Politecnico di Torino

Master's Degree Course in Computer Engineering

Master's Degree Thesis

Automatic Classification of Software Issue Report

Supervisor:
Prof. Luca Ardito

Candidate:
Andrea Simone Foderaro

Co-supervisor:
Prof. Maurizio Morisio

Academic year 2019/20
Torino

Abstract

***Context*:** Correctly classifying new software bugs is a time consuming, expensive, and error-prone process as it tends to be mostly manual. One of the main causes of this phenomenon is bound to the fact that this process is human based, and, as consequence, this usually leads to frequent mis-classifications. Developing software able to automatically identify bugs with high level of accuracy would result in significant resource and time optimization.

***Goal*:** The aim of this thesis is building a tool that will automatically classify the bug and assign it to the correct class. The tool developed is designed to be able to assign a bug to the right developers by analysing a file related to the bug, called *bug report*. In more details, the model assigns the bug to the most probable class and consequently to developers specialised for that class.

***Method*:** The tool works on Mozilla bugs taken from the BugZilla database and it focuses on classifying new bugs. Every bug is characterised by a bug report, which contains all the information on the bug from the time it was issued to the time it was solved. The model is based on the BugBug algorithm, with some modifications to best fit the purpose of the project. The tool creates either a *OneVsRest* or a *Binary* classifier using a dataset of labeled bugs.

***Results*:** The project was tested on two different scenarios: one single class, considered as the positive class, against all the others, labeled as negative class, and a multiple classes situation. The former case reached an accuracy higher than 60% for almost all labels, while the latter reached an accuracy lower than 75%, roughly 73%.

***Conclusions*:** The tool proved to be successful to fulfil the goal, but the accuracy was not optimal in all the cases. The accuracy obtained using a SVM model indeed never exceeded 74%, making it a not perfect fit for concrete application. On the other hand, the binary classifier showed an acceptable accuracy for some classes, even if it requires more time to classify a bug. Moreover, both the classifiers can be easily extended to any generic bug coming from other sources, to enlarge its field of applicability.

Contents

List of Figures

List of Tables

Chapter 1

Project background and related works

1.1 Bug classification's introduction

The bug classification activity is of paramount importance to produce bug-free software. Software are human production, hence they will never be free of bugs, and this is the reason why the bug classification process exists. The exponential growth in size of programs and applications made bug removal an expensive and long task.

The modern development in machine learning and artificial intelligence could provide an extremely useful tool to reduce the bug classification time process and costs. The development of an automatic bug classifier would speed up the classification and consequently the bug removal work.

A bug classification is characterised by a sequence of steps. Once a bug is reported, developers are requested to analyse the *bug report* and label the bug, this process is also called *bug triaging*. After the bug has been classified it is assigned to the most qualified developers for that specific type. Each classification generally involves more than one person classifying the bug separately and then discuss the classification, this activity could take days.

As bug classification is mainly focused in assigning the bug removal work to the right developer or team, reaching an high accuracy in the classification becomes the most important and time-consuming step (Akila et al., 2015), especially since

a wrongly classified bug might cause problem during the removal process.

Classifying bugs automatically, using a machine learning algorithm, would be effective both in the exploitation of resources and time but it also would reduce, and eventually remove, the human-related error component, i.e., humans make mistakes and a bug wrongly classified would be assigned to the wrong developers and would result in more time to remove it. Moreover, the maintenance of such code would be trivial in most of the cases.

In this thesis is argued that removing errors made by developers in the process of understanding the bug type can be beneficial in the process. The tool described will properly identify the developer who should be assigned to the debugging, speeding-up the bug analysis and resolution process. Removing this kind of errors highly affects the quality of the classifier and therefore it is of mandatory importance to reach an high accuracy of the model, since otherwise additional work will be needed to reassign the bug to the correct developers, enlarging the time to remove the bug.

1.1.1 Example of bug classification

In Figures 1.1, 1.2, 1.3 and 1.4 all the different information one can find in a *bug report* are presented. All the pictures are from bugs taken from the *BugZilla* database, which is the database employed in this thesis. From this example the main steps of a bug classification can be analysed.

The bug report has three main sections:

- title of the bug, is the name with which the bug is filed and it can be meaningful for the classification since in many cases synthesizes perfectly the problem at hand (Figure 1.1)

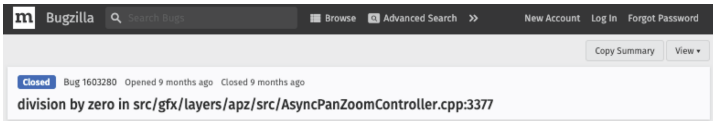


Figure 1.1: ex of *bug report* title

- a small part contains the main characteristics of the bug: category, product, component, type, priority and severity, tracking point, the people to which it is assigned, references, details and useful flags (Figure 1.2)

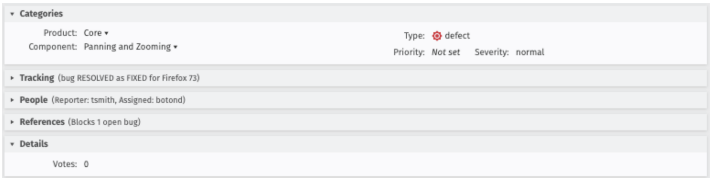
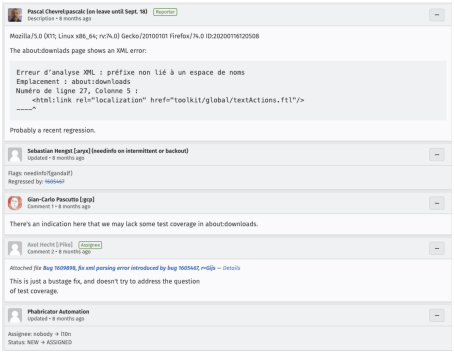


