see which ones are more truthful and perhaps also try with a finer granularity (e.g., at the

module level) to see if there is a possibility of further improving the ”zoom” without affecting the excellent precision figures obtained in this experiment.

Contents

1 Introduction 1
1.1 Motivation . . . 1
1.2 Objective . . . 2
1.3 Approach . . . 2
1.4 Structure of the thesis . . . 3

2 Background Information 5
2.1 Introduction . . . 5
2.1.1 Agile methodology . . . 5
2.1.2 DevOps . . . 8
2.2 Software Quality Assurance . . . 9
2.2.1 SQA activities during the development stage . . . 10
2.2.2 SQA activities during the release preparation . . . 10
2.2.3 How to calculate SQA . . . 11
2.3 Defect prediction . . . 12
2.3.1 The modelling pipeline of defect prediction models . . . 12
2.3.2 The granularity levels of defect predictions models . . . 13
2.4 The models . . . 13
2.4.1 Logistic regression . . . 13
2.4.2 Decision tree . . . 14
2.4.3 Random Forest . . . 15
2.4.4 Gradient Boosting Machine . . . 17
2.4.5 Neural Network . . . 18
2.4.6 Extreme Gradient Boosting Tree . . . 20
2.5 Explainable AI . . . 21
2.5.1 What is Explainable AI and why it is important . . . 21
2.6 Techniques of Explainable AI . . . 21

3 Problem Statement 27
3.1 Requirements . . . 28
3.1.1 Quality scenarios . . . 28

4 Review of the state of the art 31

5 Problem Solution **35**
5.1 Data collection 35
5.2 Clustering 37
5.2.1 Principal Component Analysis (PCA) 37
5.3 Model Construction 38
5.4 Model Evaluation 39
5.5 Model Ranking 41
5.6 Explainable AI 43
5.6.1 An explainable model: Decision Tree 43
5.7 Insertion in pipeline 52

6 Evaluation **55**

7 Discussion **59**
7.1 Difficulties: what could have done better? 59
7.1.1 Data collection 59
7.1.2 Error log 61
7.2 Positive aspects 62

8 Conclusion and Further Studies **63**

Chapter 1

Introduction

During software development, it is very common to face bugs. However, not all bugs are caught during testing, leading to bugs that remain after release.

These hidden bugs can affect the final product and require additional software updates to fix them. If a bug in a car's infotainment system goes undetected, it could pose a serious risk to drivers. Identifying and fixing post-release bugs is challenging and time-consuming, consuming valuable resources.

When a new software application is released, despite careful planning, unexpected bugs can still arise. While fixing these bugs in later releases is essential, it requires significant time, effort, and money from the development team. Identifying them could be more challenging than writing the code itself, especially when they are assigned to developers unfamiliar with the project or who have worked on it extensively. This highlights a common struggle in modern software development: the ongoing presence of bugs and the strain they put on development resources.

1.1 Motivation

Software developers face an ongoing issue with bugs that show up after software is released. These bugs can slow down progress and use up resources. Finding and fixing these bugs is a big part of keeping software in good shape, and it can take a lot of time, work, and money.

To effectively address this ongoing issue, we need a proactive and comprehensive approach to detecting and preventing bugs. Such a solution should be seamlessly integrated into the software development process, becoming an essential part of the quality assurance phase. By employing advanced technologies like automated testing, static code analysis, and anomaly detection, this solution would proactively identify potential vulnerabilities and defects before they appear in the final product.

An ideal bug detection system should not only identify defects but also predict future issues using historical data, code complexity measurements, and context analysis. By constantly monitoring the system and providing feedback, it would help development teams proactively resolve potential problems. This would strengthen the software's overall dependability and resilience.

This ground-breaking solution not only detects bugs but also transforms software development. By proactively preventing bugs, teams can use their time better. They can now concentrate on creating new features and enhancing products. This speeds up the release of high-quality software without compromising quality.

Not only users but all those involved in software development (stakeholders) would benefit from this. Individuals using the software would be happier and more confident in its reliability. Companies would also save money, improve their reputations, and gain an edge over their competitors.

Fixing bugs after releasing software is a vital task in software engineering. It impacts software developers and users. If we can find and fix bugs before releasing software, software development will improve significantly and become more reliable.

1.2 Objective

This research introduces a novel approach called predictive bug modeling. Using data analytics and machine learning, it aims to predict and resolve software bugs at the initial stages of coding. Instead of relying on traditional bug-fixing methods that address issues after they occur, this approach emphasizes proactive measures.

The objective is not just to improve code quality but to revolutionize software development by incorporating predictive bug modeling as an integral part of the process.

To improve software development, the emphasis is shifting away from merely fixing bugs after they occur. Instead, a proactive approach is being adopted to identify and prevent bugs before they can cause problems in released software. This approach is more efficient and uses resources more effectively.

1.3 Approach

This research aims to predict bugs on a file-level using various models, from simple ones like logistic regression to complex ones like neural networks and decision forests. Files are written in Python and C++, from them we extracted code-related metrics and fed them into these models. The results were analyzed from different angles to determine the most effective model for the project's objectives.

First of all it is important to say that all the data of the thesis are taken from a real project. In fact they are taken after a long internship in a factory that produces infotainment (the software of the on-board computer) for luxury car, differently from other works about this argument that product data artificially. This lead us to have a strong opportunity to improve a software in a practical way, but it also caused problems and issues. The first one was that there were only sixty-nine files in the project, so we devised a plan to create robust predictive models. We knew we needed more data, so we took several steps: first we added more data to the dataset and then we improved the way we build predictive models.

We started by augmenting our dataset with a technique to create more data samples: a Python script that matched the properties of the original dataset; it was important to overcome the limitations of our initial dataset generating synthetic data The goal was not just to add quantity but also to maintain the characteristics and distribution of the existing data. We carefully crafted the algorithms to ensure that the augmented dataset retained its integrity and usefulness.

Using the enhanced dataset, we embarked on the essential process of clustering and labeling. We employed clustering techniques to identify underlying patterns and structures in the data, enabling us to assign meaningful labels to each file. This was done grouping carefully similar files together based on their shared characteristics. By clustering and labeling the data, we brought organization and semantic meaning to it, which will aid in more effective model training and evaluation.

To build reliable predictive models, we implemented a strict data splitting method for training and testing. Our dataset was divided into two parts: a training set to develop the models and a test set to assess their performance. This separation allowed us to objectively evaluate our models on data they had not been trained on, protecting against overfitting and biased results. By following rigorous methodology, we ensured the robustness of our models and strengthened our belief in their predictive accuracy.

After preparing our data and training our machine learning models, we evaluated their performance to determine the best approach for our project. We used several algorithms, including logistic regression and neural networks. We carefully examined the performance of each model, using various metrics and cross-validation techniques to assess their effectiveness. This analysis helped us understand the advantages and limitations of each model and provided guidance for further improvement.

During the development of our AI models, we discovered the significance of using explainable AI techniques. These techniques allowed us to understand how our models make decisions. We went beyond the model predictions and investigated the relationships between different data features and the model's classifications. By combining interpretability and logical reasoning, we aimed to make our models more transparent. This provided stakeholders with valuable information about how the models function, helping them make informed decisions.

We addressed the challenges of data preparation, organization, training, and performance assessment through a rigorously designed approach. By leveraging data science principles and teamwork, we were able to unlock valuable insights from hidden data patterns, ultimately creating highly accurate prediction models that empower better decision-making.

1.4 Structure of the thesis

This thesis is divided into eight chapters: chapter 1 introduces the problem and describes the objective of the thesis, chapter 2 describes the background knowledge required for the thesis, chapter 3 goes in the details of the principal objectives of the thesis, facing first the problems that bugs can cause, chapter 4 analyzes the main other works on this topic, chapter 5 describes how it is faced the problem and how it is solved in details. then chapter 6 evaluates the project focusing on the fact if the main goals are reached or not, chapter 7 discuss about how the work is done focusing on positive and negative aspects and at the end chapter 8 concludes the thesis and focuses on related jobs that could be done citing this thesis.

Chapter 2

Background Information

2.1 Introduction

The whole work takes office in the context of the Agile Methodology, DevOps and automation. In this chapter we explain well the main differences between Agile Methodology and Classic Methodology and we face the concept of Quality Assurance, which we mentioned in the previous chapter, explaining how and when it is implicated. At the end of the chapter we will speak of the defect prediction, focusing on the modelling pipeline and on the possible granularity levels. Moreover we will face the theoretical basis of the project, for example the explanation of the models used and what is explainable AI and why it is useful.

2.1.1 Agile methodology

The agile methodology is a new way of organizing work within companies, in particular it is hugely used in companies working on software and programming. It is contrasted with the traditional methodologies, such as waterfall models or spiral models:

- The waterfall model (figure 2.1) is a traditional software development methodology that follows a linear and sequential approach. In this model, each phase of the software development process must be completed before the next phase begins. The phases typically include requirements gathering, system design, implementation, testing, deployment, and maintenance. Once a phase is completed, it is challenging to revisit or make changes, as the process flows in one direction, like a waterfall. This model is characterized by its structured and inflexible nature.

- The spiral model (figure 2.2) is an evolution of waterfall model, in fact it combines elements of waterfall model and iterative development. It was developed to overcome some of the limitations of the traditional waterfall model. In this model the software development process is divided into a series of cycles or ”spirals”. Each spiral represents a phase of the development process, such as planning, risk analysis, engineering, testing, and evaluation. The key feature of the spiral model is its

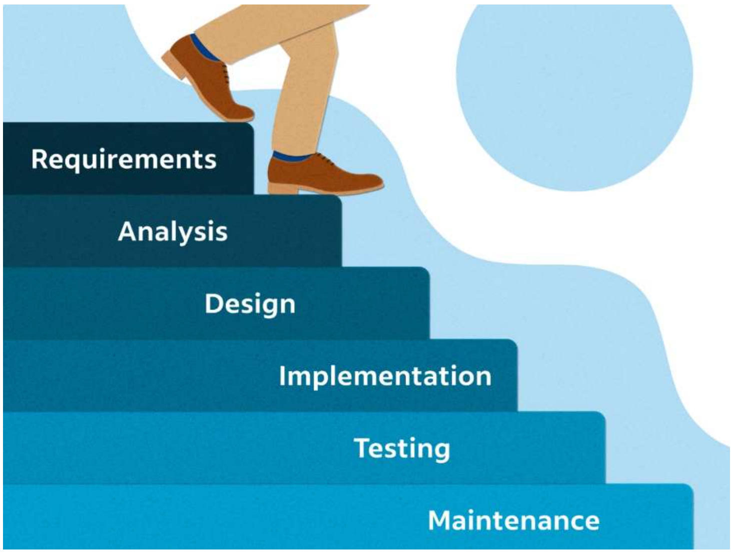


Figure 2.1: Example of a waterfall model

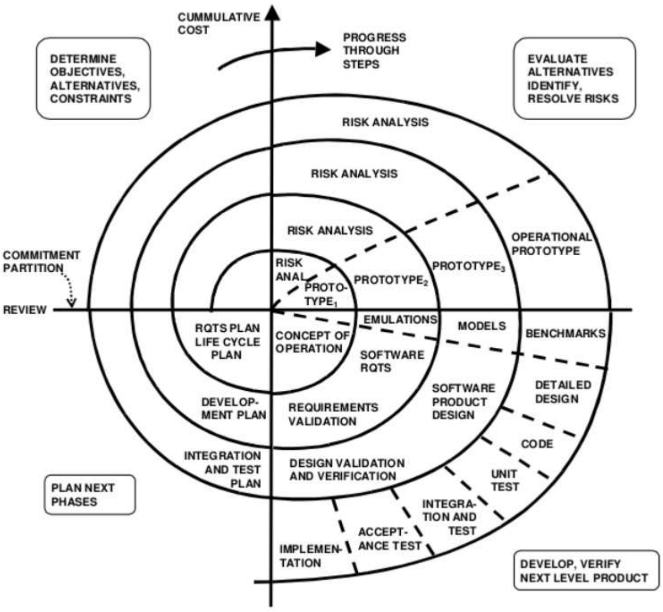


Figure 2.2: Example of a spiral model

emphasis on risk analysis and management. In particular the spiral model is based on some key principles:

- **Planning:** The project starts with the initial planning phase, where the work group defines goals and requirements.
- **Risk Analysis:** In this phase, the decision-making group identifies potential risks and challenges. Strategies are developed to mitigate and manage these risks.
- **Engineering:** The actual development and coding of the software take place. This phase is similar to the implementation phase in the waterfall model. The development department work on this stage.
- **Testing:** After each spiral, testing is performed to ensure that the software meets the specified requirements and functions correctly. In particular the validation department works on this phase.
- **Evaluation:** At the end of each spiral, an evaluation is conducted to review the progress and determine whether the project should proceed to the next spiral or be modified or terminated.

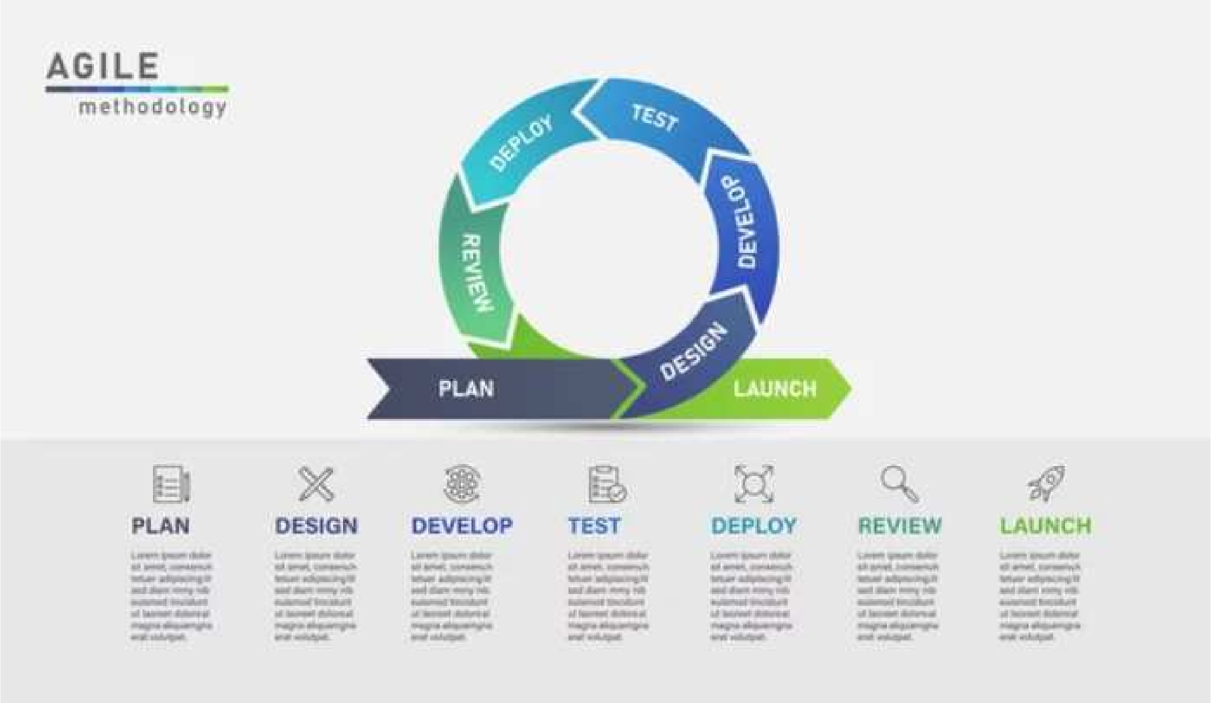


Figure 2.3: Example of Agile Methodology

The agile methodology (example on figure 2.3) involves a continuous exchange of information among all parties involved. The goal is not merely the sale of the product but the satisfaction of the customer and everyone working on the project. The work is done in teams, not individually, and there are continuous releases. It is characterized by iterative and incremental development, where work is divided into smaller units known as "iterations" or "sprints." The agile methodologies' aim is to deliver functional software quickly and adapt to changing requirements. As written above, agile projects are divided into short iterations or sprints, typically lasting two to four weeks. At the end of each iteration, a ready product increment is delivered. It is very important the customer collaboration, in fact agile teams work closely with customers and stakeholders to understand their needs and gather feedback throughout the development process and there are continuous feedback between the parts involved, to identify and address issues early in the development cycle. Teams must be flexible and open to changing requirements, even late in the development process because changes are considered as opportunities for improvement; in particular work is prioritized based on customer value, and the highest-priority items are tackled first and it may change according to the customer's requests. The most important difference between agile methodology and traditional methodologies is on the production of the documentation: while in traditional methodologies it was heavy, in the agile methodologies it is lighter because agile methodologies prioritize working software over comprehensive documentation. In particular the documentation, like the Doxygen package cited above, is largely produced automatically by the system and it is more lightweight.

2.1.2 DevOps

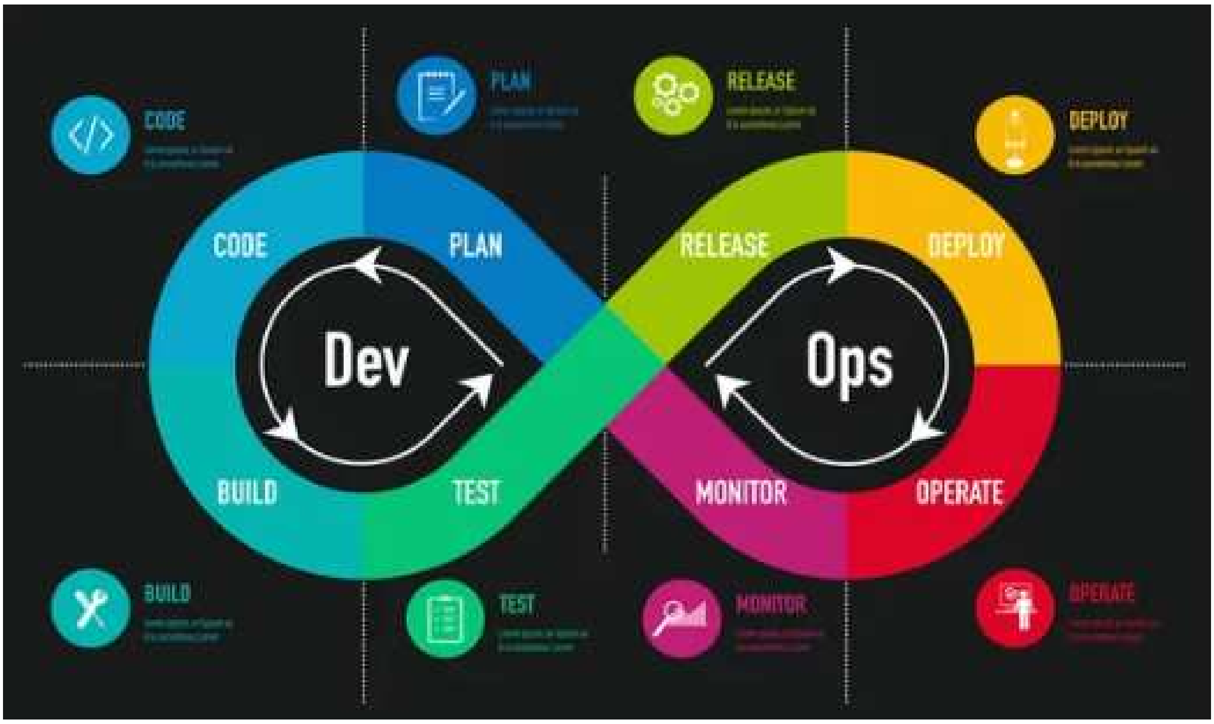


Figure 2.4: Example of DevOps

The most famous agile methodology used is DevOps (in figure 2.4). It is a set of practices and principles whose aim is to enhance collaboration and communication between the development (Dev) and IT operations (Ops) teams within an organization. It tries to automate the processes involved in software development, testing, deployment, and infrastructure management to deliver high-quality software in a more efficient and precise way. For DevOps automation is very important, in particular the repetitive and manual tasks, such as code deployment, testing, and infrastructure provisioning, to reduce errors and save time. It is facilitated through the continuous integration of code changes into a shared repository, allowing for early detection of integration issues and faster development cycles and through the continuous delivery, that is the ability to automatically and consistently deploy code changes to production or other environments, reducing the time between development and delivery. Automation testings (continuous integration and continuous delivery, shortly CI/CD) makes easier the monitoring of applications and infrastructure, combined with feedback loops, to detect and address issues quickly. It is also important to incorporate security practices into the DevOps process to identify and address vulnerabilities early in the development life-cycle. It is very important the use of Git, a web platform that allows the creation of public and private repositories where developers can upload their code and manage changes to different versions concurrently, encouraging and making easier work in team. In fact it is possible to work simultaneously with others on the same project without generating conflicts, upload one's work to his branch and then proceeding with a merge request to the master branch. This helps the tracking issues because before every merge request everyone can check the comparison between the old version and the updated one. In the following experiments it is used Gitlab as Git and the automatized pipeline is created by a file .yml. It is divided in stages, some of them consequential and some of them independent by the others; the stages are usually divided in jobs. The file .yml is written in bash and it uses docker to create the pipeline. Docker is fundamental for the DevOps technology because it offers a more agile and efficient system architecture thanks to his divisions in container that are initiated by the dockerfile. Moreover Docker is executed by computer and remote virtual machines, so that it does not stop the other's people work.