Figure 1.2: ex of *bug report* information

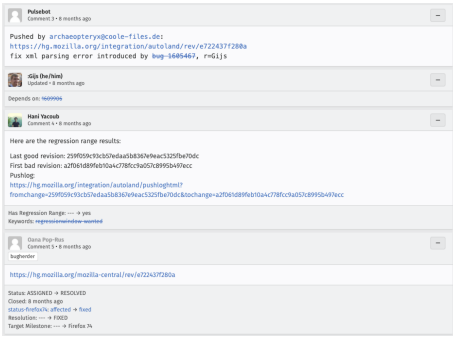
- another portion is filled with the list of links to the attachments, that in most of the cases are pictures or videos of the error presented in the report, they can also be links to important sections of code regarding the bug (Figure 1.3)

Figure 1.3: ex of *bug report* attachments

- the last part is devoted to comments. Here is where the developers assigned to the bug can exchange ideas and insights on the bug under consideration. Each comment can be a consideration or insight on the bug, a piece of code or output of the program, a link to a portion of code or an attachment like a picture or video (Figure 1.4)



(a) comments



(b) comments

Figure 1.4: ex of *bug report* comments

During the bug removal process the bug report may change especially in the comment section since comments are always added but also the information section may change as well, particularly its priority and severity, but also the developers

assigned to it may change, i.e., the bug was wrongly classified.

In case the bug is wrongly classified the wrong developers will receive it and they will signal the mistake. This process lead to a new classification and a new assignation of the bug, losing time and resources. This makes avoiding such errors of mandatory importance.

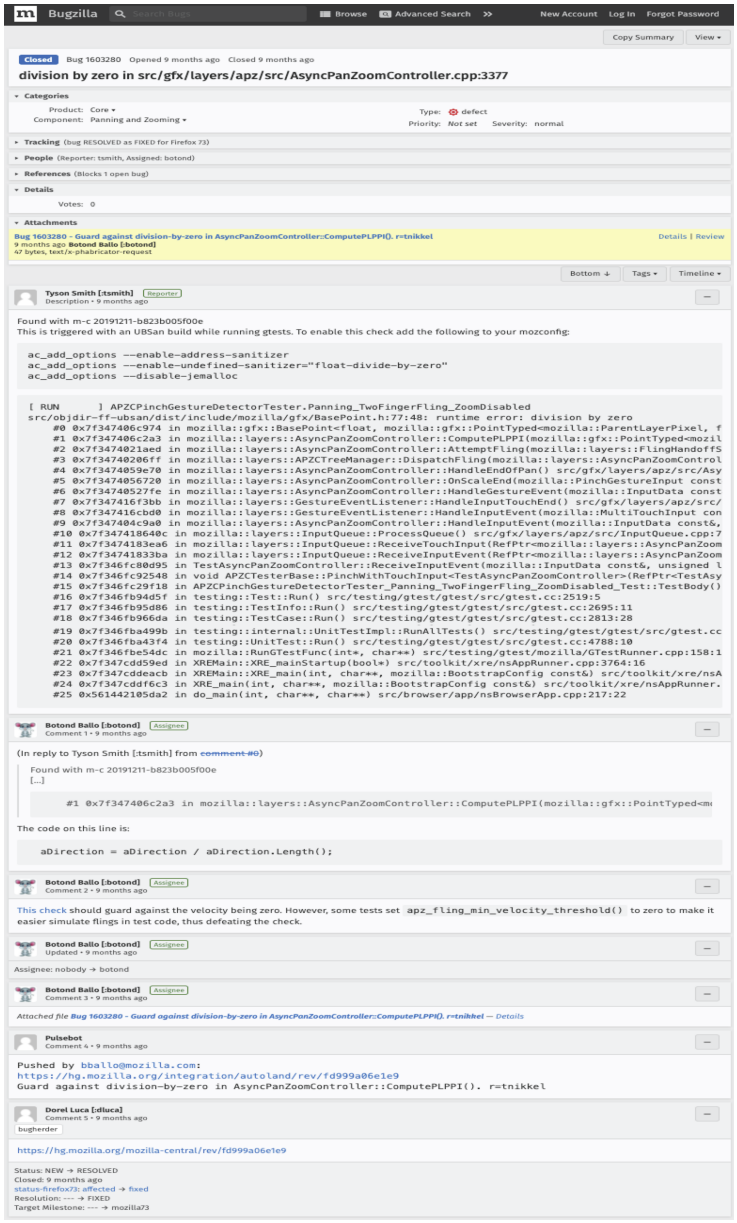


Figure 1.5: *ex of bug report*

1.2 Steps of bug classification

Classifying a bug is a long and challenging procedure. Here are all the steps involved:

- two or more parties are created
- each of the party analyses the same bug report, ex. in Figure 1.5
- the parties classify separately the bug
- a discussion is made between the parties to decide the final classification of the bug

1.2.1 Costs of labelling a bug

In the bug triaging process, cost depends on the time needed to classify the bug correctly and the amount of developers required to achieve a correct classification. It goes without saying that more complex a bug is to describe and more time it will require to be classified. At the same time involving more people for the task will certainly reduce time but it will increase the cost of personnel. In conclusion automatic bug classification will have a fundamental role in cost reduction.

To have an idea about the costs we are talking about, just think that the removal of a bug during the production phase can cost up to $ 10,000.00 [1]. In more details part of that amount is due to the bug *triaging* phase, when the bug is labeled and sent to the correct developers. Our tool aims at removing this cost and the time involved in this task.

Moreover the cost increases if possible mistakes are considered in the classification process. Indeed if a bug is assigned to the wrong class then wrong developers will be asked to solve it, resulting in a waste of time and resources.

1.3 Introduction to the taxonomy

For what it concerns the set of labels considered in this thesis, the work of Catolino, Palomba, Zaidman, and Ferrucci was explored. They defined a unified view for the

[1] Celerity: The true cost of a Software Bug

classes of bugs in the *Bugzilla* database.

The labels are divided in two sets: general and specific. The general classes are: API, Database-related, Development, GUI-related, Network Usage, Performance, Program Anomaly and Security.

Moreover, the specific labels are sub-categories of the general ones and they are presented in Table 1.1 with their associated general label. A deeper analysis of these classes is provided in Section 2.2.

Category	Sub-categories
API	Add-on or Plug-in Incompatibility Incompatibility Permission/Deprecation Web Incompatibility
Database-related	
Development	Compile Test code
GUI-related	
Network Usage	
Performance	
Program Anomaly	Crash Hang Incorrect Rendering Wrong Functionality
Security	

Table 1.1: Taxonomy Table

1.4 Goals

The aim of this thesis work is to provide an automated solution to bug classification problem, reducing time and costs of the job.

In particular, an example of use case can be described by any bug report that has to be classified and consequently resolved: [bug 1627108][2] the report involves a wrong

[2]Bug Report 1627108

selection of a thread in a browser toolbox; in this example the classification was drawn from the first comment, explaining what the program should have done and what it actually did, i.e., Wrong functionality issue (Section 2.2). In many other cases the classification is not clear at first sight but requires a deep analysis and discussion.

The tool provided would use all the information inside the report, primarily title and the comments, in order to provide the correct classification of the bug, reducing time and costs of the job.

1.5 Previous works and existing tools

The work proposed here revolves around triaging bugs according to their type with the goal of supporting and eventually speed-up this activity.

This section will provide an overview of the previous studies and existing tools in the field of automated bug classification. A comprehensive overview of the research conducted in the context of bug triaging is presented by Zhang et al. (2016).

- a machine learning model to discriminate between bugs and new features requests was defined by Antoniol et al. (2008). The model was able to discriminate the two with a precision of 77% and a recall of 82%.

- to classify the impact of bugs, Hernández-González et al. (2018) proposed an approach that with the empirical study conducted on two systems, Compendium and Mozilla, showed good results. The approach was designed according to the ODC taxonomy (Chillarege et al. (2018)).

- AutoODC, again based on the ODC, for automatic ODC classification, developed by Huang et al. (2015). This tool cast the problem in a supervised text classification. This approach was augmented by the integration of experts' ODC experience and domain knowledge. They built two models trained with two different classifiers such as Naive Bayes and Support Vector Machine on a larger defect list extracted from FileZilla.

- in 2012 Thung, Lo, and Jiang developed a classification based approach that could automatically classify defects into three super-categories, that are

comprised of ODC categories: control and data flow, structural, and non-functional.

- the previous tool was extended in 2015 (Thung, Le, and Lo), where the defect categorisation was enlarged. In more details, they combined active learning and semi-supervised learning algorithms to automatically categorize defects. They evaluated their approach on 500 defects collected from JIRA repositories of three software systems.
- analyzing the natural-language description of bug reports, evaluating their solution on 4 datasets, e.g., Linux, MySQL (for a total of 809 bug reports), Xia et al. (2014) was able to categorize defects into fault trigger categories using a text mining technique.
- using LDA Nagwani, Verma, and Mehta in 2013 proposed a method for generating taxonomic terms in order to label software bugs.
- in 2016 Zhou, Tong, Gu, and Gall combined structured data (e.g., priority and severity) with text mining on the defect descriptions to identify corrective bugs.
- in order the have bugs assigned to the right developers, text-categorization based machine learning techniques have been applied for bug triaging activities (Murphy and Cubranic (2004), Javed et al. (2012)).

Our work is mainly based on the concepts developed by Murphy and Cubranic and Javed et al.: use text-categorization and text mining machine learning techniques in order to correctly label a bug. Our tool does not take in consideration new features as bug report, those bug reports were removed from the dataset (see Section 2.3). In addition our taxonomy was developed by Catolino et al. and will be presented in full details in the next chapter.

1.5.1 BugBug

The *BugBug*[3] algorithm was the starting point of this thesis. This program was developed by Marco Castelluccio[4] and other Mozilla developers in order to get bugs

[3]Teaching Machines to triage Firefox bugs

[4]Marco Castelluccio

in front of the right Firefox engineers. As already mentioned, by presenting new bugs quickly to triage owners eventually the turnaround time to fix new issues decreases.

This tool was the follow up work of a previous project that used a technique to differentiate between bug reports and feature requests.