2.2 Software Quality Assurance

The set of processes and activities designed to ensure that software is developed and delivered with a quality suitable for its intended purpose is called software quality assurance (SQA). This field, as written above, focuses on defect prevention, on monitoring, and on continuous improvement of software development processes to ensure that the final product satisfies established quality standards. So Software Quality Assurance is fundamental on realizing software and it must be embedded as a quality culture throughout the life cycles from planning to release so teams can follow best practices to prevent defects, rather than wasting time detecting them. However with modern software development practices several difficulties can be found to apply the current SQA practices. Since the team work is encouraged, everyone is responsible of the quality, so developers have to test their own code, and this is done creating unit tests to perform an initial screening of the most obvious errors and to receive notifications quickly during the software development life cycle in case of any breakages. Usually the minimum percentage of code that must

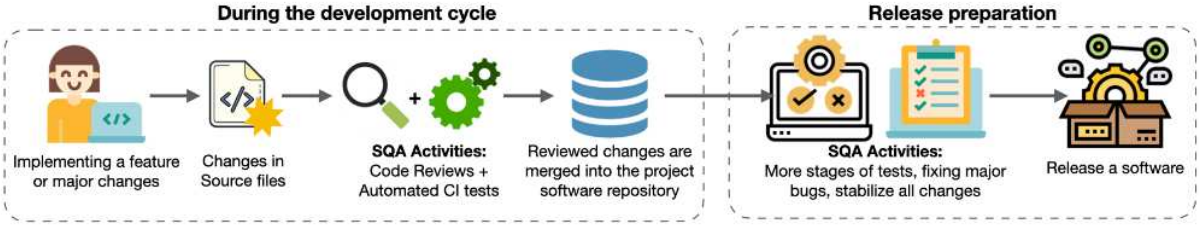

During the development cycle
Release preparation
Implementing a feature or major changes
Changes in Source files
SQA Activities: Code Reviews + Automated CI tests
Reviewed changes are merged into the project software repository
SQA Activities: More stages of tests, fixing major bugs, stabilize all changes
Release a software

Figure 2.5: An overview of the software quality assurance activities (from [1])

be covered by unit tests is eighty percent, possibly well distributed in the files. In the merge request, as it is written above, the success percentage of unit test is also compared, so this is one of the parameters for accepting or rejecting the merge request. By the way this process of static analysis does not guarantee a defect-free software product because it produces too many false positives or false alarms and it does not identify the problematic areas that may present bugs after the software release nor offer any practical advice on how to reduce risk; so it has to be integrated with some other bug prevention and detection techniques, because otherwise teams may release poor quality software product to costumers and it could cause slow project progress and high cost development, in terms of money and time.

2.2.1 SQA activities during the development stage

During the development stage, developers implement new functions and other modifications to the source code. This commit has to be submitted to SQA activities, for example continuous integration and code review, before the developers do a merge request with the principal branch. Since these commit-level SQA activities are time-consuming, it has been proposal the Just-In-Time defect preview to support developers prioritizing their limited SQA effort to modify the parts of code that more probably will present post-release defects. However, JIT defect prediction only early detects defect-inducing changes, rather than post-release defects. Despite the SQA activities work during the develop cycle, for example code revision, it is still possible that some post-release defect remain also after the release of the software product. So the SQA activities are still necessary during the release preparation.

2.2.2 SQA activities during the release preparation

During the release preparation is important to execute SQA activities to ensure that the software product is ready for release and to reduce the probability that the software product will have post-release defects. In this phase all the files that are changed during the development have to be checked to ensure that these changes will not get worse the global quality of the software products; in fact so many SQA activities, for example regression tests or manual tests are done by the validation department. However it is almost impossible to check and verify that the software product does not present defect with SQA activities in an exhaustive way, since the code is too big and there are limited resources, for example from a monetary and time perspective especially in rapid-release development practices. To help the developers to canalize resources in the optimal way it is important to identify what are the most defect-prone areas of source code that could present post-release defects. Moreover it is advantageous to obtain in real time estimation of the areas of the code source where it is more likely to find defects to help the develop team to have a more profitable management of SQA resources. Menzies et al. [2] mentioned that software contractors tend to prioritize their effort on reviewing software modules tend to be fault-prone. A case study at ST-Ericsson in Lund, Sweden by Engstrom et al. [3] found that the selection of regression test cases guided by the defect-proneness of files is more efficient than the manual selection approaches. At the Tizen-wearable project by Samsung Electronics [4], they found that prioritizing APIs based on their defect-proneness increases the number of discovered defects and reduces the cost required for executing test cases.

2.2.3 How to calculate SQA

Now we are going to face the activities that it encompasses the parameters and the methods that are used to calculate SQA. We start from the activities that it encompasses:

- Planning and management: This involves defining the quality goals for the software project and creating a plan to achieve them.

- Development: This involves creating the software product, including coding, testing, and debugging.

- Deployment: This involves making the software product available to users.

- Maintenance: This involves fixing bugs, adding new features, and updating the software product over time.

SQA parameters are the characteristics that are used to measure the quality of a software product. These parameters can be divided into two main categories: functional parameters that relate to the functionality of the software product, such as its correctness, reliability, efficiency, and usability and non-functional parameters that relate to other aspects of the software product, such as its performance, security, and maintainability.

SQA is calculated using a variety of methods, including:

- Inspections: This involves manually reviewing the software product to identify potential defects.

- Testing: This involves executing the software product to find and fix defects.

- Metrics: This involves collecting data about the software product to measure its quality.

By the way the specific SQA parameters and calculation methods that are used will vary depending on the specific software project. However, the overall goal of SQA is to ensure that the software product meets the needs of its users and is of high quality. We can see an example of the possible SQA certifications that could be taken in figure 2.6

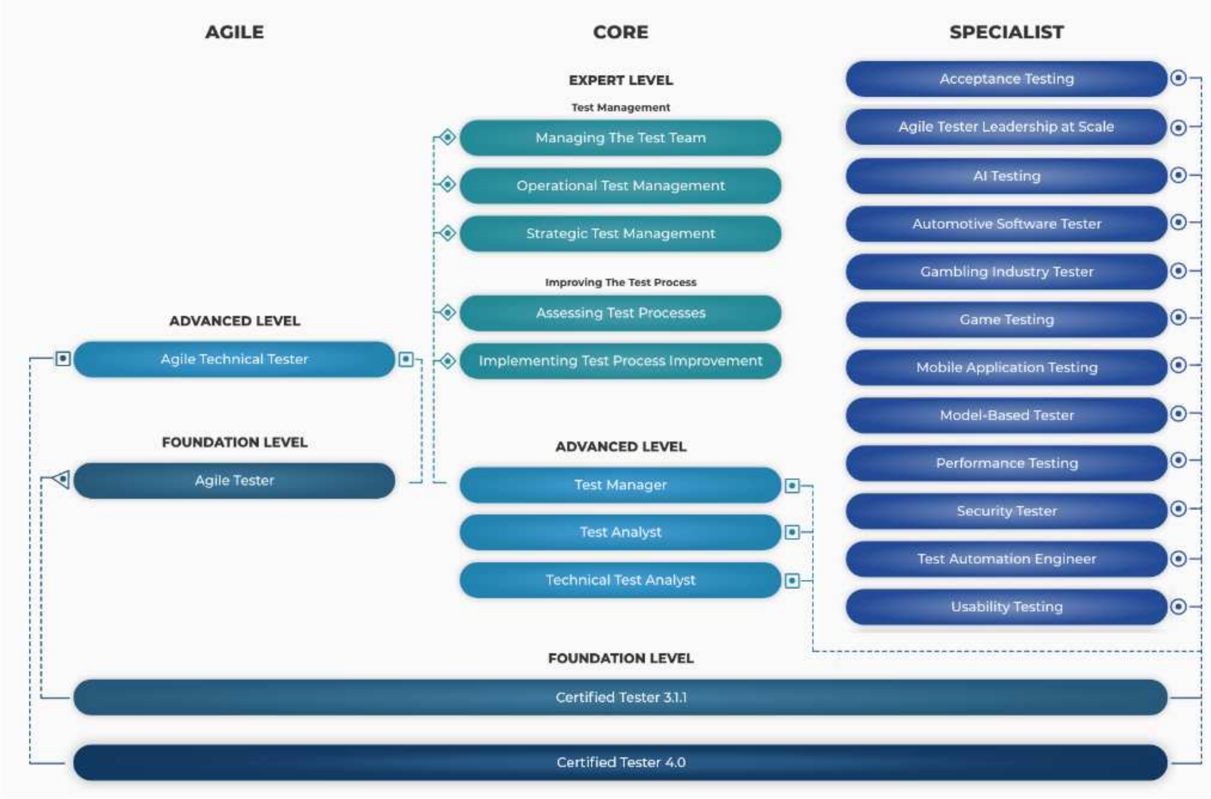


Figure 2.6: Example of SQA certifications

2.3 Defect prediction

Several models of defect prevention have been created to anticipate which are the areas of the source code that present a higher probability of having post-release defects. A model of defect prevention is a classification model which gives an estimation of the probability that a file presents a post-release defect. As it is written above, one of the principal aim of the model of defect prevention is to help the developers to spend the limited SQA resources on the code areas that present the higher probability of presenting post-release defects in the most profitable way.

2.3.1 The modelling pipeline of defect prediction models

The predictive accuracy of the model of defect prediction is based on the modelling pipelines of defect prediction models. To predict precisely the source code areas which more likely present defects, there are many past studies that conducted a complete evaluation to identify the best modelling pipeline technique of defect prediction models. For example there are feature selection techniques, collinearity analysis, class rebalancing techniques, classification techniques, parameter optimization, model validation and model interpretation. Despite the recent progress on modelling pipelines for defect prevention models, the cost-effectiveness of the SQA resources is still based on granularity of prediction.

2.3.2 The granularity levels of defect predictions models

As it is written above, the cost-effectiveness of the SQA resources depends on the granularity levels of defect prediction. Previous studies sustained that prioritizing software modules at the finer granularity is more cost-effective. For example Kamei et al [5] found out that file-level defect prediction is more effective than package-level defect prediction; Hata et al. [6] found out instead that method-level defect prediction is more effective than file-level defect prediction. There are models of defect predictions at various granularity levels, for example packages, components, modules, files and methods; however the actual granularity is still too coarse-grained, so developers could still need to waste some SQA effort on manually identifying the most risky lines. For this reason defect prediction is useful to SQA team to spend SQA effort in an optimal way to analyze and identify defects.

2.4 The models

In this section we will describe the different models that are compared in the thesis. Of course they are not the only models that could be used, but the choice was to take these because they are the most common and the easiest to work on.

Before explaining how each model works, it is important to divide them in two families: interpretable models and black box models. Logistic regression and decision tree are interpretable models, in fact it is possible to understand and interpret how the model makes decisions and generates predictions. On the other hand Random Forests, Neural Networks, Gradient boosting Machine and Extreme Gradient Boosting Tree are black box models and so we do not know how they make decision and how they work. To solve this issue we will speak later about the algorithms of Explainable AI to see how they work and to have better feedbacks from them.

2.4.1 Logistic regression

Logistic regression is a statistical method used primarily for binary classification tasks, where the goal is to predict whether an observation belongs to one of two possible categories, in our case if the file presents bugs or not. Unlike linear regression, which is suited for predicting continuous outcomes, as we can see in figure 2.7, logistic regression is tailored specifically for handling categorical outcomes.

At the heart of logistic regression lies the logistic or sigmoid function, which transforms the linear combination of input features and model coefficients into a probability score between 0 and 1. This sigmoid function is pivotal in logistic regression as it allows us to model the probability of an observation belonging to the positive class, in our case the probability of a file to be bug-free.

Before applying the sigmoid function, logistic regression computes the linear combination of input features and model coefficients: $z = \beta_0 + \beta_1 * x_1 + \beta_2 * x_2 + ... + \beta_n * x_n$, where z is the linear combination, β_0 is the intercept term and $\beta_1, \beta_2, ..., \beta_n$ are the coefficients associated with each feature $x_1, x_2, ..., x_n$.

At the end the logistic regression works like this: $\mathbf{P}(Y = 1|X) = \sigma * z$, where $\mathbf{P}(Y =$

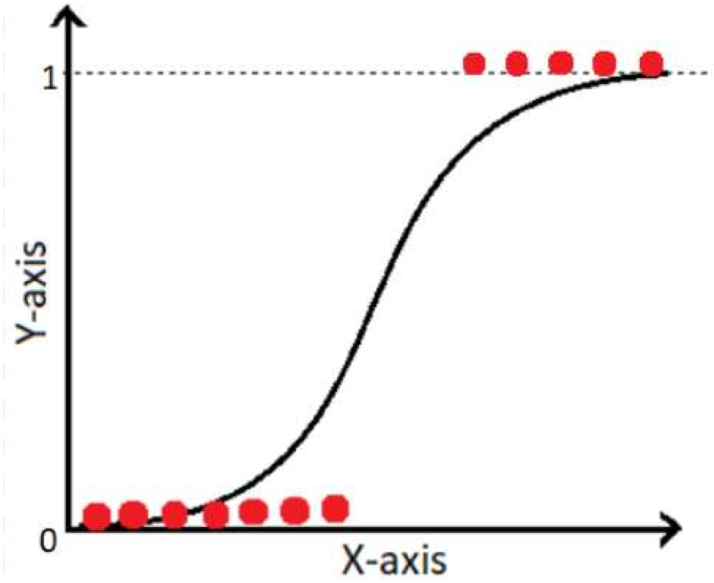


Figure 2.7: Example of a graph of logistic regression

$1|X)$ represents the probability of the dependent variable (Y) taking the value 1 given the independent variables (X), σ is the sigmoid function cited previously and z is the linear combination of input features and model coefficients.

Then there is the training phase, where logistic regression learns the optimal values for these coefficients by minimizing the discrepancy between the predicted probabilities and the actual class labels in the training data. This process usually involves iterative optimization techniques such as gradient of Newton descent.

Once the model is trained, it can be used to predict the probability of an observation belonging to the positive class based on its input features. These predicted probabilities can then be converted into binary class labels using a predefined threshold. If the predicted probability exceeds the threshold, the observation is classified as belonging to the positive class; otherwise, it is classified as belonging to the negative class.

In summary, logistic regression is a versatile and hugely used algorithm for binary classification tasks, but it cannot be used for non-binary classification tasks. Its ability to model nonlinear relationships and predict probabilities makes it invaluable in various domains, including healthcare, finance, marketing and of course bug prediction. Understanding the underlying principles of logistic regression is essential for effectively applying and interpreting its results in real-world scenarios.

2.4.2 Decision tree

A decision tree (figure 2.8) is a hierarchical decision support model that represents decisions and their potential outcomes in a tree-like structure. This includes considering chance event results, resource expenses, and utility. It serves as a visualization of an algorithm comprising solely conditional control statements. Now we are going to enter in the details in how it works.

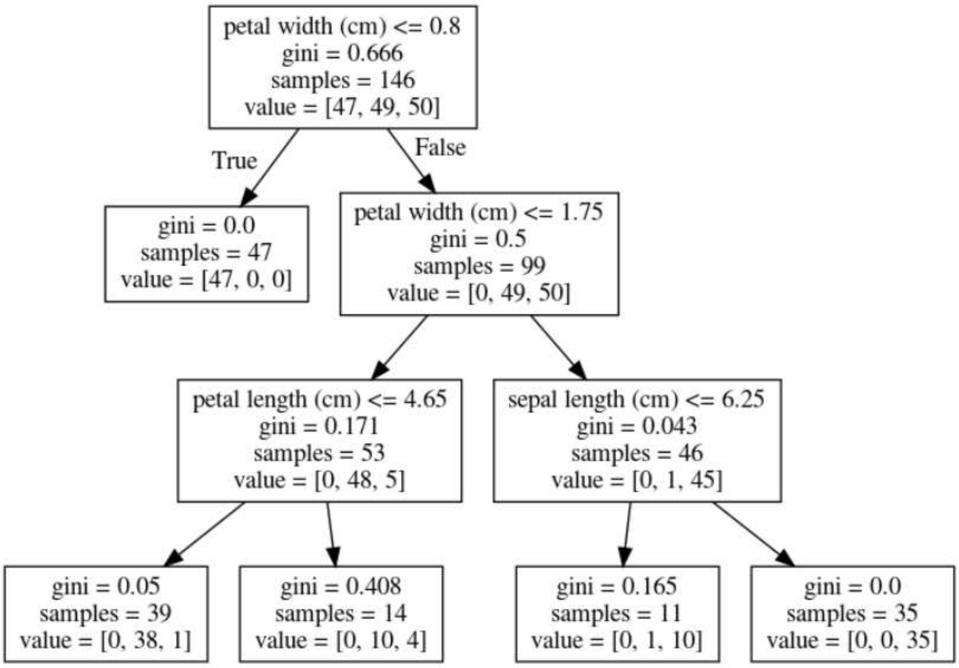


Figure 2.8: Example of Decision Tree

The tree begins at the root node, representing the entire dataset, then it splits the data into subsets based on feature values; this process is repeated recursively until certain stopping criteria are met. At each non-terminal node (decision node), a decision is made based on a specific feature's value, determining the path through the tree for each data point. The process continues until reaching a stopping criterion, that could be maximum depth, minimum samples per node, or achieving a pure node. At this point, the node becomes a leaf node, providing the final prediction or decision. Various criteria guide the splitting process, including Gini impurity, entropy, and information gain, assessing node impurity or uncertainty. To prevent overfitting, tree pruning techniques are applied post-construction, removing nodes that do not enhance predictive performance. Once constructed, the tree can predict outcomes for new data points by traversing from the root to a leaf node based on their feature values.

In summary, decision trees offer a balance of power and interpretability, making them widely applicable across domains due to their transparent nature and ability to handle diverse data types.

2.4.3 Random Forest

Random forests (example in figure 2.9) have emerged as a prominent technique in machine learning, renowned for their versatility and robustness. Unlike traditional models that rely on a single approach, random forests leverage the collective wisdom of an ensemble of decision trees. This essay delves into the core principles behind random forests, exploring how they harness diversity and randomness to construct powerful models.

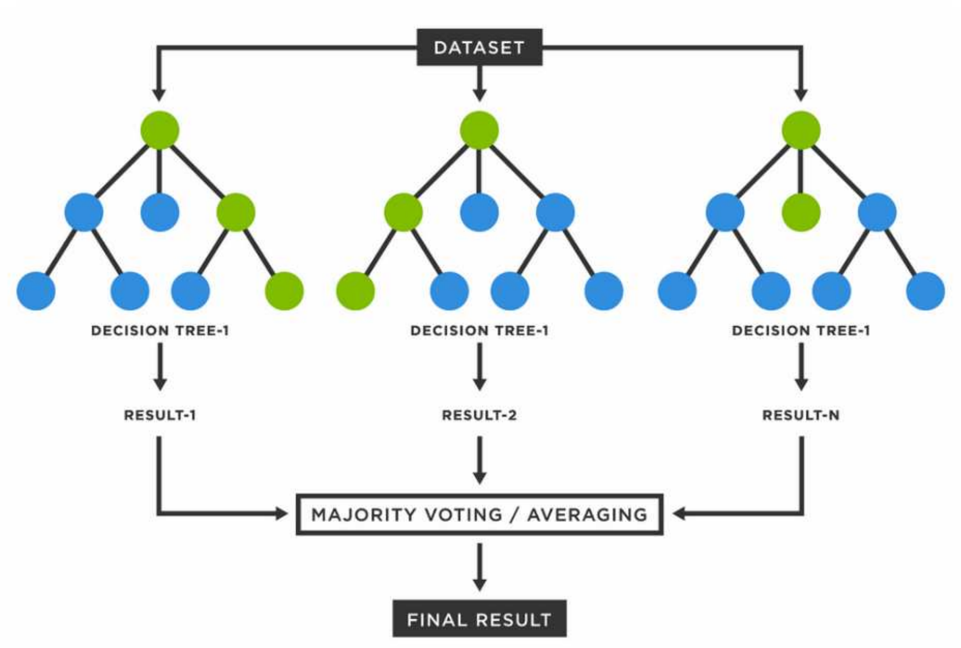


Figure 2.9: Example of Random Forest

At the heart of a random forest lies the concept of ensemble learning. Instead of a solitary model, a random forest meticulously cultivates a multitude of decision trees. Each tree operates independently, analyzing the data through a unique lens. To ensure diversity within the ensemble, random forests incorporate a touch of randomness during the training process. This manifests in two ways:

- Subsetting: Each tree is presented with a random subset of the training data. This prevents any one tree from becoming overly reliant on the entire dataset and fosters a wider range of perspectives within the ensemble.

- Feature Selection: At each branching point within a tree, a random selection of features (data attributes) is considered for splitting the data. This further diversifies the exploration paths taken by individual trees and helps prevent overfitting, a common pitfall in machine learning. The learning process within a random forest unfolds iteratively. Each tree independently partitions the data based on the chosen features, aiming to isolate regions with similar characteristics. This process continues until a predefined stopping criterion is met, ensuring the trees maintain a manageable complexity.

Once the forest has matured, individual trees don't directly make predictions. Instead, random forests leverage the power of collective intelligence. When presented with a new data point, each tree casts a vote, predicting the most likely class or value based on its unique exploration path. The final prediction is then determined by aggregating these individual votes, often through a majority rule or averaging approach. This ensemble approach, akin to combining the insights of multiple experts, leads to more robust and accurate predictions compared to a single decision tree.

The advantages of random forests are multifaceted. By design, they are less prone to overfitting compared to single decision trees. The randomness injected during training prevents the ensemble from becoming overly swayed by specific patterns within the training data, leading to models that generalize well to unseen data. Additionally, random forests can effectively handle problems with a high number of features due to their reliance on random subsets during training.

In conclusion, random forests offer a powerful and versatile approach to machine learning. Their ensemble learning strategy, combined with the strategic use of randomness, empowers them to tackle complex problems with remarkable accuracy and robustness. As research delves deeper into the intricacies of random forests, they hold immense promise for unlocking the potential hidden within data across various scientific disciplines.

2.4.4 Gradient Boosting Machine

Gradient boosting machines (GBMs) are a powerful machine learning technique that combines the strengths of multiple weak models to create a single strong model. (figure 2.10).

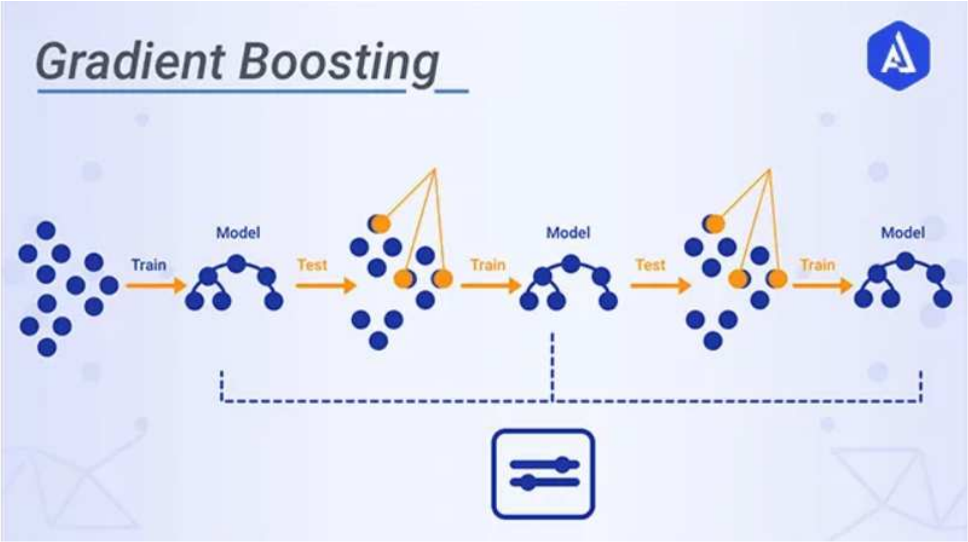


Figure 2.10: Example of Gradient Boosting Machine

Imagine a student faced with a daunting mountain of knowledge – mastering the intricacies of machine learning. Traditional methods present a solitary path, forcing the student to grapple with the complexities of a single model. But what if there was a way to learn through collaboration, leveraging the strengths of multiple perspectives? Gradient boosting machines (GBMs) offer precisely this approach, emerging as a powerful ensemble learning technique within the machine learning landscape.