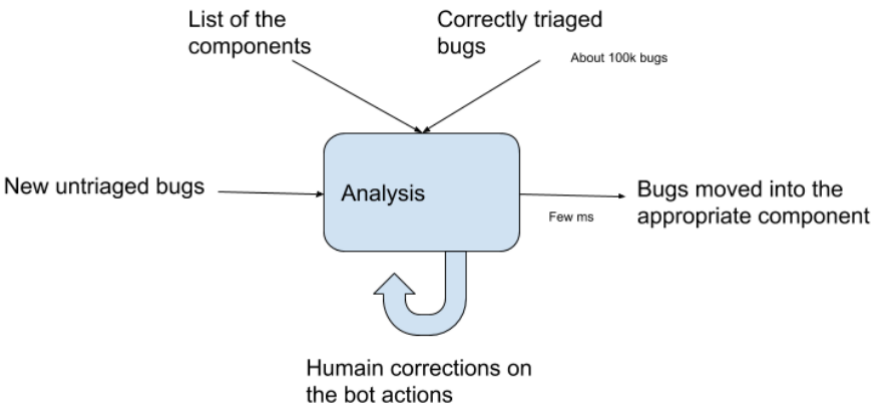


Figure 1.6: *BugBug* high-level training and operations

The training set for this project was large: two decades worth of bugs which have been review by "Mozillians" and assigned to products and components.

A "roll back" of the bug report to the time it was originally filed was performed, this was done in order to train the model using the same amount of information it would have during real operation. In this way any change to the bug after triage has been removed, these information are indeed inaccessible during real operation.

The taxonomy has also been modified since in the past 2 years, out of 396 components, only 225 had more than 49 bugs. In deeper analysis, all the components that had a number of bugs that was at least 1% of the number of bugs of the largest component were selected. This implies that only that subset is meaningful and can be analysed.

The features under analysis were the title, the first comment, and the keywords/flags that characterise the bug, meanwhile the training was performed using an

XGBoost[5] model.

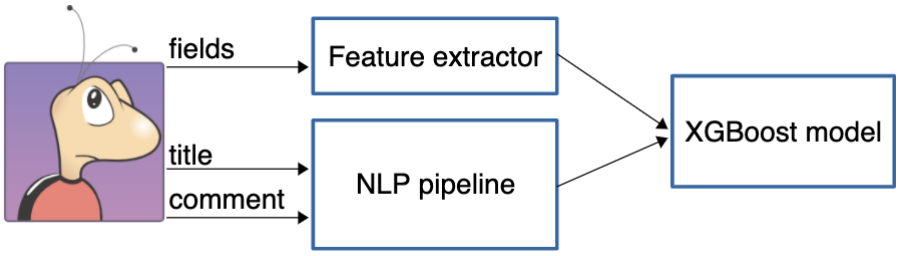


Figure 1.7: *BugBug* high-level model

In order to avoid wrong labelling, the assignment of the bug is performed only when the model has a confidence higher than a certain threshold. Using a 60% confidence threshold, the model ended up having a very low false positive ratio (it had a precision greater than 80%, using a validation set of bugs that were triaged between December 2018 and March 2019).

Tests showed that training a model on a 6-core machine with 32 GB of RAM and using a dataset of around 100,000 bugs (more than two years of data) takes roughly 40 minutes. Once the model has been produced, it label a new bug in matter of milliseconds, it never pauses, meaning that its always ready to act. The tool's classification is much faster than manual assignment, that takes around a week on average.

This thesis tool uses, as already said, BugBug's algorithm has starting point. Some changes were performed, especially in the feature selection component and in the taxonomy considered for the classification. On the other hand the library of the classifier, XGBoost (described in Section 3.1.2), was left unchanged at the beginning.

Here only a small part of the BugBug project is presented, it indeed contains 18 different classifier: assignee, backout, bugtype, component, defect vs enhancement vs task, defect, devdocneeded, duplicate, qaneeded, regression vs non-regression, regressionrange, regressor, spam, stepstoreproduce, testfailure, testselect, tracking

[5]XGBoost Documentation

and uplift. Each of them has a specific classification purpose and they will be better explained in Chapter 3. Now a simple introduction of the employed *BugType* classifier will be presented.

The BugType algorithm derives from the *BugModel*[6] which itself inherits from the *Model*[7]. The *Model* defines the main characteristics of the basic classifier used in BugBug. Meanwhile, the *BugModel* adds a database and a function to generate items from such database.

The *BugType* model inherits everything from the two aforementioned models and adds components more consistent with its purposes, i.e., label bugs according to their type. Moreover, the model extracts 16 features out of all the available ones, performs some transformations on them and it applies a *OneVsRest* classifier.

[6]BugModel GitHub

[7]Model GitHub

The remainder of this thesis is structured as follows:

- Chapter 2 will cover the definition of the characteristics of the tool, along with the explanation of the process to build the dataset;
- Chapter 3 describes the approaches followed in our work, OneVsRest and Binary classifier;
- Chapter 4 covers the results obtained by all the experiments done;
- Chapter 5 presents our conclusions and suggestions for possible future works.

Chapter 2

Architecture and Design

The tool presented has been developed to support developers in the long and expensive task of labelling a bug in order to assign it to the correct team or person. The tool is written in Python and can work in two different configurations:

- one that performs a binary classification (in Section 3.3) against a specific class (see Taxonomy in Section 2.2)
- the other uses a multi-class multi-label OneVsRest classifier (in Section 3.2 and Section 3.4)

Both the configurations have two modes: one assigning the bug to a general class and the other to a sub-category (explained in Section 2.2), when possible. The configuration and mode are chosen using the arguments passed to the algorithm from the command line.

In particular, the architecture is specified by three main components:

1. A **Feature Extractor**, which is one of the main components in a machine learning algorithm. Since the input data may have too many attributes, some of them are removed in order to avoid the curse of dimensionality. It takes as input all the available features and returns a subset of them. In this project the selected features were 16.
2. A **Column Transformer**, that comes from the scikit-learn *"sklearn"* library[1]

[1] scikit-learn

and allows to apply transformers to column array or to pandas[2] DataFrames. A transformer is a function that takes as input all the values of an attribute and applies a function to those values in order to return a new list of values. A *Column Transformer* allows different columns or column subsets of the input to be transformed separately and the features generated by each transformer will be concatenated to form a single feature space.

This and the first components are organised in a *pipeline*. A pipeline is composed of a list of transformers, two in our case, and eventually terminated by an estimator, absent in our case.

3. And a **Classifier**. This is the last and arguably most important component, is the one responsible for the classification of the bug. It receives as input the output of the pipeline and outputs a model. In this thesis tool it is possible to select between a OneVsRest and a Binary classifier.

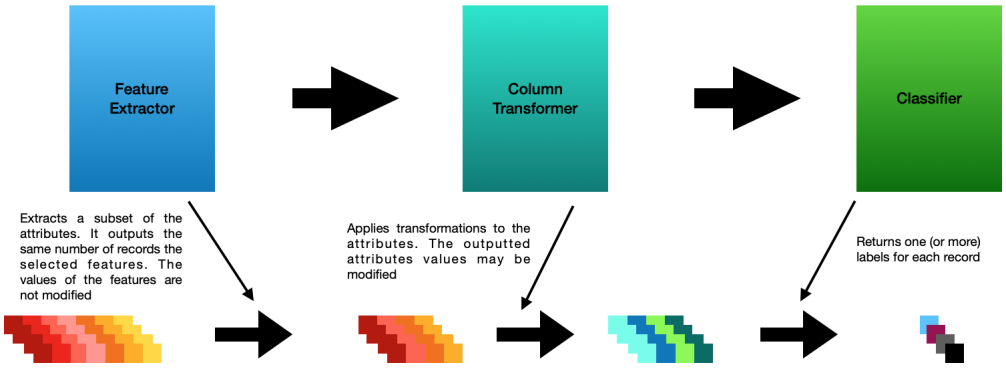


Figure 2.1: High level view of the basic model

2.1 Analysis of a bug

The first step towards this thesis goal was to understand which were the important information that allow a human to specifically label a bug. In order to do this almost 250 bugs were analysed, discussion were covered on them and a common ground was defined for the classification. This work followed again the footsteps

[2]pandas

of Fabio and Gemma (2019) and everything was done following the procedure explained in Section 2.3.

The starting point was the example of a classification of a single bug (explained in Section 2.3.1). In more details learning where to find the right information was of paramount importance both at the beginning and during the entire process. Moreover, the understanding of the main components and information of a *bug report* was of major value. In order to give weekly feed-backs to the rest of the team, the bug were analysed in batches (described in Section 2.3.2) of 50 or 100 bugs and a small review on them was provided at the end of each batch analysis.

Second point of the analysis was the discussion of the classifications. The two human classifier performed the labelling separately, in such a way there was no conditioning in the classification. The two teammates, after the private classification, had to discuss every bug label with each other and provide a final classification for each bug in the batch. Furthermore, at the end of this step, they were also responsible of producing the weekly report for the rest of the team on the work done that week.

In conclusion, the last step was the Monday discussion with the team. An overview of the analysed batch was proposed to the team and a discussion on the most argued bugs was conducted in order to solve the classification. After the building of the dataset, the coding and testing of the algorithm began.

2.1.1 Proposed metrics

In order to measure the goodness of our classification the *Cohen's Kappa* coefficient (in Section 2.1.2) was employed, which measures the degree of accuracy and reliability in a statistical classification. On the other side, precision and accuracy values were exploited to measure the performance of the model generated, as they are usually the main values considered to evaluate a model. In some cases also the ROC&AUC curve was studied to better understand the results obtained.

2.1.2 The *Cohen's Kappa*

To calculate the level of agreement between the two human classifier, also called judges, the *Cohen's kappa* coefficient was used. The Cohen's kappa is a statistical

coefficient that represents the degree of accuracy and reliability in a statistical classification. It measures the agreement between two raters (judges), who each classify items into mutually exclusive categories[3]. This statistic was introduced by Jacob Cohen in the journal Educational and Psychological Measurement in 1960. In formula:

$$k = \frac{p_0 - p_e}{1 - p_e}$$

where p_0 is the relative observed agreement among raters, and p_e is the hypothetical probability of chance agreement.

To interpret Cohen's kappa results these guidelines were followed (Landis and Koch (1977)):

- 0.01 - 0.20 slight agreement
- 0.21 - 0.40 fair agreement
- 0.41 - 0.60 moderate agreement
- 0.61 - 0.80 substantial agreement
- 0.81 - 1.00 almost perfect or perfect agreement

kappa is always less than or equal to 1. A value of 1 implies perfect agreement and values less than 1 imply less than perfect agreement. It is possible that kappa is negative, this means that the two observers agreed less than would be expected just by chance.

In our analysis was clear how more and more the two judges discussed the classification and higher the coefficient got. In more details, after the first batch discussion, the coefficient was 0.1 showing only a slight agreement, while after the second iteration the value was 0.23, i.e., fair agreement. In the third iteration the value raised up to 0.33 meaning the agreement was almost moderate, but in the last iteration it dropped to 0.29. The value of k was probably this low because the judges had never worked together and never had to deal with labeling bugs from BugZilla, making them new to the problem.

[3]I Do Statistics

Table 2.1 contains all the k results obtained in the different iterations on the different batches analysed.