This thesis delves into the fascinating world of GBMs, exploring how they orchestrate a team of weak learners to construct robust and accurate models. We shall embark on a journey to understand their core principles, unveiling the sequential learning strategy that empowers them to tackle intricate problems that might baffle individual models.

At the heart of GBMs lies the philosophy of ensemble learning. Unlike traditional methods that rely on a single model to learn patterns within data, GBMs embrace a more collaborative approach. They meticulously construct an ensemble of weak learners, typically decision trees, where each learner builds upon the knowledge of its predecessor. This sequential learning process allows the ensemble to progressively refine its understanding of the data, akin to a group of students collectively deciphering a complex concept.

The learning process within a GBM unfolds in a captivating dance. The ensemble commences with a rudimentary model, often a shallow decision tree. This initial model makes a preliminary prediction on the data, akin to a student offering a tentative explanation. However, the learning doesn't stop there. GBMs meticulously analyze the discrepancies between the model's predictions and the actual values, uncovering the areas where the model faltered. These discrepancies, termed residuals, become the cornerstone of the next stage.

Fueled by the insights gleaned from the residuals, a new decision tree emerges. This new learner acts as a dedicated tutor, focusing on rectifying the errors made by the previous model. By meticulously studying the areas of weakness, the new tree refines the overall understanding of the data. This iterative process continues, with each new decision tree building upon the accumulated knowledge of its predecessors, akin to students sharing their insights and collectively deepening their comprehension.

The brilliance of GBMs lies in their ability to leverage the strengths of each individual learner within the ensemble. Each decision tree, though weak on its own, contributes a specific piece to the puzzle. Through a process of aggregation, the predictions from these individual trees are combined, culminating in a robust and accurate ensemble model. This final model, much like a group of students who have mastered the subject through collaboration, surpasses the capabilities of any single learner.

The advantages of employing GBMs are manifold. By harnessing the collective intelligence of the ensemble, they can capture intricate relationships within data that might elude a solitary model. This collaborative approach often leads to more accurate and reliable predictions, a boon for researchers and practitioners alike. Furthermore, the flexibility of GBMs allows them to adapt to diverse tasks. By selecting different base learners and loss functions, they can be tailored to a wide range of problems, showcasing their versatility across various domains.

In conclusion, gradient boosting machines offer a compelling alternative to traditional machine learning methods. Their sequential learning strategy and ensemble approach empower them to tackle complex problems with remarkable accuracy and flexibility. As we delve deeper into the captivating world of GBMs, we unlock a powerful tool for unlocking the secrets hidden within data, paving the way for advancements in various scientific disciplines.

2.4.5 Neural Network

Artificial neural networks (ANNs) are a strong tool in machine learning. They are inspired (as the name suggests) to the structure and the function of the human brain to work on complex tasks. Now we are going to delve into the core principles behind ANNs, exploring

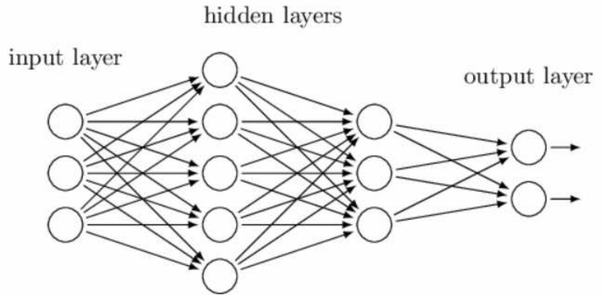


Figure 2.11: Example of a Neural Network

how these network of interconnected nodes process information and learn from data.

First of all there is a big difference with the traditional machine learning algorithms with predefined models: Artificial Neural Networks adopt a more flexible approach, in fact they are built upon a network of interconnected processing units called artificial neurons. These artificial neurons are organized in layers, with information flowing from the input layer through hidden layers, arriving at the end of the process to the output layer, we can see a scheme of a Neural Network on figure 2.11.

Each artificial neuron is equipped with weights and biases that influence how it processes incoming signals. During the training process, these weights are adjusted, basing on the errors between the network's predictions and the actual outcomes. ANNs iteratively update their internal parameters to minimize these errors, gradually improving their performance on the task after the other. In this way ANNs are very flexible and are very able to learn in a fast and reliable way.

Then, the input layer receives data points, which are transformed by activation functions living within artificial neurons. These activation functions introduce into the network non-linearity elements, making the network able to capture relationships within the data that linear models may not look. Progressing through the hidden layers, the data is transformed by the network, according to the value of weights and biases of the neurons.

The training process in ANNs often involves a technique called back propagation. This algorithm calculates the gradients of the error function with respect to each weight and bias in the network. The gradients essentially indicate how much each parameter contributes to the overall error and by adjusting the weights and biases in the opposite direction of the gradients, the network gradually learns to minimize the error and improve its predictions.

ANNs are very good at learning complex patterns from data and this power makes them appropriate for a wide range of tasks; moreover, ANNs can handle high-dimensional data in a very effective way, making them a valuable tool for researchers working with intricate and big datasets.

In conclusion, artificial neural networks offer a powerful and versatile approach to machine learning. Their great ability to learn from data and capture complex relationships has revolutionized various fields. As research delves deeper into the intricacies of ANN architectures and training algorithms, they hold immense promise for unlocking new frontiers in artificial intelligence and beyond.

2.4.6 Extreme Gradient Boosting Tree

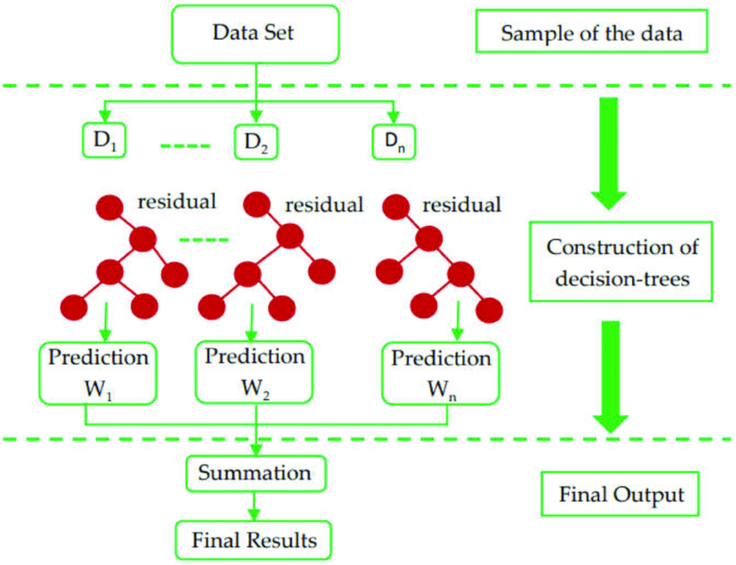


Figure 2.12: Example of eXtreme Gradient Boosting Tree from [7]

Extreme Gradient Boosting (XGB) Tree is a very important technique in machine learning, renowned for its remarkable accuracy and robustness in tackling complex tasks. Now we are going to describe the core principles behind XGBoosting, exploring its sequential learning strategy and ensemble approach.

Unlike traditional models that learn in isolation, Extreme Gradient boosting leverages the collective wisdom of an ensemble. However, it departs from conventional random forests by adopting a more strategic approach to ensemble construction (figure 2.12). XGboosting meticulously builds its ensemble in a stage-wise manner, with each new model (typically a decision tree) focusing on improving the predictions of the previous ones. In fact XGboosting starts with a simple model, often a shallow decision tree, that makes an initial prediction on the data. But the learning doesn't stop there.

XGboosting analyzes the discrepancies between the model's predictions and the actual values. These discrepancies, termed residuals, highlight areas where the initial model failed; XGboosting then harnesses the power of these residuals to create a new decision tree that targets the errors made by the previous model, with the scope to refine the overall understanding of the data.

The brilliance of XGboosting lies in its focus on boosting. Each new decision tree, instead of being built from scratch, leverages the knowledge accumulated by the ensemble so far. This sequential approach ensures that the ensemble progressively improves its performance with each iteration.

However, XGboosting doesn't simply add the predictions from each tree. It meticulously assigns weights to them based on their individual performance. This ensures that trees that contribute more accurate predictions have a greater influence on the final ensemble output.

The advantages of XGboosting are multifaceted. By harnessing the power of ensembles and focusing on boosting, it achieves superior accuracy compared to single models. Additionally, XGboosting incorporates efficient algorithms for tree construction and split selection, making it computationally efficient and well-suited for large datasets. Furthermore, XGboosting offers built-in mechanisms to prevent overfitting, a common pitfall in machine learning.

In conclusion, Extreme Gradient Boosting stands as a powerful and versatile tool in the machine learning landscape. Its sequential ensemble learning strategy, combined with a focus on boosting and efficient algorithms, empowers it to tackle complex problems with remarkable accuracy and robustness. As research delves deeper into optimizing XGboosting architectures and exploring its applications, it holds immense promise for unlocking breakthroughs in various scientific endeavors.

2.5 Explainable AI

As it is written in the previous chapter, we used some model that are clearly explainable, that everybody can comprehend how they works and why they give some results and they are contrapposed to black box models, that work in a misterious way and we cannot comprehend why they give their output. For this reason Explainable AI (XAI) is important: it is a set of techniques and methodologies that aim to shed light on the inner workings of a model. It helps us understand how the model arrives at its predictions by providing insights into the features used by the model and their contributions to the final outcome.

2.5.1 What is Explainable AI and why it is important

There are several reasons why XAI is becoming increasingly important: first of all trust and transparency is crucial in many applications to understand why a model makes a certain decision. XAI helps build trust in AI systems by providing explanations for their actions. Moreover by understanding how features contribute to predictions, XAI can help identify potential biases or weaknesses in a model, leading to improvements in its performance. Moreover it can help uncover potential biases in the model's predictions by analyzing how features from different subgroups contribute to the output. This is crucial for ensuring fair and ethical use of AI systems. Finally in some industries, regulations are emerging that require AI systems to be explainable. XAI helps ensure compliance with these regulations.

2.6 Techniques of Explainable AI

There are various XAI techniques available, each with its own strengths and weaknesses and now we are going to explore them.

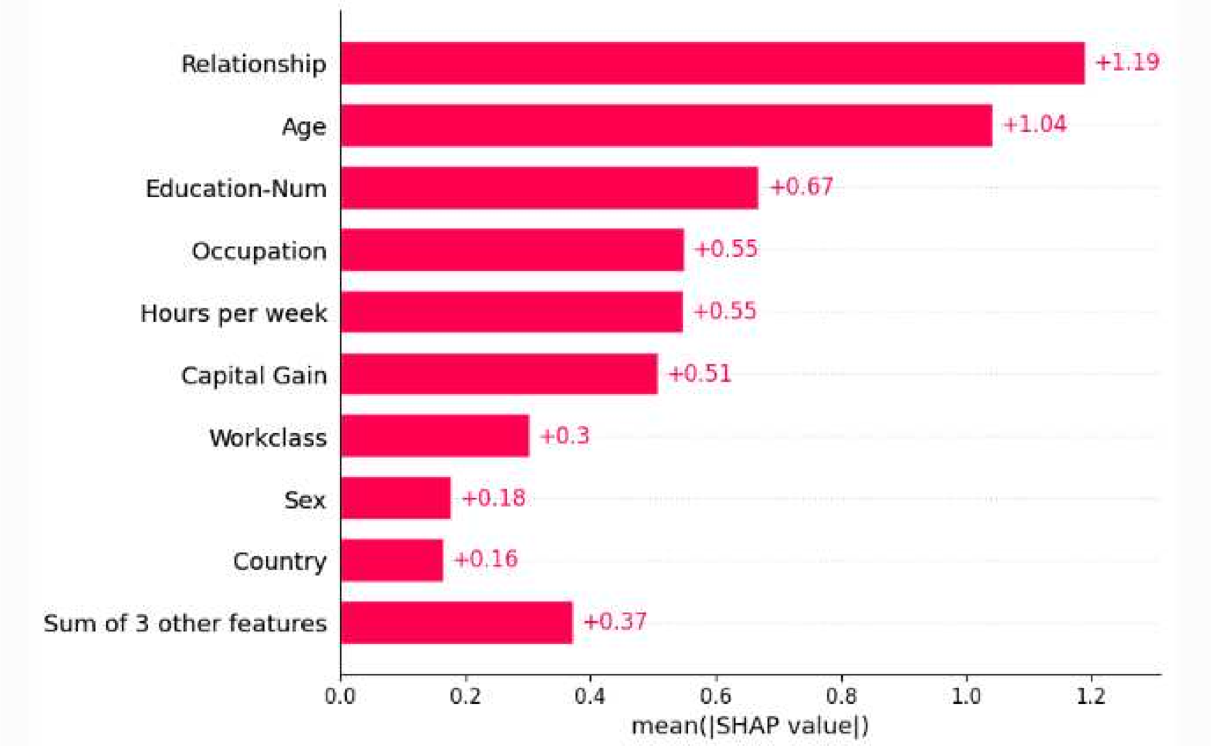


Figure 2.13: Example of SHAP

SHAP

SHAP, which stands for SHapley Additive exPlanations, is a specific technique used in Explainable AI (XAI) to understand the inner workings of complex machine learning models. It helps us interpret why a model makes a particular prediction by calculating the contribution of each feature to the final outcome. We can see an example on figure 2.13.

SHAP leverages concepts from game theory to explain individual predictions. It calculates a value (SHAP value) for each feature in a prediction, representing its contribution to the final outcome. It essentially asks: "If we removed this feature from the data point, how much would the model's prediction change?"

SHAP estimates the impact of each feature on the model's prediction. It first calculates the local value with the following formula: $SHAP_i(x) = \mathbb{E}[f(x_i, \hat{x}_{-i}) - f(\hat{x})]$, where $f(x)$ is the prediction function of the model, x_i is the value of feature x for data point i, $\hat{x}_{-i}$ is the vector of features for data point i with feature x replaced by its average value and $\mathbb{E}$ is the expected value operator. By calculating all the local SHAP values, we can calculate the global SHAP value, that for a feature x is defined as $SHAP(x) = \sum_{i=1}^{N} SHAP_i(x)$, where N is the number of data points.

Positive SHAP values indicate that the feature increases the model's prediction, while in the opposite way negative SHAP values indicate that the feature increases the model's prediction. Furthermore a high SHAP value for a feature indicates it significantly influenced the prediction, while a low value suggests minimal influence. Moreover SHAP can

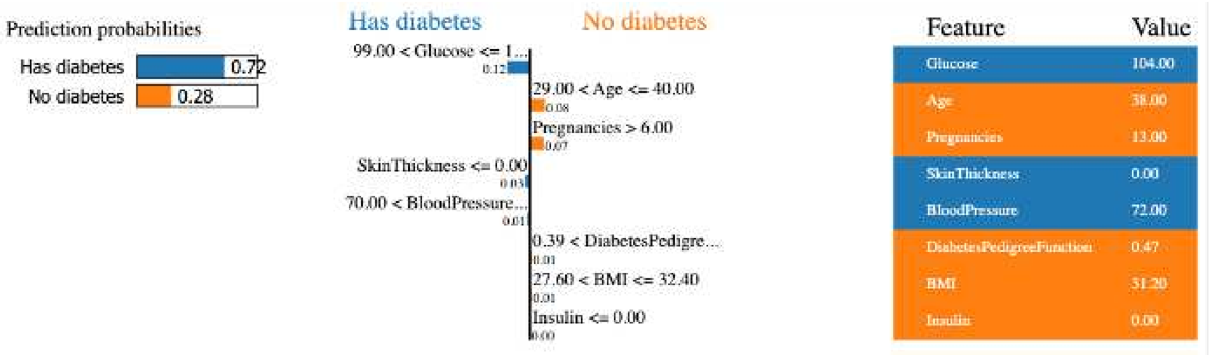


Figure 2.14: Example of LIME

provide both local and global explanations: local explanations break down the reasons behind a specific prediction for a single data point while global explanations offer an overview of how different features generally contribute to the model's overall behaviour. To show the results, SHAP uses various visualization techniques like force plots and dependence plots to illustrate the feature contributions and their interactions. These visualizations make it easier to grasp the reasoning behind the model's decisions.

Using SHAP is very convenient for various reason: first of all by understanding the contributions of individual features, users can have greater confidence in the model's decisions and identify potential biases, moreover insights from SHAP explanations can help to identify features with unexpected behavior or redundant information, aiding in model refinement and XAI with SHAP can help uncover potential biases in the model's predictions by analyzing feature contributions from different subgroups. Overall, SHAP is a powerful tool within XAI, providing valuable insights into how complex models make decisions. This understanding fosters trust, enables better decision-making based on model outputs, and facilitates model improvement.

LIME

LIME, which stands for Local Interpretable Model-agnostic Explanations, is another technique used in Explainable AI to understand how machine learning models make predictions. Unlike SHAP, which focuses on explaining individual feature contributions, LIME explains a model's prediction for a specific data point by creating a simpler, interpretable model locally around that point. Here on figure 2.14.

LIME tackles interpretability by focusing on local explanations. For a particular data point (instance) you're interested in, LIME creates a new, interpretable model (often a linear model) that approximates the behavior of the original complex model only around that specific data point. This local model helps explain why the original model made the prediction it did for that particular instance.

LIME generates a set of alternative explanations for the data point by slightly modifying its features (e.g., removing or changing feature values); then it assigns weights to these explanations based on how well they agree with the original model's prediction for that data point. Finally, LIME builds a simple interpretable model (usually a decision tree or linear model) that uses the weighted explanations to approximate the original model's behaviour around the data point of interest.

First of all LIME can be applied to any type of machine learning model, regardless of its internal workings. This makes it a versatile tool for XAI; moreover LIME explanations are specific to a single data point, providing insights into why the model made a particular prediction for that instance. Another key aspect is that the local model generated by LIME is often simpler and easier to understand than the original complex model, making it easier to grasp the reasoning behind the prediction. Overall, LIME offers a valuable approach to XAI by providing local explanations for individual predictions. This helps users understand how the model makes decisions for specific data points, fostering trust and enabling better decision-making.

Let us focus now between the key differences between LIME and SHAP:

- Focus: LIME focuses on explaining individual predictions for specific data points, while SHAP focuses on explaining feature contributions globally across all predictions.

- Model-Agnosticism: LIME can be applied to any model, while SHAP works best with models that have some inherent feature importance (for example decision trees).

- Interpretability: Both techniques aim for interpretability, but LIME directly creates a simpler model, while SHAP provides insights into feature contributions through techniques like force plots.

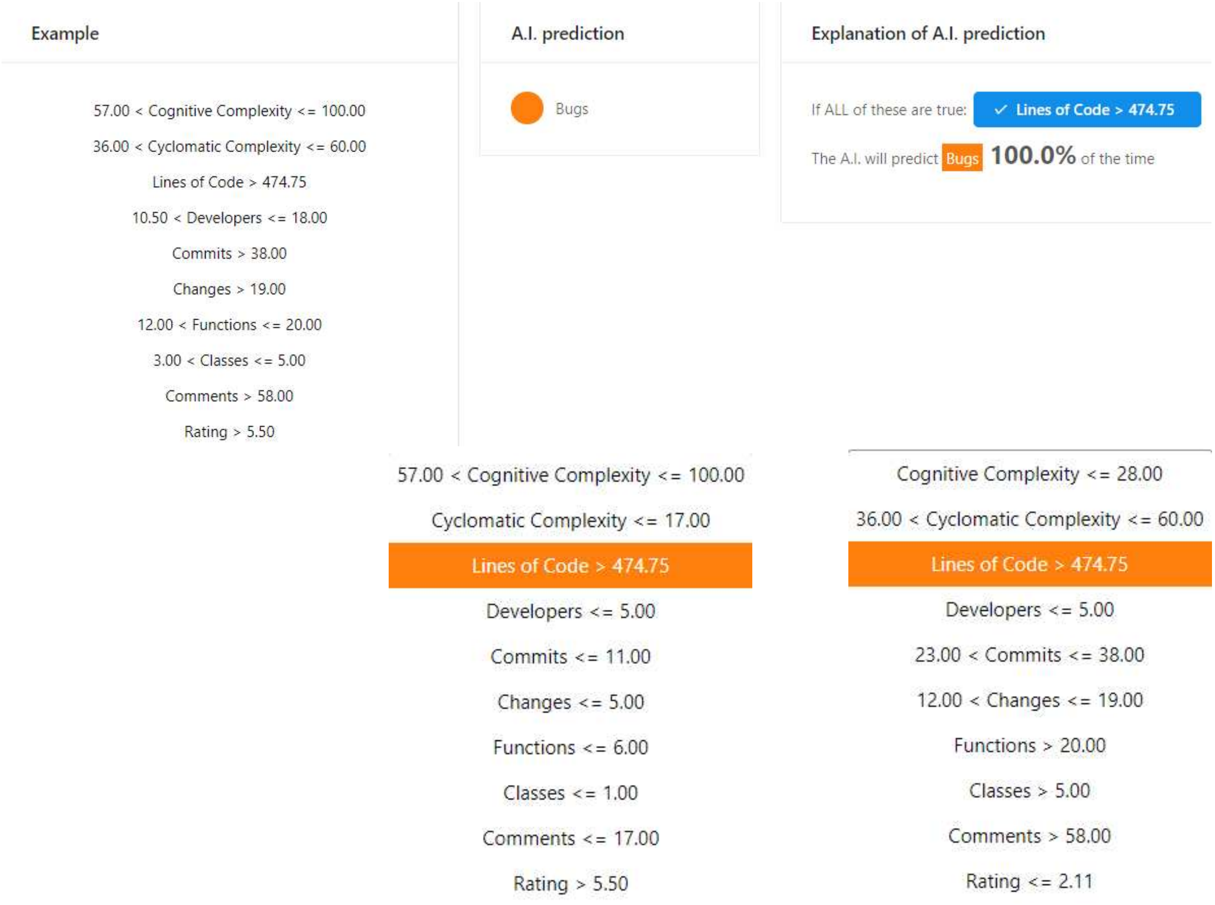
- Choosing between LIME and SHAP depends on your specific needs. To understand the reasoning behind a specific prediction for a particular data point, LIME might be a good choice, while if the main aim is to understand the overall importance of features across all predictions, SHAP could be more suitable.