Iteration	Total N. Bugs	A[4]	B[5]	C[6]	D[7]	K	% of Agreement
1	50	26	12	15	14	0.09	56.72%
2	96	56	26	24	23	0.23	63.57%
3	143	82	44	31	29	0.33	67.74%
4	243	130	62	52	45	0.29	66.44%

Table 2.1: Cohen's Kappa results
[4]Both judges agree to include
[5]Both judges agree to exclude
[6]Only first judge wants to include
[7]Only second judge wants to include

Another measure considered was the percentage of agreement, i.e., the value of p_0 in the *Coehn's kappa* equation. This value was always above 60% after the first iteration.

2.2 Taxonomy

In this thesis, as mentioned earlier, the main focus was to obtain the best possible accuracy when labeling a bug, achieving a reduction in costs and time in the *bug triaging* process.

The starting point of the analysis was the work of Fabio and Gemma (2019), who defined the bug's taxonomy in full details in their paper.

2.2.1 *Categories and Sub-Categories*

In this section the taxonomy employed in this thesis will be presented in full details:

- **API:** this category regards bugs concerned with building configuration files. Most of them are related to problems caused by (i) external libraries that should be updated or fixed and (ii) wrong directory or file paths in XML or manifest artifacts.

  Subcategories:

  - **Add-on or Plug-in Incompatibility:** the program does not work correctly for a major add-on/plug-in or many add-ons/plug-ins due to incompatible APIs or libraries, or a functionality, which was removed on purpose, but is still used in the wild.
  - **Incompatibility:** refers to generic errors that do not belong to any of the other categories.
  - **Permission/Deprecation:** bugs in this category are related to two main causes: on the one hand, they are due to the presence, modification, or removal of deprecated method calls or APIs; on the other hand, problems related to unused API permissions are included.
  - **Web Incompatibility:** here the program does not work correctly for a major website or many websites due to incompatible APIs or libraries, or a functionality, which was removed on purpose, but is still used in the wild.

- **Database-related:** collects bugs that report problems with the connection between the main application and a database. For example, this type of bug report describes issues related to failed queries or connection.

- **Development:** are issues related to errors made during the development, it can be due to the addition of a new feature or a change in the code that causes the breakage of a test case or a failure in the built.

  Subcategories:

  - **Compile:** compiling errors.
  - **Test Code:** is concerned with bugs appearing in test code. They are usually related to problems due to (i) running, fixing, or updating test cases, (ii) intermittent tests, and (iii) the inability of a test to find delocalized bugs.

- **GUI-related:** refers to the possible bugs occurring within the Graphical User Interface (GUI) of a software project. It includes issues referring to (i) stylistic errors, i.e., screen layouts, elements colors and padding, text box appearance, and buttons, as well as (ii) unexpected failures appearing to the users in form of unusual error messages.

- **Network Usage:** this category is related to bugs having connection or server issues, due to network problems, unexpected server shutdowns, or communication protocols that are not properly used within the source code.

- **Performance:** collects bugs that report performance issues, including memory overuse, energy leaks, and methods causing endless loops. It can also refer to correct functionalities that have a slow response or are delayed.

- **Program Anomaly:** this are bugs introduced by developers when enhancing existing source code, and that are concerned with specific circumstances such as exceptions, problems with return values, and unexpected crashes due to issues in the logic (rather than, e.g., the GUI) of the program. It is important to note that bugs due to wrong SQL statements do not belong to this category but are classified as database-related issues because they conceptually relate to issues in the communications between the application and an external database, rather than characterizing issues arising within the application. It is also worth noting that in these bug reports developers tend to include entire portions of source code, so that the discussion around a possible fix can be accelerated.

  Subcategories:

  - **Crash:** program unexpectedly stops running.
  - **Hang:** program keeps running but without response.
  - **Incorrect Rendering:** components or video cannot be correctly rendered.
  - **Wrong Functionality:** incorrect functionalities besides rendering issues.

- **Security:** vulnerability and other security-related problems are included in this category. These types of bugs usually refer to reload certain parameters and removal of unused permissions that might decrease the overall reliability of the system. They signal that there is one or more vulnerabilities in the code.

- **Other:** other reasons, e.g., data corruption and license incompatibility.

2.3 Building the Dataset

After learning the taxonomy, the second phase of the work was the building of the dataset. The dataset was built starting from resolved[8] bugs, taken from the *BugZilla* database. BugZilla contains all the bugs ever filed, solved and unresolved, related to Mozilla products.

2.3.1 One Single Bug

The entire job of building the dataset is based on the classification of the single bug. As mentioned in the previous chapter and as the Figure 1.5 depicts, a *bug report* has multiple information that can be used to label the bug.

Starting from the title, the keywords and the flags a preliminary classification can be performed. Moreover, some classes (from Section 2.2), like Security and Performance and Program Anomaly (Crash and Hang) are immediately classified thanks to the corresponding flags and keywords in their reports.

Furthermore, the comments section provides the entire discussion between the developers assigned to the bug. Thanks to their insights, in many cases, the nature of the bug can be discovered. For instance, in many cases one of the firsts comments is a description of the bug behaviour, a piece of code or the log of the error.

As last resource, at the end of the comments section, the last comment is the one that closes the report and it contains the links to the code that removed the bug. Sometimes the code can be the key to label the bug to the correct class, because it shows the piece of code that was changed to remove it.

In some cases labeling a bug is impossible for different reasons. There can be lack of information, both in the flags and in the comments sections, to assign a bug to a specific class, even the title in some situations can be misleading. In some other cases the bug report is used to signal the lack of a feature or a small change to be done that did not causes any problem in the program, these are not bugs and were not classified and not considered in the analysis.