Anchor

In Explainable AI, Anchor is a technique that focuses on identifying sufficient conditions for a model's prediction. Unlike LIME and SHAP, which delve into feature contributions or local explanations, Anchor aims to pinpoint the core reasons behind a prediction through a set of "anchors." We can see an example of output Anchor in the figure.

Anchor is a model-agnostic technique, meaning it can be applied to various machine learning models. It works by identifying a set of conditions, called anchors, that are sufficient to guarantee a specific prediction from the model. These anchors essentially represent the core reasons why the model makes a particular prediction for a given data point.

Anchor starts by simplifying the complex model's decision function into smaller, easier-to-understand rules. Then it searches for these simplified rules (anchors) within the data point's features. An anchor essentially represents a combination of feature values that is sufficient to guarantee the model's prediction. There might be multiple anchors fulfilling



Example

57.00 < Cognitive Complexity <= 100.00
36.00 < Cyclomatic Complexity <= 60.00
Lines of Code > 474.75
10.50 < Developers <= 18.00
Commits > 38.00
Changes > 19.00
12.00 < Functions <= 20.00
3.00 < Classes <= 5.00
Comments > 58.00
Rating > 5.50

A.I. prediction

● Bugs

Explanation of A.I. prediction

If ALL of these are true: ✓ Lines of Code > 474.75

The A.I. will predict Bugs 100.0% of the time

57.00 < Cognitive Complexity <= 100.00
Cyclomatic Complexity <= 17.00
Lines of Code > 474.75
Developers <= 5.00
Commits <= 11.00
Changes <= 5.00
Functions <= 6.00
Classes <= 1.00
Comments <= 17.00
Rating > 5.50

Cognitive Complexity <= 28.00
36.00 < Cyclomatic Complexity <= 60.00
Lines of Code > 474.75
Developers <= 5.00
23.00 < Commits <= 38.00
12.00 < Changes <= 19.00
Functions > 20.00
Classes > 5.00
Comments > 58.00
Rating <= 2.11

Figure 2.15: Example of Anchor

this condition for a single prediction. The identified anchors are presented in a human-readable format, such as logical statements or data point subsets. This makes it easier to understand the core reasons behind the model's prediction.

Anchor is used because it has some potentiality, for example his ability to work with various models makes it a flexible tool in the XAI toolbox; moreover by identifying sufficient conditions (anchors), Anchor emphasizes the key factors driving the prediction, offering a more concise explanation compared to techniques that analyze all features. A key role is played by the human-readable explanations as it is written above, in fact the identified anchors are easy to understand for humans, promoting trust and fostering better decision-making based on model outputs.

Overall, Anchor provides a valuable approach to XAI by identifying the core reasons behind a prediction through a set of interpretable anchors. This helps users gain insights into the model's decision-making process and fosters trust in its capabilities.

Chapter 3

Problem Statement

In the previous chapters of this thesis, we discussed in general terms of the problems that can be caused by code bugs; in this chapter, we are going to enter in the details of the principal objectives of the thesis, facing first the problems that bugs can cause.

Bugs are a common occurrence in software development, they can be caused by mistakes in coding, misunderstandings in requirements, or problems with integration. Not all bugs are found and fixed before the software is released, so they can end up in the final product as post-release bugs.

Post-release software bugs are a major problem, affecting both developers and users. They can cause the software to malfunction, slow down, or even crash, which is frustrating for users and it can be expensive for companies. Since the data of the thesis are extracted from a project which develops the infotainment system for a luxury car, we will face an example from the reality. If there is a post-release bug on the infotainment it could happen that this is a minor bug such as the touch screen that does not work perfectly or that it is not aligned; but it could also happen that the bug affects a more sensitive area such as the fuel level or the electric part which controls the brake balance of the car. It could be that the car has no more fuel in the middle of a highway without any gas station nearby or, even worse, that the car does not brake and that this bug causes an incident.

This thesis aims to examine methods for preventing, finding, and managing post-release bugs in software. By studying real-world examples, using different development techniques, and applying specialized tools, we hope to create new solutions to this serious problem in software development.

Our aim is to develop more dependable, secure, and superior software by reducing the effects of bugs that arise after software has been released, which benefits businesses and end-users. Specifically, our research focuses not only on the complex task of finding bugs. In fact it is usually harder finding bugs than fixing them. So the aim of the thesis is to help developers to find bugs and to do this we will use probability-based models of different types, starting from logistic regression and moving on to black box models such as neutral networks or random forest. We will confront these models to find which is the best for this project and trying to understand and investigate why this model is the best and, if it is a black box model, how it works in the part inaccessible to us.

3.1 Requirements

We know that it can be harder to spot bugs than to fix them. So, we want to create and use probability-based models to make bug detection easier. This will help us find and deal with bugs quickly and effectively.

Of course the project is crucial for all parties involved, for example customers benefit from early bug detection, resulting in a higher-quality product delivered quickly, enhancing their experience. With fewer bugs after release, customers can trust the software's dependability, fostering brand trust and loyalty.

Our approach simplifies bug detection for programmers, saving them from the time-consuming and expensive task of manually searching for errors. By using probability-based models, they can quickly pinpoint issues, freeing up their time for more important things like creating new features and improving the application. This not only makes them happier but also makes them more productive, leading to better projects and increased success.

Investing in strong bug detection systems benefits companies in many ways. Apart from providing customers with top-notch software, it also creates a favorable brand identity. Satisfied customers tend to recommend the company's offerings, leading to increased customer loyalty and the acquisition of new customers through positive feedback.

Furthermore, by reducing time and resources spent on fixing bugs, the company can improve its efficiency and use of resources. This leads to cost savings and higher profits in the long term because fewer employees spend time troubleshooting and fixing problems after the release.

The aim of this work is to make happier everyone who is involved in software, from helping developers in their every-day job of finding bugs and of course the customers, to offer them a quality product as soon as possible.

3.1.1 Quality scenarios

In a large tech company's software development pipeline, our bug prediction tool is seamlessly integrated. As developers work on different projects, our tool automatically analyzes their code to detect potential bugs early on. Each time new code metrics are created, the tool diligently examines the code, searching for any red flags that could indicate upcoming issues.

Precision is crucial for our bug prediction model. False positives (files incorrectly reported as bugged) waste developer time and resources on non-existent issues. Conversely, false negatives (missing actual bugs) result in undetected problems reaching production,they would transform in post-release bugs, potentially leading to software failures or disruptions. To make sure that our bug prediction model is reliable, we set a rule that it must be at least eighty-five percent accurate. This means that when the model says there is a bug, there is a good chance that there really is one in the file, and of course when the model says that there is not a bug it is very probably that the file is bug-free. To get this level of accuracy, we have to carefully adjust the model's algorithms, test it and keep watching and improving it as we get more information.

Our bug prediction model helps developers find bugs in a file-level with high accuracy. This gives them specific information about possible bugs, in particular in which file is more probably to find a bug, so they can decide which ones to fix first. The importance of reducing the number of false alarms is that developers can spend their time fixing real problems, improving the quality and reliability of the software they make. Moreover, it makes the development process run more smoothly and gets the software to users faster.

Chapter 4

Review of the state of the art

Before solving this problem it was important to read who tried to solve it in the past and more in general who worked on this argument. In this chapter we will face every work that was important to reach our objective in this thesis.

Explainable AI for software Engineering [1] makes the comparison between different machine learning models for software defect prediction starting from the difference between model-specific techniques and model-agnostic techniques for generating explanations (global or local, depending on which technique it is chosen). Then the authors used a dataset of 10 open-source software projects to train and evaluate the models, working on a project-level bug prediction. They considered four different types of defects: crashes, security vulnerabilities, code smells, and documentation defects. The authors evaluated the models using several metrics, including precision, recall, F1-score, and AUC-ROC. At the end in the paper it is evaluated the explainability of the models with different techniques, in particular are used LIME, Shap and LoRMIkA.

Analisi della complessità di tecniche di offuscamento [8] is a thesis not properly on this argument, but it touches the theme of the data collection and of the metrics of the software, so we will describe briefly only what it is about this argument. It investigated the use of code complexity metrics to analyze the effectiveness of different software obfuscation techniques. The thesis starts examining various complexity metrics, which are numerical representations of how complex a code is in terms of interactions between different entities. The study then explored several types of obfuscation techniques currently in use and how each one modifies software, both at the code and control flow levels, based on its intended function and the type of protection to be introduced. Then it automate the application of specific protections and the extraction of certain metrics (from both source files and binaries) using tools.

Use of relative code churn measures to predict system defect density [9] studies how to predict the likelihood of defects in software systems early in the development process. The paper proposes a technique to predict the density of defects (number of defects per unit of code) early on in the development process. This prediction is based on a set of relative code churn measures. These measures do not just consider the absolute amount of code changes, but also factors like component size, in particular they focus on how much code is being changed in the component and the temporal extent of churn, that is how the code changes over time are spread out and if they were all done at once or spread out over a longer period.

Software Defect Prediction using Machine Learning Algorithms: Current State of the Art [10] explores the field of Software Defect Prediction (SDP), a crucial area within software quality assurance. Software engineering encompasses various methods for predicting aspects of software quality, focusing in particular on the following metrics:

- How much effort will be required for testing (test effort prediction)
- How much cost can be prevented by identifying defects early (cost prevention)
- Where errors are likely to occur (error prediction)
- How reusable code components might be (reusability prediction)
- Whether the software meets safety standards (safety prediction)
- How consistent the codebase is (consistency prediction)

The paper acknowledges that many of these prediction methods are still in their early stages and require further research. However, there is a growing interest in SDP from both academia and industry. This is because SDP offers mechanisms to increase the efficiency of software quality assurance activities and to allocate resources more effectively by focusing efforts on areas most likely to contain defects. The paper delves into the current state of the art, discussing how machine learning algorithms are being utilized for software defect prediction.

Don't Touch My Code! Examining the Effects of Ownership on Software Quality [11] investigates the connection between ownership patterns in large software development projects and the likelihood of software failures. The study focuses on two major projects, Windows Vista and Windows 7.

The research explores how different ways of measuring ownership, such as the number of developers with limited experience working on a code section and the concentration of ownership with a single developer or small group. These ownership measures are examined in relation to defects identified before the software's release (pre-release faults) and failures encountered after the software's release (post-release failures). The findings suggest a correlation between a particular ownership structure and the occurrence of software failures. The paper delves deeper by exploring why developers with less expertise might be making changes to specific code sections.

Furthermore, the research analyzes the impact of removing contributions from less experienced developers on defect prediction models based on code changes. This analysis reveals a significant decrease in the model's performance, suggesting that even these contributions hold valuable information.

By drawing on these insights, the paper concludes by offering recommendations for source code change policies that might influence ownership patterns and resource allocation strategies, such as employing code inspections more effectively based on the ownership structure.

How, and why, process metrics are better [12] discusses about the different metrics used in defect prediction techniques for software development. In particular investigating two categories of metrics, process metrics and code metrics, it analyzes data from 85 releases of 12 open-source software projects and finds out that process metrics are generally more useful than code metrics for predicting defects. In particular code metrics tend to be less dynamic, in fact they do not change much between different versions of code, leading to models repetition and identifying always the same file as problematic. Overall, the paper argues that despite code metrics are more used, it would be more effective to focus on process metrics for bug prediction.

A Survey on Bug Prediction Models [13] provides a comprehensive overview of bug prediction models, classifying them based on various criteria and discussing their advantages and disadvantages. In particular it offers a broad overview of bug prediction methods, with an emphasis on statistical and machine learning models. In particular it focuses on deep learning techniques for defect prediction to automatically analyze code, identify defective part of the software by patterns and improve software quality.

Bug Prediction: A Literature Review [14] offers a broad overview of bug prediction methods, with an emphasis on statistical and machine learning models.

Predicting Defects for Eclipse [15] maps defects from the bug database of eclipse (one of the largest open-source projects) to source code locations. The resulting data set lists the number of pre- and post-release defects for every package and file in the eclipse releases 2.0, 2.1, and 3.0. It additionally annotated the data with common complexity metrics. All data is publicly available and can serve as a benchmark for defect prediction models.

Mining software repositories for comprehensible software fault prediction models [16] discusses using a specific data mining technique called Ant Colony Optimization (ACO) to predict errors in software modules. The paper proposes using data mining techniques to analyze software repositories (collections of software code) to predict these errors beforehand. This allows managers to address potential problems before they become critical. Moreover it specifically investigates the use of an ACO-based technique called AntMiner+, that is a data mining technique inspired by how ants find optimal paths. It is considered a comprehensible technique, meaning the results are relatively easy to understand by humans. Then it compares AntMiner+ to other common classification techniques (C4.5, logistic regression, support vector machines) on real-world software datasets and it arrives as a conclusion that the accuracy of AntMiner+ in predicting errors is competitive with other techniques and additionally, the paper argues that

AntMiner+ offers superior comprehensibility. Finally the paper suggests that AntMiner+, an ACO-based data mining technique, can be a valuable tool for software managers to proactively identify potential errors in software projects. This can help improve software quality, stay within budget, and meet deadlines.

A historical perspective of explainable Artificial Intelligence [17] discusses about the Explainable AI, in particular it focuses on the (short) history of this technology, focusing on differences between explanations in expert systems, in recommender systems, in machine learning, underlining the difference between the global explanations, local explanations and counterfactual explanations; and in neural-symbolic learning. In particular it discusses how explainability was conceived in the past, how is understood now and how it might be developed and understood in the future.

Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI [18] speaks about the field of Explainable Artificial Intelligence (XAI), which focuses on making machine learning models more understandable and interpretable. It solves the problem of explainability of new techniques of machine learning, identifying future research directions of the XAI field and discussing potential implications of XAI and privacy in data fusion contexts.

Discussion In conclusion, there are several studies on the topic, some advanced, some less so, and many study software bug prediction, which is a very important topic at this time in history. However, a real comparison between the various models with a comparison of the explainable part is not present in any of these works. Furthermore, the strength of this thesis is the fact that the data is real, in other words it is taken from a real project and not artificially created. This makes the thesis unique in the field of software bug prediction in this sense and makes it a work that is not only theoretical, but applied to real problems.

Chapter 5

Problem Solution

In this chapter we are going to present the solution. First of all it is important to collect data from the code (section 5.1), then the file are divided in two cluster based on the presence of bugs in the file (section 5.2). Then we construct the models (section 5.3), we evaluate them (section 5.4) and we rank based on some values (section 5.5). At the end we are going to use Explainable AI to explain why the models label a file bug-free or defective (section 5.6) and at the end we insert the code in the pipeline (section 5.7).

5.1 Data collection

First of all, there was a need for the project to start from, as mentioned in the previous chapters: it is a software project for the infotainment system of a luxury car. Given this project, the first step was to choose the level of granularity for the research. The best choice seemed to be working at a file-level granularity, as a coarser granularity implied less data and therefore the work would have been less effective. However, a finer granularity posed challenges in extracting the data because it was not clear who had worked on each method and how the work on the project had been done in general. Moreover, it is not evident that working at a finer granularity would make developers' work easier, so we opted for a file-level granularity.

The decision to adopt a file-level granularity was made to ensure that each individual file, containing specific code components, could be analyzed comprehensively. This approach allowed for a detailed examination of the software components, including modules, functions, and classes, within each file. By focusing on the file level, we could capture the nuances and intricacies of the code structure and behaviour, which are crucial for accurate bug prediction.

Moreover, working at the file level provided a balance between granularity and feasibility. It offered sufficient detail to identify patterns and correlations related to bug occurrences, while also ensuring that the dataset remained manageable and interpretable. This granularity level enabled us to gather relevant metrics and features from the source code, which served as input for bug prediction models.

However, it is worth noting that choosing the appropriate granularity level is not a one-size-fits-all decision and may vary depending on the nature of the software project and the specific research objectives. In this case, the file-level granularity was deemed most suitable for achieving the goals of the bug prediction study.

To accumulate the necessary data, we primarily utilized Git and Jira. Git served as the primary source for code-related metrics, such as the number of lines of code per file or the number of commits per file. On the other hand, Jira was used to track past errors and determine how many developers had worked on each file within the project.

Additionally, to acquire further metrics such as the number of functions and classes per file, we leveraged an online tool. This tool, in addition to the above mentioned metrics, provided insights into cognitive complexity, cyclomatic complexity, and code ratings. Consequently, for each file, a comprehensive set of ten metrics was available, including commits, modifications, comments, developers who worked on the file, lines of code, classes, functions, cognitive complexity, cyclomatic complexity, and code rating. Now we are going to explain what these data represent:

- **Lines of code** is the number of lines of the code of the file. It is calculated by Git;
- **Commits** is the number of times that the file has been modified in the past and after that is pushed. It is calculated by Git;
- **Changes** it is the number of times that the file has been modified but, differently from commits, it counts also the time that it is not saved. Of course it is always greater or equal than the number of commits and it is calculated by Git;
- **Comments** is the number of the lines that contains a comment. It is calculated by Git;
- **Functions** is the number of functions declared in each file. It is calculated by Git;
- **Classes** is the number of classes declared in each file. It is calculated by Git;
- **Developers** is the number of people that worked on the file and modified it. It is calculated by Jira;
- **Cyclomatic complexity** is typically calculated using a formula developed by Thomas McCabe Sr. in 1976. The formula considers the number of nodes and edges in the program's control flow graph. It is calculated by SonarQube;
- **Cognitive complexity** shares some similarities with with cyclomatic complexity, but it is not a purely structural measure, but it incorporates additional factors that influence human understanding, such as lines of code, decision points, cognitive load (like nesting, recursion, lack of comments and meaningless names of the variables). It is calculated by SonarQube;
- **Rating** is an evaluation given by some tools that rates a file considering a set of metrics and quality indicator, like code maintainability, code reliability, code coverage by unit tests, security and the amount of duplicated lines of code. It is calculated by SonarQube.

By the way this elaborated approach did not ensure a robust dataset for our analysis yet, because the files were not enough to have a good data sample, because we could not divide it in two parts and have a numerically sufficient-sized dataset. Indeed, with a restricted dataset, the data is too limited to provide a significant and accurate result: there would be too much variance between different attempts, and consequently, the outcome would be too influenced by random factors. For this reason, we proceeded to expand the dataset. A small Python script was used, based on the Sci-Py library, which increased the number of samples in a non-random way, but following the distribution of the original data. This approach involved carefully generating additional data points that, as long as possible, had the same statistical distribution as the original dataset, effectively enlarging the dataset while preserving its underlying characteristics. By imitating the distribution of the original data, we ensured that the newly generated samples represented the same patterns and trends present in the actual project data. This augmentation process enabled us to overcome the limitations posed by the initial dataset's size and variability, resulting in a more comprehensive and representative dataset for our analysis. As a result, we managed to obtain a sufficiently large dataset, equivalent to approximately five hundred files, enhancing the robustness and reliability of our analysis.

5.2 Clustering

Now it is important to divide the dataset in clusters, based on the probability of having bugs in the file. Unfortunately Jira tracks the error but not highlighting in which file are they. So we do not know what are the file with an higher probability of containing bugs. So the dataset is divided into clusters, also in this case using the SciPy library, based on the similarity of the data, particularly in geometric proximity in a hypothetical $\Re^{10}$ space. To choose how many clusters, we rely on Principal Component Analysis (PCA), which helps in reducing the dimensionality of the dataset while retaining most of the relevant information. This step is crucial for grouping similar instances together and forming clusters that can later be analyzed individually.

5.2.1 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a statistical technique used in data analysis and in machine learning to analyze and reduce the dimensionality of large datasets. Its main objective is to identify patterns hidden in the data while simultaneously simplifying complexity through transformation into a new coordinate system, ensuring that variables are uncorrelated and maximizing the information contained in the original data.

To apply Principal Component Analysis we start by calculating the covariance matrix and so that we can find the most influential components, that are the ones which have the biggest impact on the others. Then we calculate eigenvalues and eigenvectors and we sort the eigenvalues in a descending order and we associate the eigenvectors at the correspondent eigenvalue. In this way we can see the "principal direction" and the magnitude in that direction. Now in this project we use Principal Component Analysis to see in how many clusters we have to divide the dataset. To do this we see when there is a big difference between two consecutive eigenvalues. In our model it is at two, so we decided to divide the dataset into two clusters and we assigned a label to each file (zero or one) based on which cluster it belonged to. Clearly files which are close in the hypothetical $\Re^{10}$ space will stay in the same cluster.

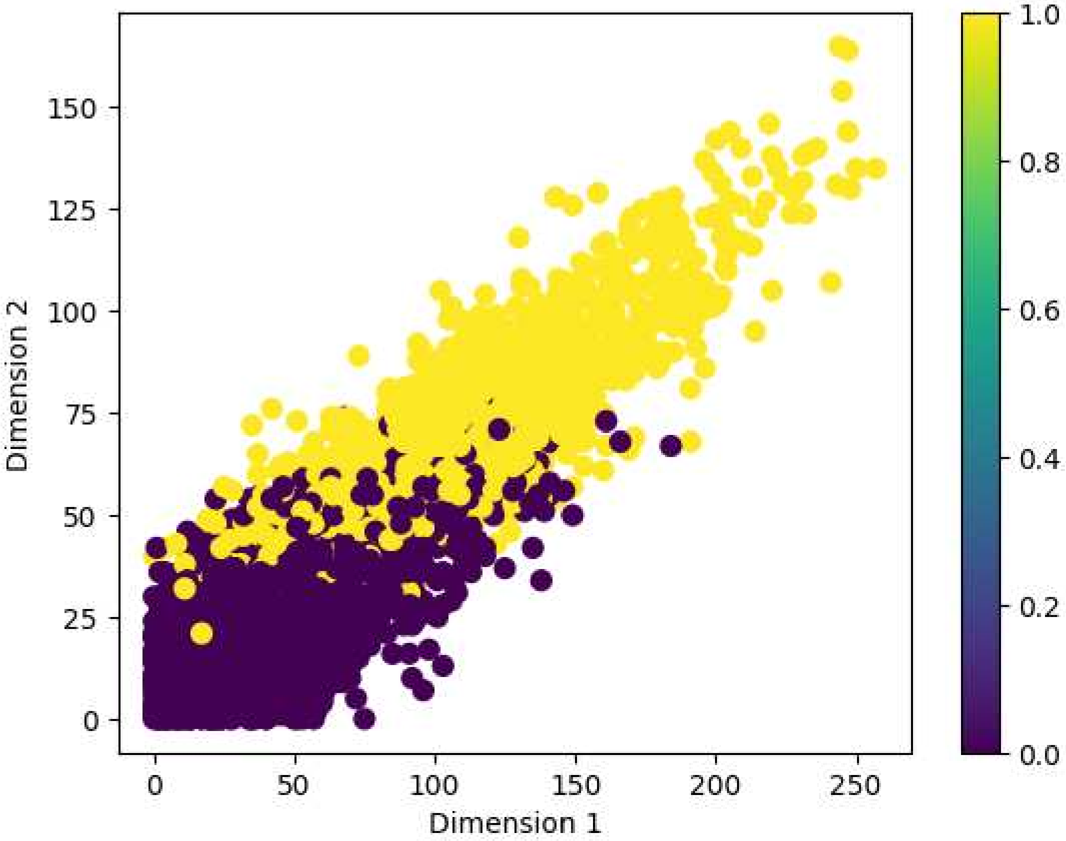


Figure 5.1: Cluster projection in $\Re^2$

We can see from the graph in figure 5.1 of a projection in the $\Re^2$ space of the clusters on the two principal components. In particular the violet points are the files that, according to the division, are bug-free, while the yellow points are the ones that have bugs. We can see that the two regions are almost separated a part from a small amount of points, so the cluster has worked well and we can go on.

5.3 Model Construction

Now the whole dataset has to be divided in two parts (this is the principal reason for the data augmentation tool that we explained above in the previous sections): the first part is used as a train for the model and the second part is the test part.

In the training part the model learns patterns and relationships in the data, adjusting the parameter of the model in order to minimize the difference between the actual and the predicted values. After the training part, the model's performance is evaluated with the test set, which is not used during the training part of course. Here the model makes a prediction on the values of the test set and then it compares with the real values, estimating how well the model will perform on new data. It is important to use cross-validation to data: it is a resampling procedure used to evaluate the performance of a prediction model. It involves repeatedly training and evaluating the model on different partitions (folds) of the data. This helps to ensure that the model is not simply memorizing the training data and that it will generalize well to new, unseen data. First of all it divides the data into folds: The dataset is split into k roughly equal-sized folds. In our model the choice is k=10. Then there is the training part: it uses all folds except the current fold to train the model. Now it evaluates the trained model on the current fold, which was not used for training and it repeats the process for all k folds. At the end it calculates the average performance metric (e.g., accuracy, F1-score) across all k folds. This average represents the model's overall generalizability.

Cross-validation it is used because it has many benefits that we are going to see now. First of all it prevents over-fitting, which occurs when a model learns the training data too well and fails to generalize to new data. Moreover cross-validation provides a more unbiased estimate of the model's true performance and it can be used also to compare different machine learning models fairly, as they are all evaluated on the same data partitions.

There are some types of cross-validation:

- k-fold cross-validation: As described above, this is the most common type, where the data is split into k folds.
- Leave-one-out cross-validation (LOO): In LOO, there are k=n folds, where n is the number of data points. Each point is left out as a validation fold once.
- Stratified cross-validation: This method ensures that each fold contains a similar proportion of classes as the overall dataset, especially important for imbalanced datasets.

Cross-validation is a crucial step in the machine learning process, providing a robust way to evaluate model performance and select the best model for a given task. It helps to ensure that the model will perform well on unseen data, which is the ultimate goal of any predictive model.

5.4 Model Evaluation

Now we have all the data that we need, so we can do a classification of the models to determine the one that have worked better. To do this first we have to introduce on what we base on to evaluate the model and to compare the different models one to each other.

The models are ranked basing on performance metrics, that are usually precision and the area under the receiver operator characteristic curve (that is often abbreviated with area under the curve: AUC).

The precision is very important, because it measures the proportion between the number of files defective correctly identified as defective and the number of files that are really defective, whether they are recognised or not. In particular the formula used in the thesis is $\frac{TP}{TP+FP}$, where TP is the number of files defective identified as defective and FP is the number of false positive, that is the number of the files identified as defective but which are not defective. It is important because a high precision value indicates that the models can correctly identify high number of defective lines.

Another important aspect to measure is the False Alarm Rate (FAR). It is the proportion between the number of clean files identified as defective and the real number of clean files. More precisely, the false alarm rate can be calculated with a calculation of $\frac{FP}{FP+TN}$, where FP is the number of false positives and TN is the number of clean files that are not predicted as defective. Of course the false alarm rate is lower as long as is lower the number of false positives or bigger the number of the clean files considered clean by the model. The false alarm rate is very important, because it measures the effective accuracy and the degree of confidence that developers should have in the code, in the sense of how confidently they can go ahead to fix bugs in a particular file.Using these two metrics, we obtain valuable insights into the performance and reliability of our bug prediction model.

We also will use the true positive rate (TPR), that is the proportion of true positives correctly identified (TP) by the model over the positives present in the project, that include also the false negatives (FN), that are the bugs not identified by the model; and it can be calculated with a calculation of $\frac{TP}{TP+FN}$.

In our case since it is a binary classification problem, we can calculate the area under the receiver operator characteristic curve, that is the typical metric that is measured for models that have the aim to identify if something is defective or non-defective. The model predicts the probability that an instance belongs to the defective class, and the AUC assesses how well the model ranks instances by this probability. To calculate the AUC we used a machine learning formula from sci-kit learn.metrics, that is a Python library. Let us see better how it works: to compute the AUC, first, it generates a Receiver Operating Characteristic (ROC) curve that plots the true positive rate (TPR) against the false positive rate (FPR). After plotting the graph of the Receiver Operating Characteristic curve, the AUC is the area under that curve. It ranges from 0 to 1: when the value is 1 it means that the model is a perfect classifier (all true positives and no false positives), while when the value is 0,5 it suggests that the model has no discriminatory power, in other words it means that the model is not better than deciding randomly if a file is bug-free or not.

After that, we calculate the precision of each model. This is done using code from the scikit-learn library and in particular it gives as output the number of file with bugs that are correctly reported (true negative), the number of file with bugs that are not reported (false positive), the number of file bug-free that are clean to the model (true positive) and the number of files bug-free that are reported as defective by the model (false negative) between the part of files that are in the test part of the dataset. Then we calculate the precision that is the ratio between the false positive files and all the clean files of the test part of the dataset.

5.5 Model Ranking

In this section we will analyze the models and we will try to do a classification.

First of all we start with the precision score, that we indicate in this table 5.1 and the relative graph:

Model	Precision score	$\begin{pmatrix} \text{True Positive} & \text{False Positive} \\ \text{False Negative} & \text{True Negative} \end{pmatrix}$
Decision Tree	0.9942028985507246	$\begin{pmatrix} 343 & 2 \\ 4 & 558 \end{pmatrix}$
eXtreme Gradient Boosting Tree	0.9914040114613181	$\begin{pmatrix} 346 & 3 \\ 1 & 557 \end{pmatrix}$
Random Forest	0.9913544668587896	$\begin{pmatrix} 344 & 3 \\ 3 & 557 \end{pmatrix}$
Gradient Boosting Machine	0.9884726224783862	$\begin{pmatrix} 343 & 4 \\ 4 & 556 \end{pmatrix}$
Neural Network	0.912	$\begin{pmatrix} 342 & 33 \\ 5 & 527 \end{pmatrix}$
Logistic Regression	0.7611940298507462	$\begin{pmatrix} 306 & 96 \\ 41 & 464 \end{pmatrix}$

Table 5.1: Precision score of models

We can see in figure 5.2 that the results are very close one to each other except for the logistic regression that has clearly a lower precision score, in fact it has many false positives and false negatives. Also Neural Networks seem like that is worse than the best models in terms of precision score (0,912 versus around 0,99 of Decision Tree, XGB Tree, Random Forest and GBM). In particular the best model for the precision is the **Decision Tree**, in fact it has the lowest number of false positive and of consequence the lowest ratio of False positive over all positives.

By the way this is not the only parameter that we took in consideration, but it is important also the AUC, that is the Area under the ROC curve that we explained above. Here there is a table and the relative graph of the different values for every model of the AUC.

We can see from the table and the graph in figure 5.3 that the value of the AUC is almost similar for every model except the logistic regression, whose AUC is around 0.885, and Neural Network that has AUC around 0.931. The other four models have the same value rounded to the nearest hundredth.

In conclusion Decision Tree, Random Forest, eXtreme Gradient Boosting Tree and Gradient Boosting machines are almost equivalent looking to AUC and precision, while Neural Network and Logistic Regression are clearly worse than the others. It is pretty surprising that Decision Tree has a similar rating to Random Forest and eXtreme Gradient Boosting Tree that are evolution of Decision Tree. Probably it is due to the fact that the number of terms of evaluation is not so large (there are only ten metrics taken into consideration).

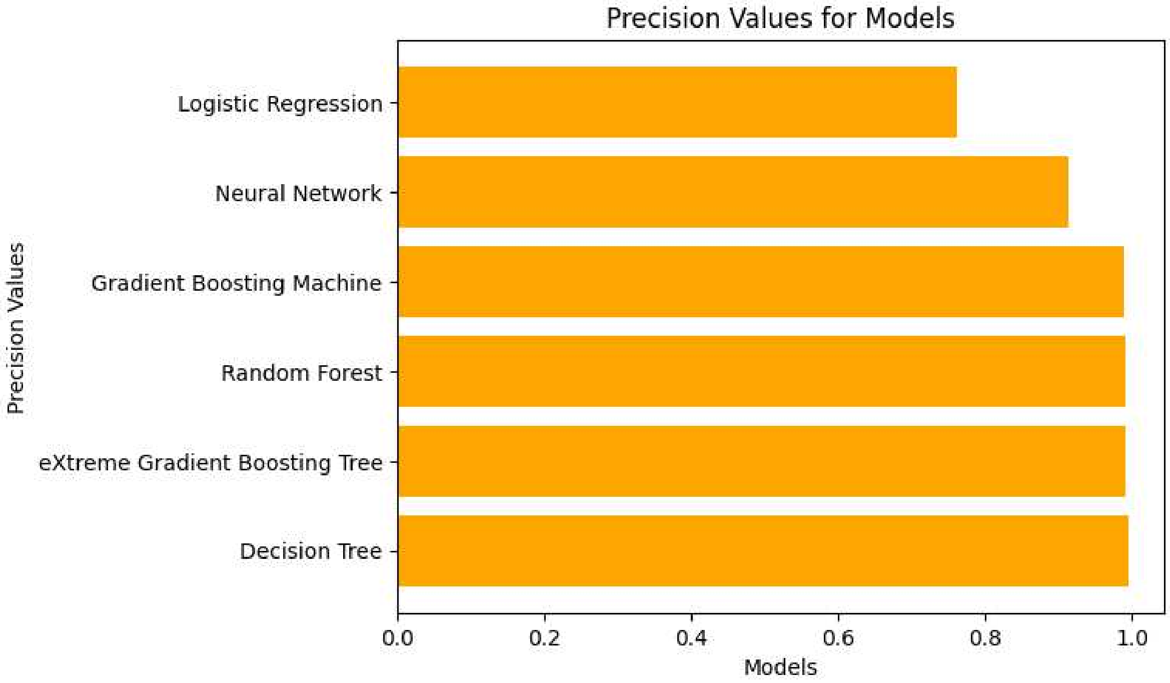
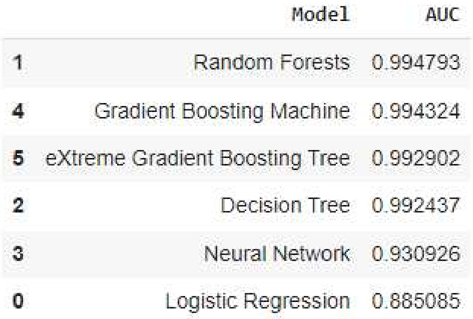
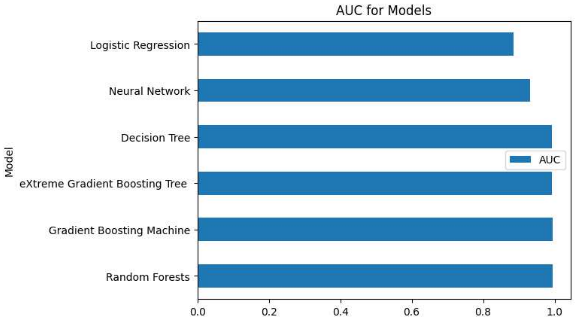


Figure 5.2: Precision Values for Models



	Model	AUC
1	Random Forests	0.994793
4	Gradient Boosting Machine	0.994324
5	eXtreme Gradient Boosting Tree	0.992902
2	Decision Tree	0.992437
3	Neural Network	0.930926
0	Logistic Regression	0.885085

(a) AUC Table for Models



(b) AUC Graph for Models

Figure 5.3: AUC for Models

5.6 Explainable AI

Lastly, it is important to understand the rationale behind the model's choices and the motivations that lead to classifying a file as bug-free or as containing bugs.

First of all there is a main difference between the models: in fact two of them (Logistic Regression and Decision Tree) are explainable models, while the other (eXtreme Gradient Boosting Tree, Neural Network, Gradient Boost Machines and Random Forest) are black box models. The first two models can be explained without any help: in fact they are "transparent" models and we can see with our eyes the process that lead the model to label a file as bug-free or defective. On the other hand we do not know the process of black box models, they are as a black box where we do not know anything inside it, but we know only the input and the output. This can lead to difficulties in comprehending why a file is defective, what the developer can do to fix it and where the developer has to focus to fix the bug and to write better the code.

To make black box models less mysterious to us, it is fundamental the technology of Explainable AI. In particular we used three techniques (SHAP, LIME and Anchor) to explain the process inside the black box and the reason why a model labels a file as defective or bug-free, which are the main metrics that influence the final result and what patterns they follow to classify files.

5.6.1 An explainable model: Decision Tree

In figure 5.4 there is the decision tree of the project: we can see that it is coloured: in fact with every decision it goes towards labeling the file that it is examining as bug-free or defective. In this case blue means that it is a defective file, while orange means that it is bug-free. On every node the model has to take a decision, now we are going to see what is written in one of the rectangles (it was not possible to make it more visible inside the graph for reasons of space) and explore the meaning, in this case we will take the first decision node (the one at the top).

$(Lines \quad of \quad Code \leq 394.5, \quad ngini = 0.452, \quad nsamples = 2114, \quad nvalue = [1384, 730])$

- $Lines \quad of \quad Code \leq 394.5$: This line represents the condition that determines how data points are directed through this node. In this case, it's considering the number of "Lines of Code" (a feature in your data). Data points with a value of "Lines of Code" less than or equal to 394.5 are sent down one branch of the tree (in the bug-free branch, so to the left), while those greater than 394.5 go down another branch (to the right).

- $ngini = 0.452$: This value, called the Gini impurity, indicates how well-separated the classes (possibly target variables) are at this node. A Gini of 0 represents perfect separation (all data points belong to the same class), while a value closer to 1 signifies a more even mix of classes. In this case, 0.452 suggests some level of separation but not perfect.

- $samples = 2114$: This tells us the total number of data points that reach this node in the decision process.

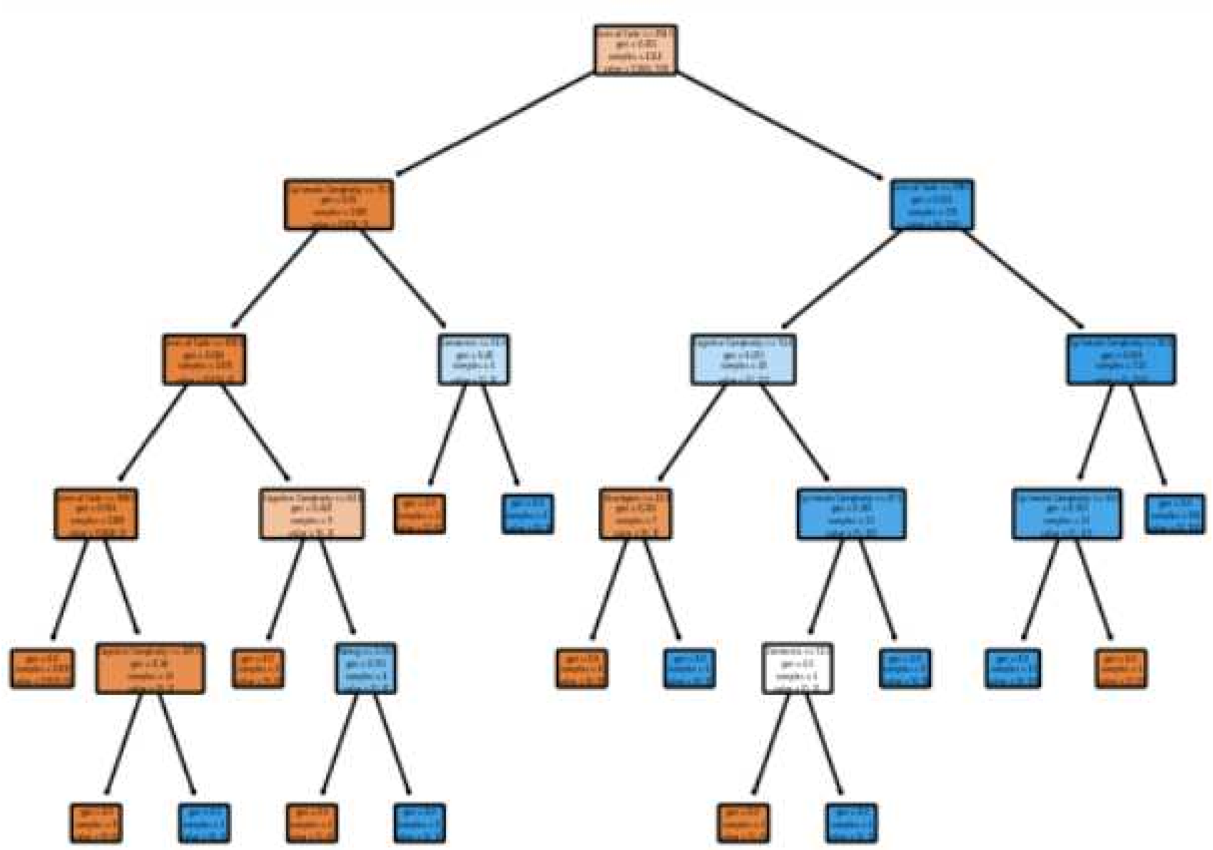


Figure 5.4: Decision tree in the model

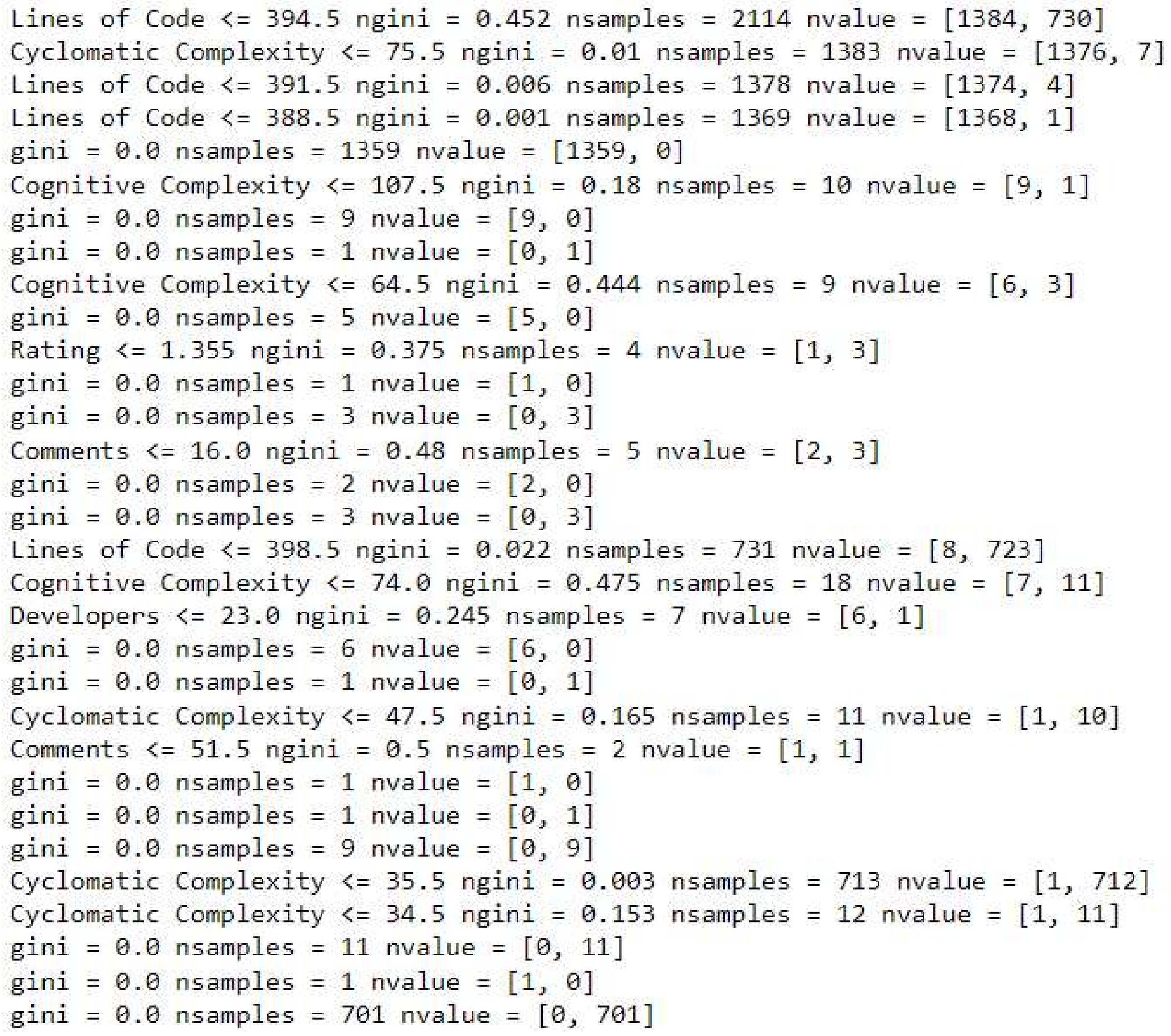
- *value* = [1384, 730]: This likely represents the distribution of class labels (or target values) among the 2114 data points at this node. The bracketed numbers, [1384, 730], could indicate the count of data points belonging to two different classes. In our case 1384 belong to class 0 (the orange one) while 730 belong to class 1 (the blue one).