[8] A resolved bug is a bug that has been already labeled, assigned to the correct developers and removed

2.3.2 Batch Classification

The batch classification phase was done in four separate moments, each one was focused on a different batch of bugs to be classified. Moreover, all the four moments had two tasks and were performed by two person:

1. each person separately worked on and labeled the bug
2. the two persons discussed together the best label (or labels) for the considered bug
3. step 1 and 2 were repeated for each bug in the batch

The batches had respectively 50, 50, 50 and 100 bugs, for a total of 250 bugs.

As showed before, for the analysis, two sets of labels were used to classify all the bugs, a generic and a specific one. All the bugs were first labeled using the generic classes and then the specific ones were tried to be applied. In all the cases in which it was not possible to assign a sub-category the generic label was used instead. This indeed means that the specific labels are an extension of the generic ones.

Furthermore, more than one class can describe a bug in some cases and in those occurrences the bug was assigned to all the classes, with a maximum of three, making the problem a multi-label classification problem.

2.3.3 The Dataset

After classifying all the batches, removing the duplicate bugs and the one that were not bugs [9], the dataset contained 188 bugs. Since 188 are not enough input records for a good analysis, the bugs classified by Fabio and Gemma were added to the dataset, reaching a total of 526 bugs.

Table 2.2 contains the dataset composition regarding the general labels, while Table 2.3 shows the composition with the specific label applied. As its clear by the numbers some of the bugs were only classified with a general label, as already said before, this is due to the lack of information to select a specific sub-category label for the given bug.

[9]Some bug report are opened like they are bug but at the end they are new feature request or some other changes, not actual errors in the code

Label	Number of Bugs
API	82
Development	77
GUI-related	109
Network Usage	22
Performance	15
Program Anomaly	240
Security	13

Table 2.2: Bugs classification with general labels

Label	Number of Bugs
Add-on or plug-in incompatibility	38
API	13
Compile	19
Crash	64
GUI-related	109
Hang	6
Incompatibility	25
Incorrect Rendering	12
Network Usage	22
Performance	15
Permission/Deprecation	4
Program Anomaly	80
Test Code	58
Security	13
Web Incompatibility	2
Wrong Functionality	78

Table 2.3: Bugs classification with specific labels

Moreover, the entire dataset will be used for training and validation of the model. In more details it will be divided in two sets: a training set of 473 records and a test set of 53, which means the former is 90% of the dataset while the latter is the remaining 10%. Of course the generation of the two sets is performed randomly[10].

[10]A random state was fixed in order to have the same items in the training and test sets on different runs of the algorithm

Furthermore, the classes population is highly unbalanced, going from 240 bugs for *Program Anomaly* down to 13 for *Security*, considering the general labels. This is indeed also clear if we focus on the sub-categories. This kind of behaviour can lead to a wrong analyses and can result in a wrong model, i.e., a model with a low accuracy on the test set.

Chapter 3

Experiment Design

3.1 The Model

The machine learning models chosen in this project will be presented here, together with the related advantages, disadvantages and the main part regards the two configurations used. The aim was to train a model on labeled bugs, which are bug already resolved, classified by two humans (as described in Section 2.3), and then to use that model to classify untriaged[1] bugs.

The presented work started, as previously mentioned, with the Bugbug (code available at the BugBug GitHub repository[2]) project analysis. Firstly the main focus was on the comprehension of the structure used and secondly on the enhancement of such structure to achieve the goal of this thesis. As starting point the *bugtype* model was considered (again its code is available on GitHub[3]).

3.1.1 BugBug

BugBug was developed to help different activities and tasks, like bug and quality management, and other software engineering tasks using different machine learning techniques. All BugBug knowledge comes from BugZilla[4], because the data present

[1]Bugs that have to be labeled

[2]Bugbug

[3]Bugtype

[4]BugZilla

in that database is used to track everything related to any filed bug, from the feature request of hardware or software malfunction, passing through different bug categories.

Being a huge project, Bugbug contains many different components. It counts as many as 18 different classifiers, each with its own purposes. The classifiers are:

- **Assignee:** it suggests the appropriate assignee for a bug
- **Backout:** detects patches that might be more likely to be backed-out (because of build or test failures). It could be used for test prioritization/scheduling purposes
- **Bugtype:** classifies bugs according to their type
- **Component:** it assigns product/component to untriaged bugs
- **Defect vs Enhancement vs Task:** it extends the defect classifier in order to detect differences also between feature requests and development tasks
- **Defect:** this classifier distinguishes between bugs that are actually bugs and bugs that aren't[5]
- **Devdocneeded:** the aim of this classifier is to detect bugs which should be documented for developers
- **Duplicate:** it finds duplicate bugs
- **QAneeded:** the aim of this classifier is to detect bugs that would need QA verification
- **Regression vs non-Regression:** this classifier detect bugs that are regressions, because the regression keyword in BugZilla is not used consistently
- **Regressionrange:** the aim of this classifier is to reveal regression bugs that have a regression range vs those that do not

[5]Bugs on BugZilla aren't always bugs. Sometimes they are feature requests, refactoring, and so on

- **Regressor:** detects patches which are more likely to cause regressions. It could be used to make riskier patches undergo more scrutiny
- **Spam:** finds bugs which are spam
- **Stepstoreproduce:** reveals bugs that have steps to reproduce vs those that do not.
- **Testfailure:** the aim of this classifier is to detect patches that might be more likely to cause test failures.
- **Testselect:** selects relevant tests to run for a given patch.
- **Tracking:** detects bugs to track.
- **Uplift:** the aim of this classifier is to detect bugs for which uplift should be approved and bugs for which uplift should not be approved.