This text is present for every decision node (figure 5.5) and it explains what is the process that lead the decision tree to label a file as bug-free or defective.

SHAP

SHAP shows which are the most important metrics for the software for every model. In particular we show on figure x the graphs of every black box model. We can note that for every model except for Neural Networks the most important metric to determine whether a file present or not a bug is the number of lines of the code. Then we can also note that the models do not work in the same way: for example the metric that indicates the rating of a file and the one that indicates the number of functions for neural networks is much more important than for the other models, while lines of code are less important.

We start studying Gradient Boosted Machines model. In figure 5.6 we can note that the most important feature for Gradient Boosted Machines model is definitely Lines of



```
Lines of Code <= 394.5 ngini = 0.452 nsamples = 2114 nvalue = [1384, 730]
Cyclomatic Complexity <= 75.5 ngini = 0.01 nsamples = 1383 nvalue = [1376, 7]
Lines of Code <= 391.5 ngini = 0.006 nsamples = 1378 nvalue = [1374, 4]
Lines of Code <= 388.5 ngini = 0.001 nsamples = 1369 nvalue = [1368, 1]
gini = 0.0 nsamples = 1359 nvalue = [1359, 0]
Cognitive Complexity <= 107.5 ngini = 0.18 nsamples = 10 nvalue = [9, 1]
gini = 0.0 nsamples = 9 nvalue = [9, 0]
gini = 0.0 nsamples = 1 nvalue = [0, 1]
Cognitive Complexity <= 64.5 ngini = 0.444 nsamples = 9 nvalue = [6, 3]
gini = 0.0 nsamples = 5 nvalue = [5, 0]
Rating <= 1.355 ngini = 0.375 nsamples = 4 nvalue = [1, 3]
gini = 0.0 nsamples = 1 nvalue = [1, 0]
gini = 0.0 nsamples = 3 nvalue = [0, 3]
Comments <= 16.0 ngini = 0.48 nsamples = 5 nvalue = [2, 3]
gini = 0.0 nsamples = 2 nvalue = [2, 0]
gini = 0.0 nsamples = 3 nvalue = [0, 3]
Lines of Code <= 398.5 ngini = 0.022 nsamples = 731 nvalue = [8, 723]
Cognitive Complexity <= 74.0 ngini = 0.475 nsamples = 18 nvalue = [7, 11]
Developers <= 23.0 ngini = 0.245 nsamples = 7 nvalue = [6, 1]
gini = 0.0 nsamples = 6 nvalue = [6, 0]
gini = 0.0 nsamples = 1 nvalue = [0, 1]
Cyclomatic Complexity <= 47.5 ngini = 0.165 nsamples = 11 nvalue = [1, 10]
Comments <= 51.5 ngini = 0.5 nsamples = 2 nvalue = [1, 1]
gini = 0.0 nsamples = 1 nvalue = [1, 0]
gini = 0.0 nsamples = 1 nvalue = [0, 1]
gini = 0.0 nsamples = 9 nvalue = [0, 9]
Cyclomatic Complexity <= 35.5 ngini = 0.003 nsamples = 713 nvalue = [1, 712]
Cyclomatic Complexity <= 34.5 ngini = 0.153 nsamples = 12 nvalue = [1, 11]
gini = 0.0 nsamples = 11 nvalue = [0, 11]
gini = 0.0 nsamples = 1 nvalue = [1, 0]
gini = 0.0 nsamples = 701 nvalue = [0, 701]
```

Figure 5.5: Decision Tree text

Code; in fact the mean SHAP value of the feature is almost four times more than the SHAP value of cyclomatic complexity. This means (and we will see examples in figure 5.14 in next pages when we will face the Anchor technique) that lines of code are really important for this model and if a file will have many lines of code, automatically it will be flagged as a defective file. On the other way if a file has a few number of lines of code, it will be flagged as a bug-free file. Other important features are cyclomatic and cognitive complexity, while rating and number of commits are almost negligible.

Then we are going to face SHAP on eXtreme Gradient Boosting Tree model. In figure 5.7 we can see that the graph is really similar to the 5.6 of SHAP for Gradient Boosted Machines model. For eXtreme Gradient Boosting Tree too the most important feature is the number of lines of code and in this case too it is so much more important than the other features, by the way here cognitive complexity is more important then cyclomatic complexity and we can see that number of classes and comments are very close for importance to the previous two features.

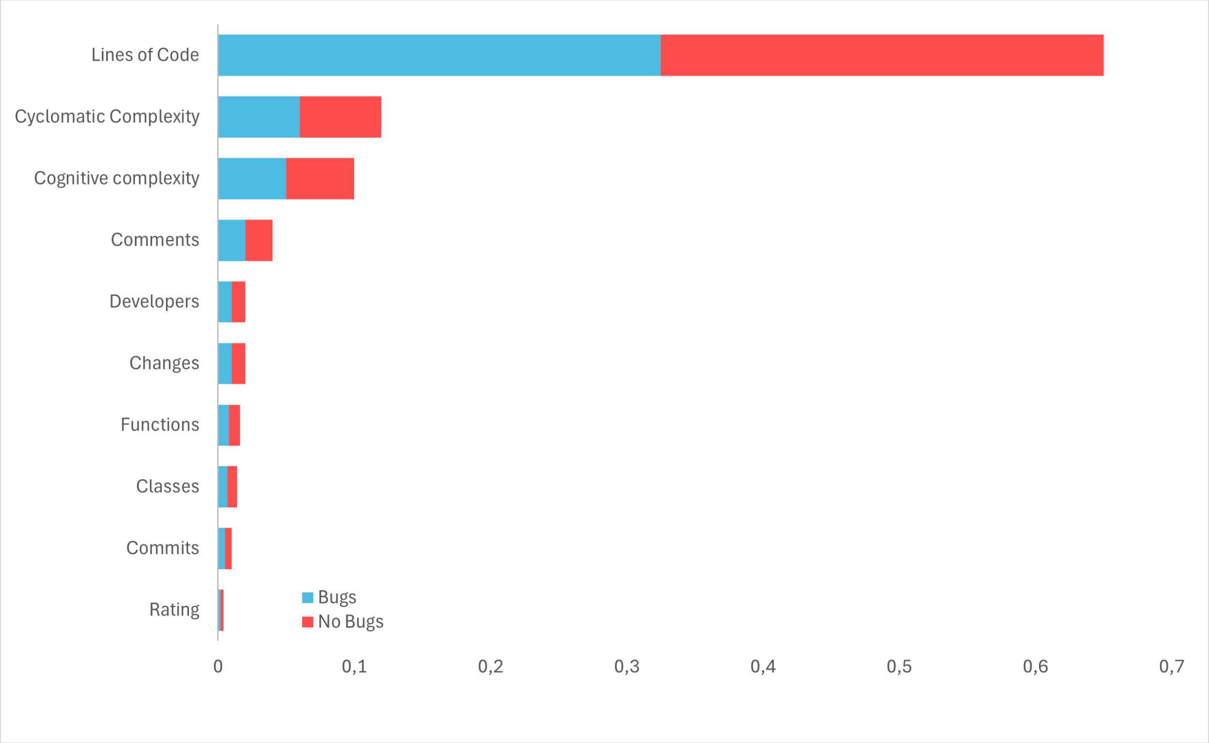


Figure 5.6: SHAP Gradient Boosted Machine

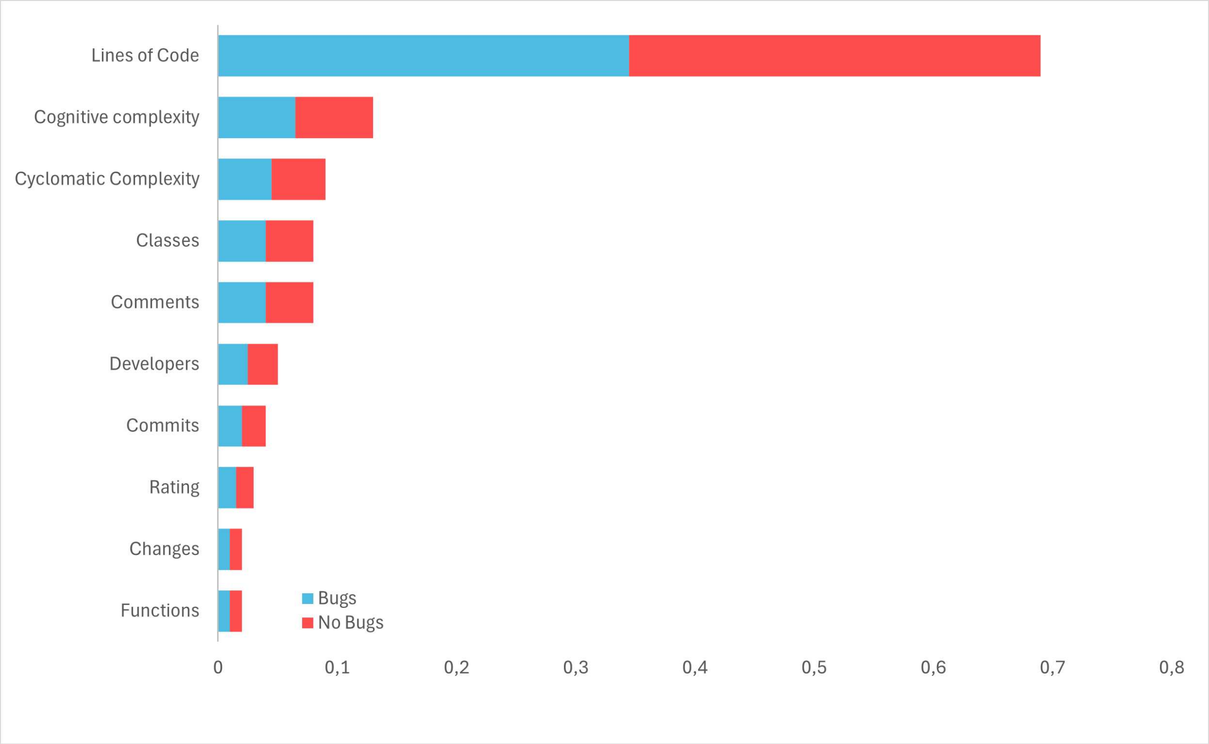


Figure 5.7: SHAP eXtreme Gradient Boosting Tree

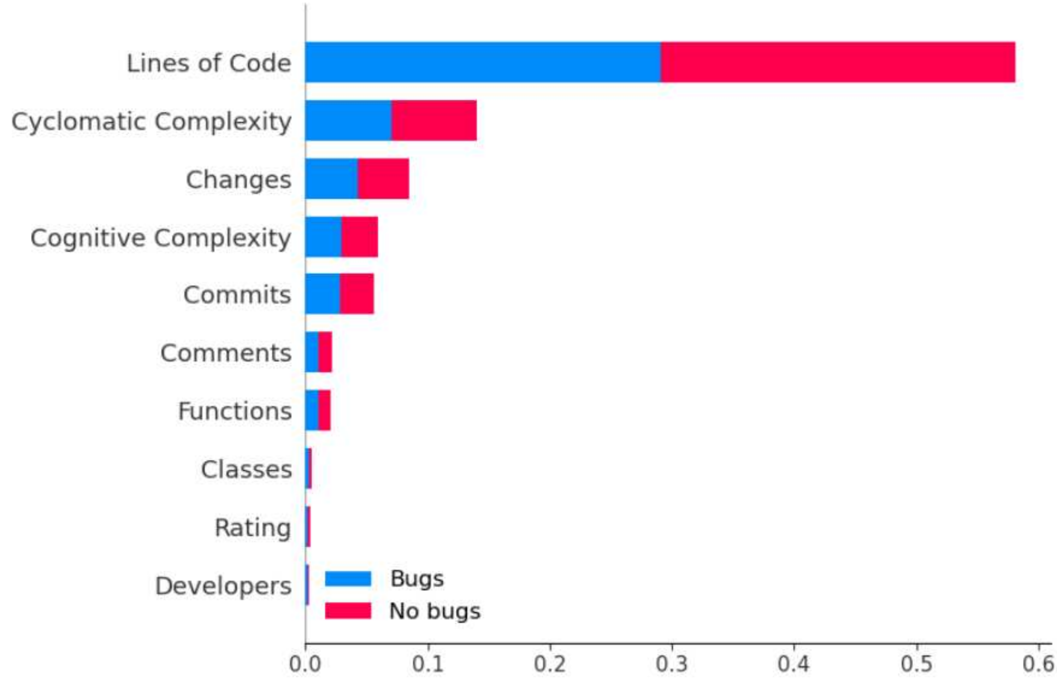


Figure 5.8: SHAP Random Forest

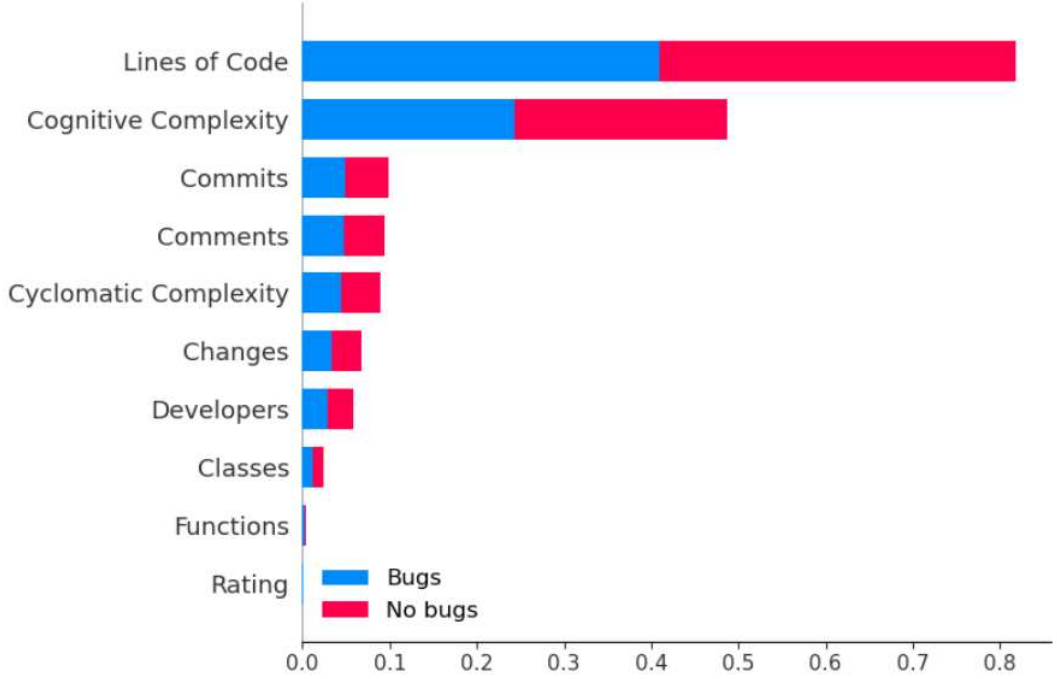


Figure 5.9: SHAP Neural Network

For random forest things does not change so much: we can see from figure 5.8 that lines of code are still the most important feature, followed by cyclomatic complexity that for this model is more important in proportion to lines of code respect to the other. Moreover some features like number of classes, rating of files and number of developers are really close to zero, so they are almost negligible because their weight in the decision of the Random Forest are very low.

At the end we can see how neural network works on figure 5.9. Neural Network is different from the previous model as behaviour of the model and reason why it flags a file as defective whether not. In fact here the number of lines of code is the most important feature, but it is very important cognitive complexity too, differently from the previous models where lines of code was so much more important than every other feature. By the way the number of lines of code is still the most important feature, but not as by far as for other models. Moreover there are two features almost negligible: the number of functions and the rating of the file.

LIME

Differently from SHAP, LIME is a punctual technique of explainable AI and it works on a single file to show why it has bug whether it is bug-free and why. In fact it collects the metrics and it underlines which metric is the most important for the result, that is the metrics that have a major contribution to the possibility of a file to have bugs. Now we can see four example of LIME applied to each black box model.

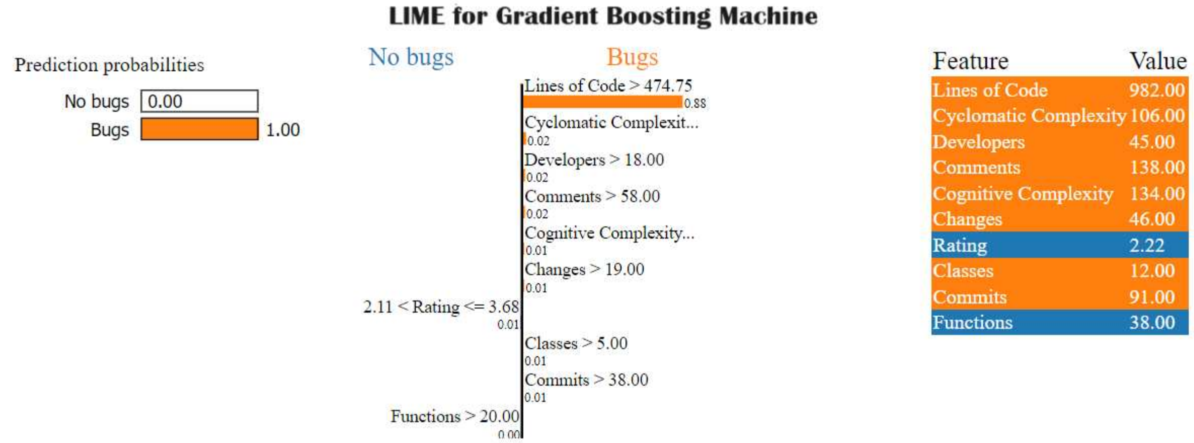
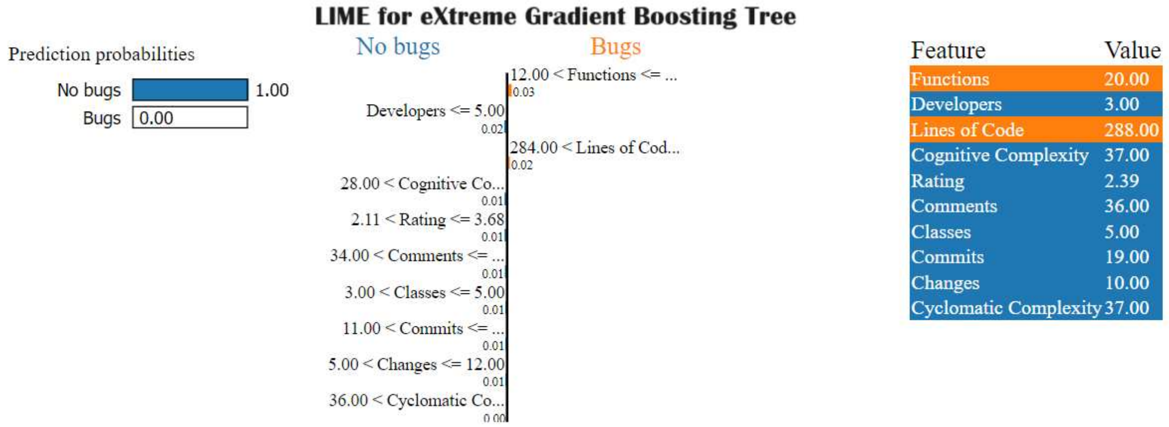


Figure 5.10: LIME Gradient Boosted Machine

For the Gradient Boosting Machines the file presents bugs according to the model. In particular from figure 5.10 we can deduce that this is caused principally by the magnitude of lines of code that has a big impact on the system; also the cyclomatic complexity contributes, but we can see from the bar graph in the middle that it has a contribution almost insignificant respect to the importance of the lines of code.

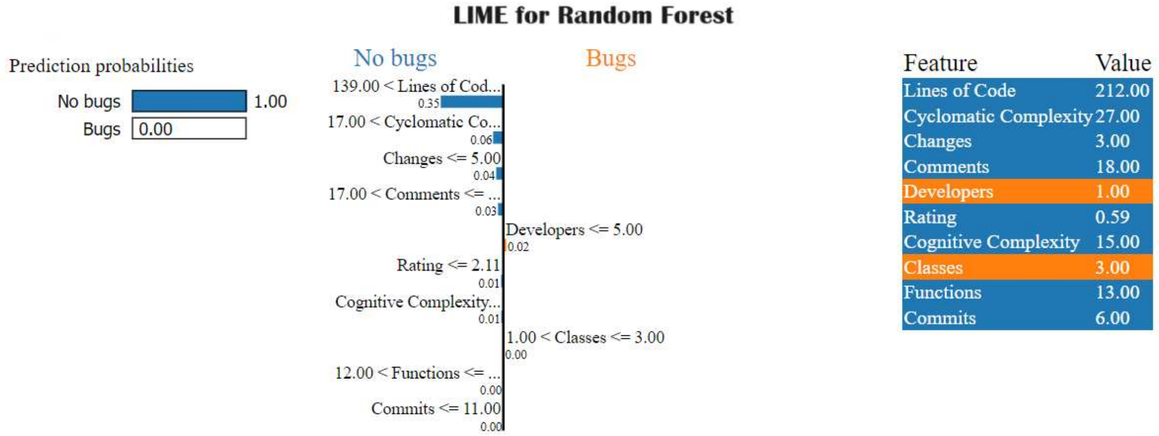
In the example of LIME for eXtreme Gradient Boosting Tree above in figure 5.11 the file analyzed is bug-free. We can see that the feature that more contributes to the choice, differently from the LIME for Gradient Boosted Machines previously cited, is the number of functions, but it is not a decisive contribution. In fact despite the number of functions is pretty big and it would have lead the file to have a potential bug, the other metrics are good and they balance the fact that with that number of functions it is probably that a file presents a bug.



Feature	Value
Functions	20.00
Developers	3.00
Lines of Code	288.00
Cognitive Complexity	37.00
Rating	2.39
Comments	36.00
Classes	5.00
Commits	19.00
Changes	10.00
Cyclomatic Complexity	37.00

Figure 5.11: LIME eXtreme Gradient Boosting Tree

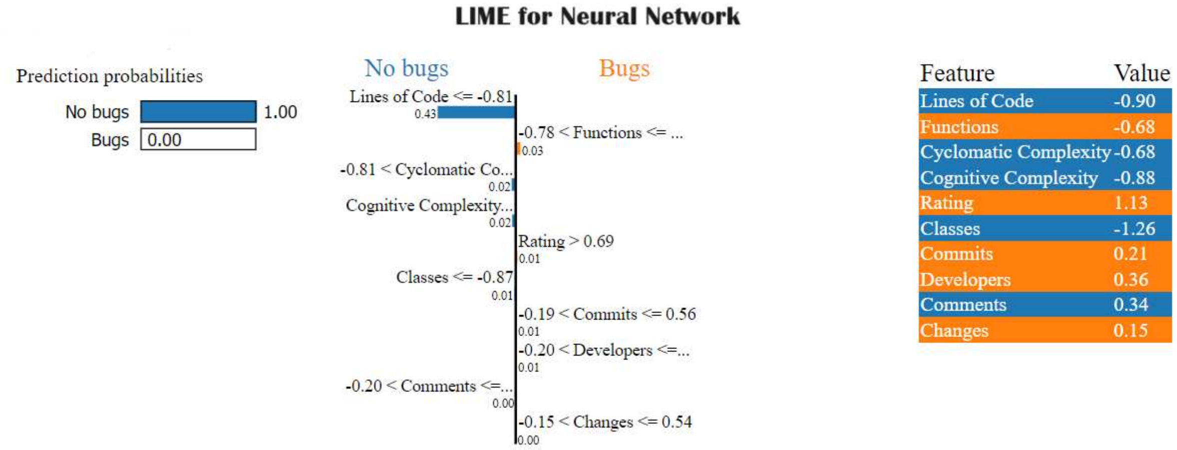
In the file examined with LIME for Random Forest (figure 5.12) we have that, as in the previous example of Gradient Boosted Machine, that the number of lines of code plays an important role on the classification of the files. In fact it is considered that the file has a number of lines of code that lead the Random Forest to classify it as a non defective file. The other metrics are negligible respect to the metric that counts lines of code, by the way almost all the metrics lead to classify the file as non defective.



Feature	Value
Lines of Code	212.00
Cyclomatic Complexity	27.00
Changes	3.00
Comments	18.00
Developers	1.00
Rating	0.59
Cognitive Complexity	15.00
Classes	3.00
Functions	13.00
Commits	6.00

Figure 5.12: LIME Random Forest

At the end we are going to examine LIME for Neural Network in figure 5.13. The file examined results not defective, but in Neural Networks we can observe, differently from the previous example that we faced, that some metrics have negative values. Of course that value does not indicate the value of the metric (it is impossible that lines of code are negative), but it is calculated in a different way and it is the impact that the metric has on the choice. Then it is multiplied by the weight of the metric and we obtain that the file has no bug, in particular thanks to the number of lines of code that is typical of a bug-free file.



Feature	Value
Lines of Code	-0.90
Functions	-0.68
Cyclomatic Complexity	-0.68
Cognitive Complexity	-0.88
Rating	1.13
Classes	-1.26
Commits	0.21
Developers	0.36
Comments	0.34
Changes	0.15

Figure 5.13: LIME Neural Network

Anchor

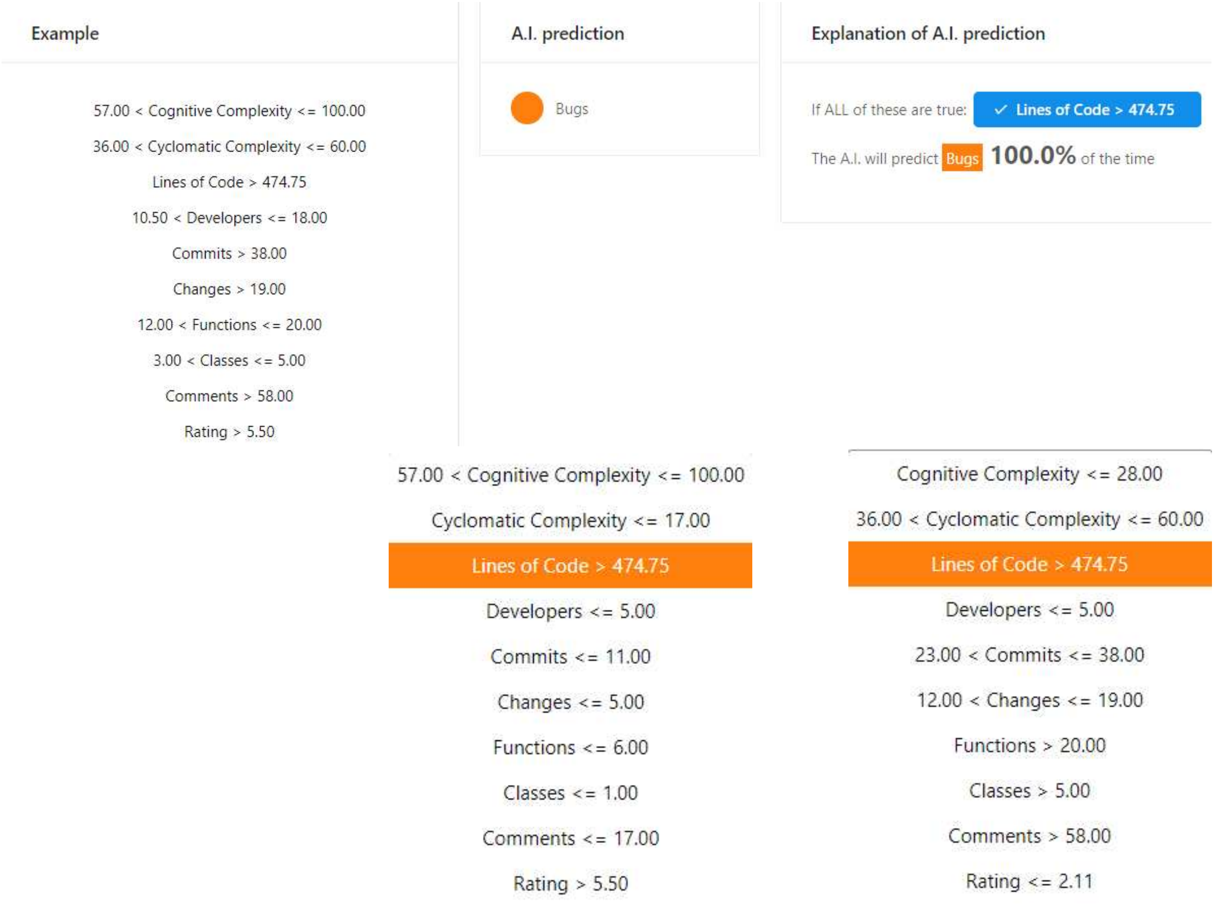
Finally we used the Anchor technique to explain how the AI works. First of all it is important to recall that Anchor is a punctual technique as LIME is, so we will proceed as above analyzing one file with each model.

We start with Anchor used to explain how Gradient Boosted Machines works on a file (figure 5.14): we note first of all that Anchor provides not the exact value of the metrics, but a range: in fact it underlines that Cognitive Complexity is minor than 100, while the lines of code are major than 474. In particular Gradient Boosted Machines classifies the file as defective; it then explains us why it is defective: in fact it underlines that every file that has more than 474 lines of code is classified by AI as a defective one. Moreover it provides us some examples (we attached only two of them, but they were ten) where the AI classifies the files as defective. We note that in both the cases lines of code are more than 475 and as a consequence the file is classified as defective.

Then in figure 5.15 Anchor is used to explain eXtreme Gradient Boosting Tree. We can note that the file is bug-free, in fact according to Anchor, if a file has less than 284 lines of code it surely does not contain bugs according to the eXtreme Gradient Boosting Tree model. In particular it provides ten examples (as before we will show two) in which the file has less than 284 lines of code and as a consequence it results bug-free.

In figure 5.16 Anchor is used to explain Random forest. We can note that the file considered is bug-free, in fact according to Anchor, if a file has less than 139 lines of code it surely does not contain bugs according to the eXtreme Random Forest model. In particular it provides ten examples (as before we will show two) in which the file has less than 284 lines of code and as a consequence it results bug-free. We can see that also if the other features change, the file is still bug-free.

By the way this is not true for Neural Network model. We can see in figure 5.17 in fact that for Anchor this model presents a particularity: it takes into consideration lines of code and cyclomatic complexity and if the number of lines of code is more or equal than 475 and the cyclomatic complexity is higher than 60, the AI will predict bugs $99,9\%$ of times. First of all we can underline that it was expected from SHAP (figure 5.9) that



Example

57.00 < Cognitive Complexity <= 100.00
36.00 < Cyclomatic Complexity <= 60.00
Lines of Code > 474.75
10.50 < Developers <= 18.00
Commits > 38.00
Changes > 19.00
12.00 < Functions <= 20.00
3.00 < Classes <= 5.00
Comments > 58.00
Rating > 5.50

A.I. prediction

● Bugs

Explanation of A.I. prediction

If ALL of these are true: ✓ Lines of Code > 474.75

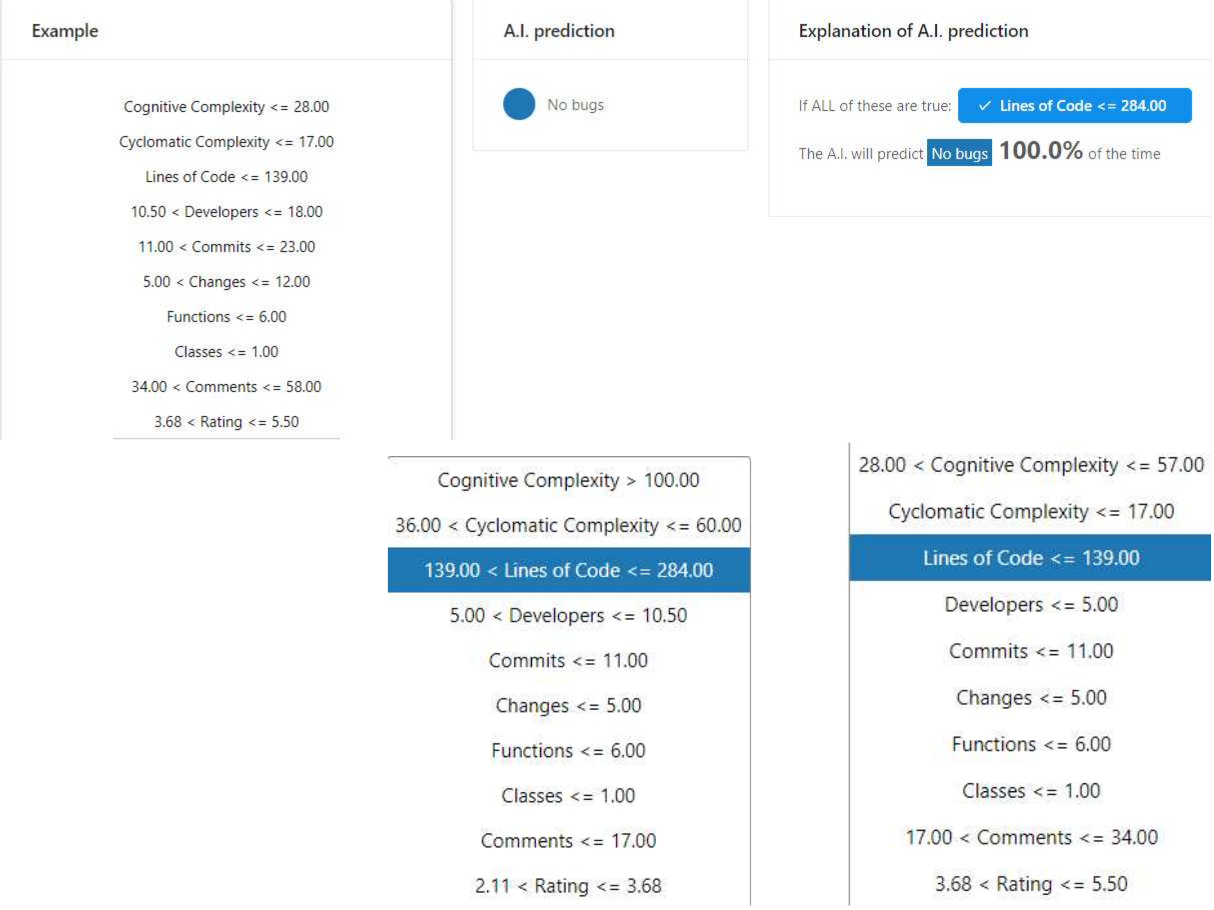
The A.I. will predict Bugs 100.0% of the time

57.00 < Cognitive Complexity <= 100.00	Cognitive Complexity <= 28.00
Cyclomatic Complexity <= 17.00	36.00 < Cyclomatic Complexity <= 60.00
Lines of Code > 474.75	Lines of Code > 474.75
Developers <= 5.00	Developers <= 5.00
Commits <= 11.00	23.00 < Commits <= 38.00
Changes <= 5.00	12.00 < Changes <= 19.00
Functions <= 6.00	Functions > 20.00
Classes <= 1.00	Classes > 5.00
Comments <= 17.00	Comments > 58.00
Rating > 5.50	Rating <= 2.11

Figure 5.14: Anchor Gradient Boosted Machine

in Neural Network could happen that the result depend by two features, while in the other models it could not. In fact we discussed above that the cyclomatic complexity is important for neural network almost as much as the number of lines of code. Moreover it is interesting that Anchor provides us ten examples of file that present a bug for neural network with the number of lines of code is more or equal than 475 and the cyclomatic complexity is higher than 60, but it presents also the counterexample, that is the file that does not present bug.

At the end we can conclude that the most important feature for all the models is the number of lines of code and that in general there are some features (in particular rating, number of classes and number of functions) that are almost negligible for all the models. Moreover also the other features has an impact very limited respect to the one that have the number of lines of code that is in fact the principal component that we calculated in section 5.2.



Example

Cognitive Complexity <= 28.00
Cyclomatic Complexity <= 17.00
Lines of Code <= 139.00
10.50 < Developers <= 18.00
11.00 < Commits <= 23.00
5.00 < Changes <= 12.00
Functions <= 6.00
Classes <= 1.00
34.00 < Comments <= 58.00
3.68 < Rating <= 5.50

A.I. prediction

● No bugs

Explanation of A.I. prediction

If ALL of these are true: ✓ Lines of Code <= 284.00

The A.I. will predict No bugs 100.0% of the time

Cognitive Complexity > 100.00
36.00 < Cyclomatic Complexity <= 60.00
139.00 < Lines of Code <= 284.00
5.00 < Developers <= 10.50
Commits <= 11.00
Changes <= 5.00
Functions <= 6.00
Classes <= 1.00
Comments <= 17.00
2.11 < Rating <= 3.68

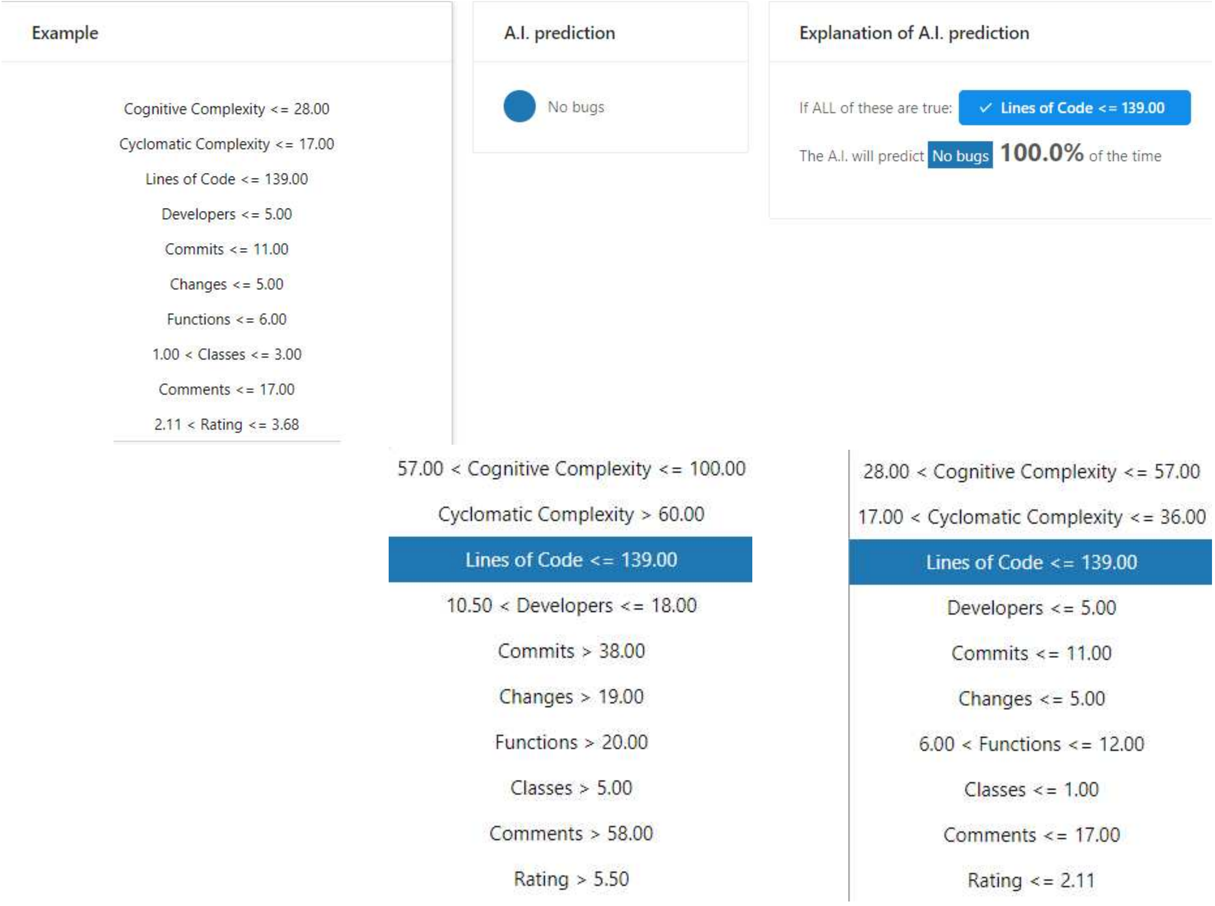
28.00 < Cognitive Complexity <= 57.00
Cyclomatic Complexity <= 17.00
Lines of Code <= 139.00
Developers <= 5.00
Commits <= 11.00
Changes <= 5.00
Functions <= 6.00
Classes <= 1.00
17.00 < Comments <= 34.00
3.68 < Rating <= 5.50

Figure 5.15: Anchor eXtreme Gradient Boosting Tree

5.7 Insertion in pipeline

At the end of this job it is important to insert all this file in the pipeline to work on it every time a modification is done by the developers. To do this it is enough to insert the whole python file on the one that regulates the flow of the pipeline (the one with the extension .yml that is cited in chapter 2). It is important to make this file interactive in real time with the rest of the project: in particular it is important to insert the values dynamically and to make a call to Git, Jira and the tools that calculate cyclomatic complexity, rating of the file and cognitive complexity to get data.

Since it is a test it has to be inserted in the test job of the pipeline and it has to be subordinated to every other job, in particular to the build job which find the bugs in the code and to the unit test part (that in this project are not presented, but that usually are), of course with the model (or models) that is chosen for the project; that could be one between Decision Tree, Random Forest, eXtreme Gradient Boosting Tree and Gradient



Example

Cognitive Complexity <= 28.00

Cyclomatic Complexity <= 17.00

Lines of Code <= 139.00

Developers <= 5.00

Commits <= 11.00

Changes <= 5.00

Functions <= 6.00

1.00 < Classes <= 3.00

Comments <= 17.00

2.11 < Rating <= 3.68

A.I. prediction

● No bugs

Explanation of A.I. prediction

If ALL of these are true: ✓ Lines of Code <= 139.00

The A.I. will predict No bugs **100.0%** of the time

57.00 < Cognitive Complexity <= 100.00

Cyclomatic Complexity > 60.00

Lines of Code <= 139.00

10.50 < Developers <= 18.00

Commits > 38.00

Changes > 19.00

Functions > 20.00

Classes > 5.00

Comments > 58.00

Rating > 5.50

28.00 < Cognitive Complexity <= 57.00

17.00 < Cyclomatic Complexity <= 36.00

Lines of Code <= 139.00

Developers <= 5.00

Commits <= 11.00

Changes <= 5.00

6.00 < Functions <= 12.00

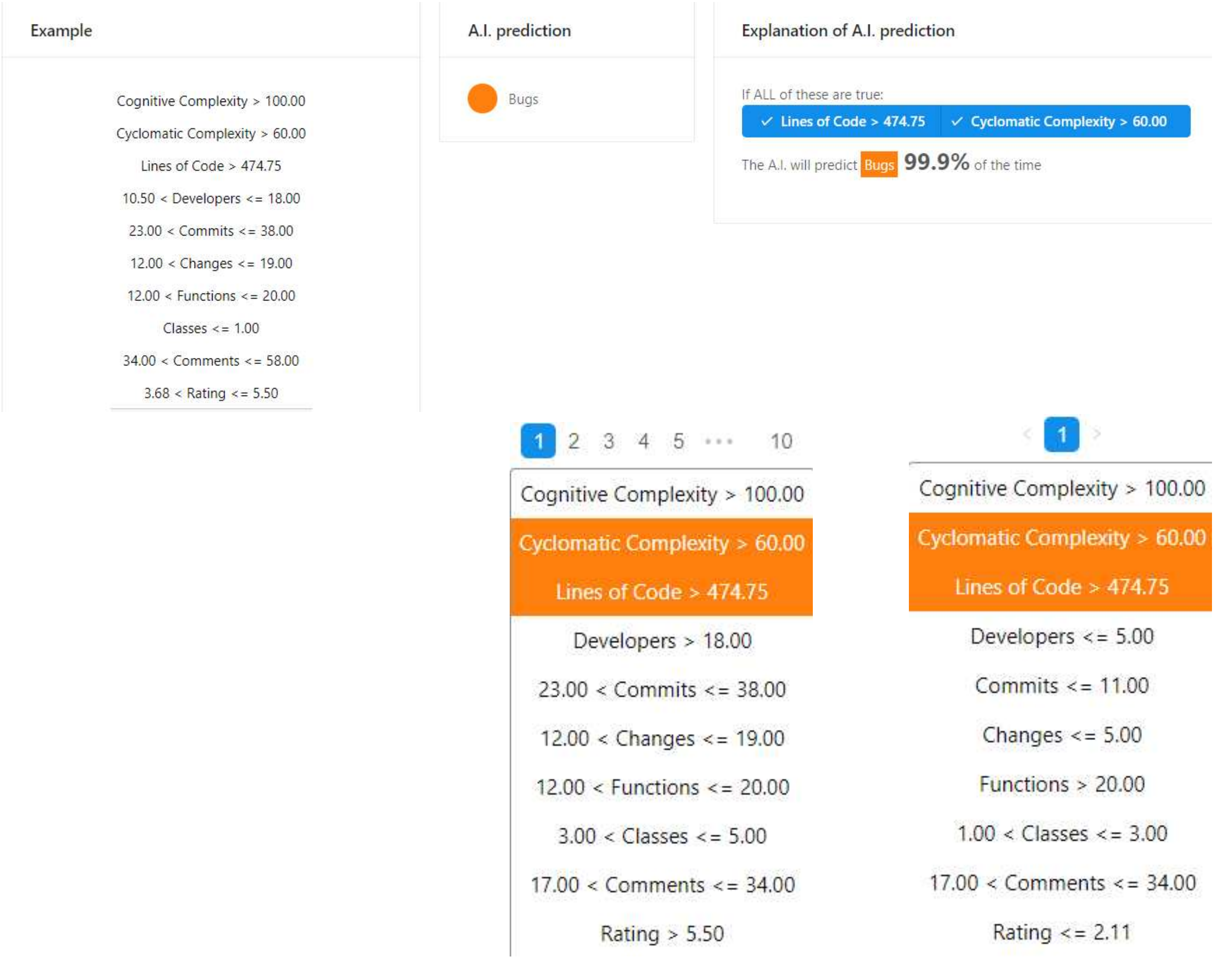
Classes <= 1.00

Comments <= 17.00

Rating <= 2.11

Figure 5.16: Anchor Random Forest

Boosting Machines that we have seen above that are almost equivalent. Then every time that a developer makes a modification the models work and the developers know which is the probability of finding a bug in the various files and have all the Explainable AI part available to see why a file could have bugs and to comprehend what developers can do to fix them.



Example

Cognitive Complexity > 100.00

Cyclomatic Complexity > 60.00

Lines of Code > 474.75

10.50 < Developers <= 18.00

23.00 < Commits <= 38.00

12.00 < Changes <= 19.00

12.00 < Functions <= 20.00

Classes <= 1.00

34.00 < Comments <= 58.00

3.68 < Rating <= 5.50

A.I. prediction

● Bugs

Explanation of A.I. prediction

If ALL of these are true:

✓ Lines of Code > 474.75 ✓ Cyclomatic Complexity > 60.00

The A.I. will predict Bugs 99.9% of the time

Cognitive Complexity > 100.00

Cyclomatic Complexity > 60.00

Lines of Code > 474.75

Developers > 18.00

23.00 < Commits <= 38.00

12.00 < Changes <= 19.00

12.00 < Functions <= 20.00

3.00 < Classes <= 5.00

17.00 < Comments <= 34.00

Rating > 5.50

(a) File with bugs

Cognitive Complexity > 100.00

Cyclomatic Complexity > 60.00

Lines of Code > 474.75

Developers <= 5.00

Commits <= 11.00

Changes <= 5.00

Functions > 20.00

1.00 < Classes <= 3.00

17.00 < Comments <= 34.00

Rating <= 2.11

(b) File with no bugs

Figure 5.17: Anchor Neural Network

Chapter 6

Evaluation

The objective of the thesis was to find which is the best model (at least for this project) to predict bug, to check if this model was good enough for a factory that works on software, to learn how it works with Explainable AI techniques and to insert in the pipeline. Of course since we have some different models and terms of comparison, the comparison can be done and it is in line with expectations. In this chapter we are going to delve into details on how we get to know that we reached our original objective.

First of all it is important to say that all the models worked very well and that they had very good numbers in terms of rate of false positives and in terms of Area under the curve. We saw that there is not a model that is the best for sure, but there are four equivalent models.

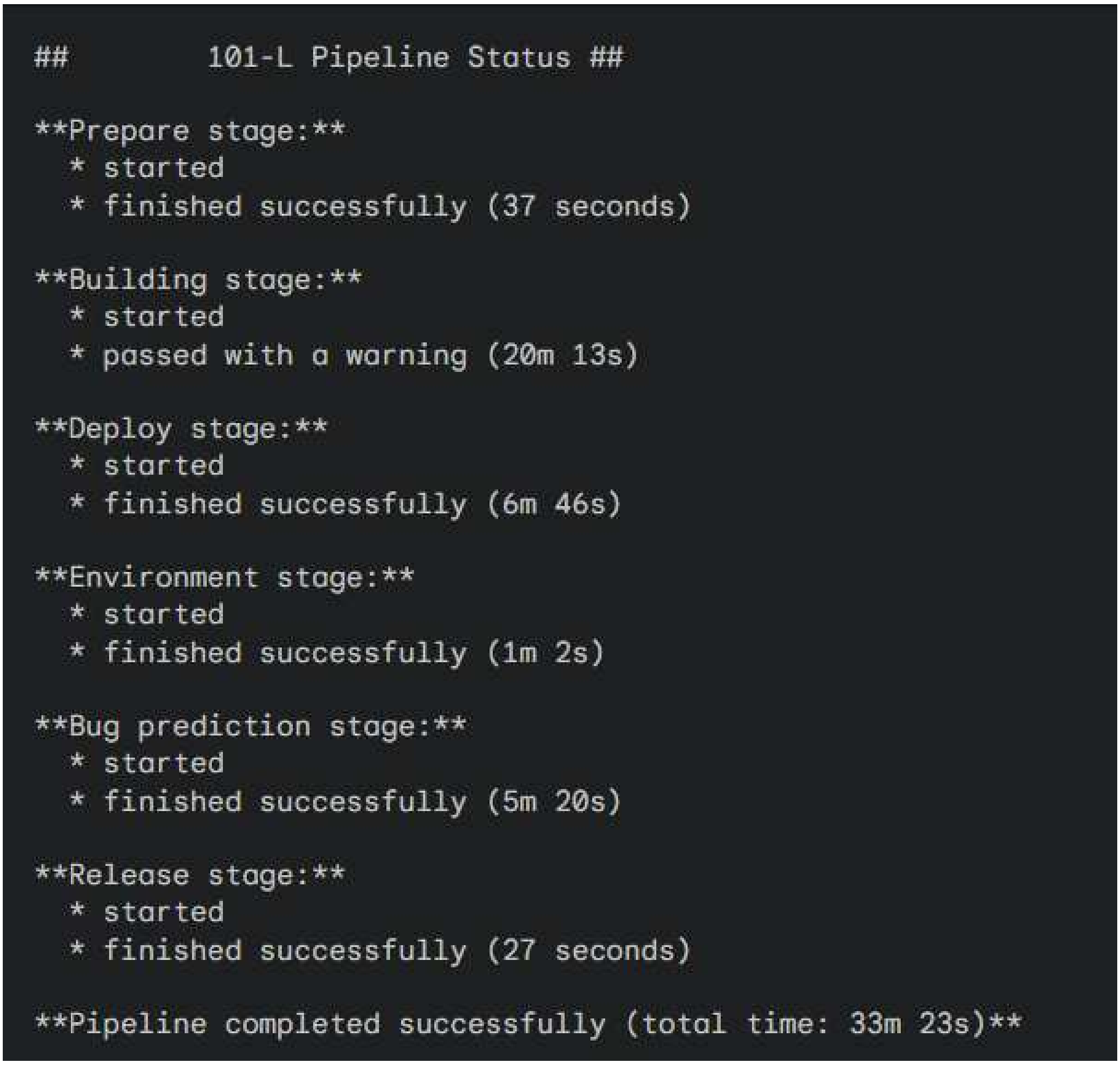
Before starting we expected that the black box model would be more precise and more appropriate with this amount of data, because they are usually more accurate. In fact it is obvious for instance, that a random forest or a Extreme Gradient Boosted Tree works more accurately than a Decision tree because they are an upgrade of the decision tree, using many trees within each model. We expected reliable models, but perhaps not to this extent, and we were surprised. In fact, previously we had tried to build the same model with only a quarter of the data (thus applying data augmentation of 500 data points instead of 2000), and the models had an AUC of approximately 0.999, rounded to the first three decimal places. This led to the decision to further increase the data to introduce a discrepancy, although we expected, as described in the previous chapters, not to be very large, in order to have a point of comparison.

The cross-validation was an important aspect for this model and it makes the results more truthful: since we cross-validate the data, since we perform cross-validation on the data, it is not taken in order but in a random order. This means that by trying and retrying to evaluate the models, they will always work on the file in a different way and consequently by doing more experiments we will have a truthful view of which model works best on the data. This has been done: data from different simulations have been collected and the average has been calculated. In this way the result is not too much influenced by bias.

The Explainable AI explains us that every model works based on a logic and all the logic are similar, in fact for all the models the principal component is the number of lines of the file, which was also the principal component calculated to cluster the dataset. This means that there is coherency between the way the models reason and so the results are comparable, and consequently we can figure out which is the best model, at least for this project. There is also a correspondence between the SHAP technique, that is a global technique, and the LIME and Anchor techniques that are punctual. This is good because it means that the examples that LIME and Anchor take in consideration is not a "tail" example, but an example that behaves in a similar way to the other that could be taken.

At the end the file was inserted in the pipeline and it was chosen for the file the Decision Tree model because, since it was almost equivalent in terms of precision and AUC respect to the best, but it was faster than eXtreme Gradient Boosting Tree, Random Forests and Gradient Boosting Machine. It works well, in particular it is bug-free and it last in a short time.

In fact we can see from the figure 6.1 (the name of the factory is hidden) the output of Linux when the pipeline is launched; since unit tests are completely absent (we will focus on this in chapter 7) there is not the test stage, that would have contained that part. Moreover the time to complete the bug prediction stage is less than the one of the deploy stage, so it is a short time that does not make the developer wait too much. We can also see that the total time of the pipeline does not coincide with the sum of times; in fact the Environment stage does not depend by the Building stage, so they are done at the same time and the consequence is that it is time-saving. By the way it is the only stage independent, there would have been the stage test that is independent of the deploy stage, but here it is not present for the reason explained above.



```
##        101-L Pipeline Status ##

**Prepare stage:**
  * started
  * finished successfully (37 seconds)

**Building stage:**
  * started
  * passed with a warning (20m 13s)

**Deploy stage:**
  * started
  * finished successfully (6m 46s)

**Environment stage:**
  * started
  * finished successfully (1m 2s)

**Bug prediction stage:**
  * started
  * finished successfully (5m 20s)

**Release stage:**
  * started
  * finished successfully (27 seconds)

**Pipeline completed successfully (total time: 33m 23s)**
```

Figure 6.1: Output of the pipeline

Chapter 7

Discussion

In this work there have been positive and negative aspects, some of them depending on poverty of resources in terms of tools that could be used, some of them depending on the original project. In this chapter we will write about them and we will explain in details if in this thesis there are positive aspects or negative aspects and why and which difficulties we encountered and how we managed to overcome them.

7.1 Difficulties: what could have done better?

In this section we will discuss about the difficulties and the negative aspects, dividing them in subsection every step and focusing on how they impact in the correctness and robustness of the model and the ways that can be used to overcome them.

7.1.1 Data collection

First of all let us start with the first difficulty: how collect data. In the previous chapters, we briefly discussed the challenges encountered during data collection, but now we are going looking deeper into these obstacles.

The main problem is the poverty of the data: we had too few data in terms of quantity of files and in terms of quantity of data about files. Now we are going to see what data could be studied by the model and the reason why they are not present in our model.

First of all, the coding approach was not the best: in fact, as it is written in the previous chapters, it is very important to detect bugs as soon as possible and the best way to do it before using these models is to create unit tests for the code. In fact they are indispensable to detect bugs that do not make work the software, so they are crucial to do a first screening. Moreover it is important for prediction bug models to know which is the part screened by the unit test to do a better job of prediction; of course without this screening the model has to work on more bugs and consequently it could lose some power. By the way, the project was done without considering the possibility of building a model to predict bugs, so unit tests were considered non-essential.

Moreover the documentation was insufficient: it is important not only for developers to make easy the study of the code in a successive bug fixing session, but also for the model to have more data and to be more precise in finding bugs. In fact the documentation for every file is important to detect the probability of presenting a bug for every file: more documentation is written about a file, more is the probability of being bug-free for that file.

Another important challenge of the data collection is that it was difficult to find the metrics. In fact many online tools are available by paying important sums or they were not available for students, so since the money budget was not high we could not use these tools. By the way there was a free-tool that returned some secondary metrics of the software, such as cognitive complexity, cyclomatic complexity and rating of the file. However since the files of the project are written half in Python and half in C++, there is a potential compatibility issue. The tools that calculate complexity metrics for files written in C++ and Python analyze the source code to identify control flow structures, such as loops and conditions, and compute the number of possible execution paths through the code. Complexity metrics, like cyclomatic complexity and cognitive complexity, are then calculated using specific algorithms tailored to each metric.

However, the complexity values generated by these tools may not be directly comparable between files written in C++ and those written in Python due to intrinsic differences in the two languages and their structures. This is due to several reasons, first of all C++ and Python have different syntax and semantics, meaning the same code construct could have a different impact on complexity in the two languages. For example, pointer management and memory handling in C++ may affect complexity differently compared to object management and Python's automatic memory management.

Moreover, while both languages support the same control flow structures (like loops and conditions), the specific implementations of these structures can vary, affecting the number of possible execution paths and thus the calculated complexity.

Handling of Libraries and Dependencies: C++ and Python often use different libraries and frameworks, which can introduce additional complexity to the code. Calls to library functions, use of external APIs, and dependency management may impact complexity differently in the two languages.

Another difference between languages is in programming styles and development practices and they can influence code complexity. For instance, C++ is often used for low-level application and embedded systems development, while Python is more common in high-level development environments and web application development.

Therefore, while the complexity values calculated by the tools can provide a measure of the relative complexity of files in each language, it may not be directly possible to compare them between C++ and Python due to these intrinsic differences. It is important to consider the project context and language specifics when interpreting and comparing complexity metrics.

Coming back to the original problem, the decision taken about this problem is that we trust in the tool, supposing that the results that gives to us are comparable.

As it is written above the trickiest problem was that the number of files was too few. In fact there were only eighty-six files, that had to be used to train models and test them and it was an insufficient number. This problem could be faced in various ways but all these ways wold have an impact on the final product. For example one idea could be working on something smaller than files (methods or classes for example), but this would have made the research of data more difficult and would have transform data in less useful because many classes and methods could not contain comments and it could have made the data of the files very similar to each other with many repetitions and so we could not have much more data. Moreover also for developers is better to concentrate on a file than on a class to find bugs because he would have a good vision of the whole file. To overcome this issue, it was decided to create a small Python script for data augmentation based on the Sci-Py library, which increased the number of samples in a non-random way, but following the distribution of the original data. This decision helped us to have of course more data, but some of them are not original data but similar to the original data. It is not perfect of course, it would have been better if we had a project of more file or more projects, but this seemed the best way to compensate for the numerical need for data while maintaining the veracity of the original sample as much as possible. In fact the additional data points that, as long as possible, had the same statistical distribution as the original dataset and so they could be considered effectively metrics of a real file. The number of data we decided to have was five hundred, with this script was possible to do it. Another reason not to work on classes is the increase in false positives. This would indeed be a significant issue, as described earlier, because the number of false positives is a major determinant of model quality. It would almost guarantee to the developer that the bug is located in a specific place, consequently facilitating its rapid detection and reducing the time spent on each one.

7.1.2 Error log

Another significant challenge encountered was the difficulty in accessing the comprehensive error log for each individual file within the project. While leveraging Jira, we managed to uncover a trove of past bugs reported by the validation team. However, due the project's structure it was not indicated which file contained the bug. Although these bugs were logged against the final product, the convoluted nature of the project's architecture made it arduous to pinpoint the exact file harboring the reported bug.

Furthermore, the absence of a detailed record regarding the bug resolution process compounded the issue, leaving us in the dark about when and where the bug was rectified. To address this glaring gap in our data, we took proactive measures by augmenting the dataset with supplementary entries that offer insights into the historical bug occurrences associated with each file. Indeed, metrics such as the number of individuals who have worked on the file, the number of file commits, or the revision history, while not replacing the count of past bugs, can provide a similar description. Consequently, the entry for past bugs exhibits a high correlation with these metrics. Therefore, while not entirely redundant, it would not have been one of the primary components discussed in the previous chapters. Instead, it would have been almost a dependent component and, as a result, would have been given little consideration by the predictive model.

7.2 Positive aspects

In this section we will find out the positive aspects of this work and we will introduce the next chapter of the related works.

First of all it is important to underline that this thesis is based on a real project and so it is based on real data and on a real problem. This is important because it makes this work spendable for a project in the "real world" and it could be an important application in the software development world.

Another positive aspect is certainly the fact that the various models are compared using cross-validation, therefore an important work is done under the aspect of bias reduction, as explained in the previous chapters. This leads to being a well-done work under the aspect of model evaluation and consequently leads this ranking of models to be taken into consideration for future work, in case some company should choose a model to control the quality of their software and to try to improve it through the research and removal of bugs.

Furthermore, the Explainable AI part is very useful for explaining what the bug search is based on and, if the model is reliable, it would also help developers to write code differently, being careful about the parameters that lead to more errors in the software files. In fact, new metrics or parameters could be added to the model and see if they have an important influence or not, so that, in addition to carrying out a study on how to find bugs in the software, a statistical study can be conducted on which metrics and which aspects of the code lead to more errors. In fact Explainable AI could help developers to understand the complex relationships between different parts of a software system. This could help them to identify potential problems before they occur and it could also help developers to debug software more effectively. By understanding how the software works, they could more easily identify the root cause of a bug. The immediate consequence is an improvement of the performance of software by understanding how the software uses resources, in fact in this way developers could identify areas where it could be optimized and they could also create more user-friendly software, understanding how users interact with the software, designing it to be more intuitive and easier to use.

Chapter 8

Conclusion and Further Studies

In summary, in this thesis we attempted to solve the problem of bugs within software. We extracted software metrics, but since they were not enough, we increased their number using data augmentation techniques. Then, we split the dataset into two parts: files with bugs and bug-free files. We trained various models and tested them to obtain the results. Finally, we used Explainable AI techniques to understand how the various models worked and integrated everything into the pipeline.

We have seen that there are some models that are more precise than the others (in particular

However, this is only a first step. The models could be improved, and above all, better data could be obtained, as mentioned in the previous chapters. Furthermore, one could look for a way for the models to work primarily not on static metrics, such as the number of lines, but rather on dynamic metrics, such as the number of changes since the last commit or the number of people who have worked on it since the last bug-free commit. This is because it shows the possibility that a file may have a bug between two commits, assuming that it was bug-free before the previous commit.

It is also possible to work on a non-deterministic prediction, i.e., not to work on a model that only outputs whether a file has a bug or not, but rather to divide the whole thing into more nuances (definitely bug-free, low probability of bug in the file, medium probability of bug in the file, high probability of bug in the file, definitely has a bug in the file) to have a more complete view.

Finally, one could work on comparing these models between projects and not between files to see which ones are more truthful and perhaps also try with a finer granularity (e.g., at the module level) to see if there is a possibility of further improving the ”zoom” without affecting the excellent precision figures obtained in this experiment.

We hope that this thesis could be a springboard for the study of other explainable AI techniques and for improving code quality.

Bibliography

[1] Chakkrit Tantithamthavorn and Jirayus Jiarpakdee. *Explainable AI for Software Engineering*. Monash University, 2021. Retrieved 2021-05-17.

[2] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[3] Emelie Engström, Per Runeson, and Greger Wikstrand. An empirical evaluation of regression testing based on fix-cache recommendations. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 75–78, 2010.

[4] Mijung Kim, Jaechang Nam, Jaehyuk Yeon, Soonhwang Choi, and Sunghun Kim. Remi: defect prediction for efficient api testing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 990–993, New York, NY, USA, 2015. Association for Computing Machinery.

[5] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken-ichi Matsumoto, Bram Adams, and Ahmed E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

[6] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Bug prediction based on fine-grained module histories. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 200–210, 2012.

[7] Kaffayatullah Khan, Waqas Ahmad, Muhammad Amin, Ayaz Ahmad, Sohaib Nazar, and Anas Alabdullah. Compressive strength estimation of steel-fiber-reinforced concrete and raw material interactions using advanced algorithms. *Polymers*, 14, 07 2022.

[8] Stefano Biora. *Analisi della complessit'a di tecniche di offuscamento*. PhD thesis, POLITECNICO DI TORINO, July 2019. `https://webthesis.biblio.polito.it/11505/1/tesi.pdf`.

[9] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, page 284–292, New York, NY, USA, 2005. Association for Computing Machinery.

[10] Ramesh Ponnala and Dr REDDY. Software defect prediction using machine learning algorithms: Current state of the art. *Solid State Technology*, 64:6541–6556, 05 2021.

[11] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code! examining the effects of ownership on software quality. pages 4–14, 09 2011.

[12] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 432–441. IEEE Press, 2013.

[13] Elena N. Akimova, Alexander Yu. Bersenev, Artem A. Deikov, Konstantin S. Kobylkin, Anton V. Konygin, Ilya P. Mezentsev, and Vladimir E. Misilov. A survey on software defect prediction using deep learning. *Mathematics*, 9(11), 2021.

[14] Jalaj Pachouly, Swati Ahirrao, Ketan Kotecha, Ganeshsree Selvachandran, and Ajith Abraham. A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools. *Engineering Applications of Artificial Intelligence*, 111:104773, 2022.

[15] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*, pages 9–9, 2007.

[16] Olivier Vandecruys, David Martens, Bart Baesens, Christophe Mues, Manu De Backer, and Raf Haesen. Mining software repositories for comprehensible software fault prediction models. *Journal of Systems and Software*, 81(5):823–839, 2008. Software Process and Product Measurement.

[17] Roberto Confalonieri, Ludovik Coba, Benedikt Wagner, and Tarek R. Besold. A historical perspective of explainable artificial intelligence. *WIREs Data Mining and Knowledge Discovery*, 11(1):e1391, 2021.

[18] Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador Garcia, Sergio Gil-Lopez, Daniel Molina, Richard Benjamins, Raja Chatila, and Francisco Herrera. Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Information Fusion*, 58:82–115, 2020.