Bug Type

As mentioned before, the work presented here uses the *bugtype* model as starting point.

This model derives from the *BugModel* (code on GitHub[6]) which itself inherits from the *Model* (code on GitHub[7]). The *Model* contains all the main characteristics of the machine learning model required for our work. It contains indeed a text vectorizer, important because the dataset's records have mostly textual features, and it also contains variables and functions to handle the database, from the download to the reading of the bug reports. Furthermore, it contains functions to spot and extract the most important features for the model. The *BugModel* limits itself to add a database and a function to generate items from the database.

The *BugType* model takes everything from the two aforementioned models and adds components more consistent with its purpose, i.e., label bug according to their type. A dictionary was used for the labels, each key is a specific class and maps to the generic one. Only four classes are present here: *Crash*, *Memory*,

[6]BugModel

[7]Model

Performance and *Security*, and each of them has sub-categories, presented in Table 3.1. Furthermore, the model extracts 16 features out of all the available ones, performs some transformations on them and it applies a *OneVsRest* created from the XGBoost classifier to produce the model. The produced model always outputs the general labels, never the specific ones.

Label	Sub-categories
Crash	crash crashreportid
Memory	memory-footprint memory-leak
Performance	perf
Security	sec-critical sec-high sec-moderate sec-low sec-other sec-audit sec-vector sec-want

Table 3.1: *BugType* labels

Bug Type Classification

Starting from the *BugType* some modification were made in order to obtain the model of this thesis.

Firstly the dictionary used in BugType was substituted by two lists of labels, one containing the general classes names and the other the sub-categories names, eventually they are concatenated when the algorithm is run with selected the configuration that considers the specific labels. Moreover, the same set of attributes are selected here as in the BugType model and the same transformations are applied.

On the other hand, two other big differences were added. Initially, the possibility to select the classifier between a OneVsRest and a Binary classifier was implemented, and afterwards the choice of using a balancing technique for the dataset was added in order to try to remove the unbalanceness of the dataset.

In addition to these modifications also variations in the function to get the labels were made, but were minor changes due to the different format of the records in the input dataset.

Another important issue to remember is that two sets of labels were employed in the classification of the bugs, a generic and a specific one, as already discussed in Chapter 2. Moreover, one can argue that the problem is a multi-label classification one, since a bug can be assigned to more than one class. This was a major difference from the *BugType* code, which only outputs one label per bug.

3.1.2 XGBoost

XGBoost is an optimised distributed gradient boosting library built to be highly efficient, flexible and portable. This library defines machine learning algorithm using the Gradient Boosting framework.

Gradient Boosting is a machine learning technique for regression and classification problem. In order to create a strong classifier, it creates a model employing an ensemble of weak prediction model, decision tress in most of the cases. The purpose of an ensemble model is to use learners that make different kinds of errors, they indeed do not have to be highly accurate. The strength of the ensemble model is given by the combination of the weak learners outputs.

The different models in the ensemble are defined using the same dataset but each record has a weight. At each iteration a new model is trained and the training examples are re-weighted to focus the system on the samples that the most recently learned classifier got wrong. At the end, the classification is based on weighted voting of the weak classifiers, where the weight of a classifier is inversely proportional to its error rate. In conclusion, XGBoost provides parallel tree boosting.

3.2 OneVsRest Model

As mentioned earlier the model is based on the *bug type* classifier with the changes described before. As explained in the previous chapter, the taxonomy in Section 2.2 was used to label the records in the dataset (described in Section 2.3).

OneVsRest[8] was chosen as one possible classifier for the model. It performs a multiple class classification, as described before. The classifier works with 8 classes in the generic configuration while it handles 19 classes for the specific configuration, i.e., 11 labels are for the sub-categories. Since each bug can have more than one label the model will also have multi-label outputs, with the only constraint of maximum three labels.

Three steps define the classifier behaviour. First, one classifier is fitted over a single class, then in the second step all the classifiers are taken individually and the class is fitted against all the other classes. As last step more than one label for each bug is produced by each classifier. This classification is made using a two dimensional matrix that is filled using the following formula: given a generic cell of the matrix, $[i, j]$, where i is the sample and j is the label the value of the cell is defined as:

$$\begin{cases} \text{if i has j,} & 1 \\ \text{otherwise,} & 0 \end{cases}$$

One of the main advantages of this classifier is its interpretability. Since each class is represented by one and one classifier only, it is possible to gain knowledge about the class by inspecting its corresponding classifier. This advantage has been exploited during tests to better understand the model produced by the algorithm.

As mentioned above, the OneVsRest algorithm fits one classifier per class as first step. The aforementioned classifier has to be chosen at the creation of the object in the code, passing it as argument in the constructor. In this case the estimator selected was the implementation of the *scikit-learn* API for XGBoost classification[9].

3.3 Binary Model

As a conclusion of the testing (see Section 4.1) of the OneVsRest classifier was clear that the accuracy was far from optimal, a value of 30% accuracy cannot be employed in real operations. For this reason the choice of a new classifier was added to the algorithm implementation. In more details, a *mode* variable was included

[8]OneVsRest Classifier

[9]XGBoost xgbclassifier documentation

to select between the OneVsRest classifier depicted in the previous section and a Binary classifier, which analyses a single class at a time.

In machine learning a binary classifier is an estimator that assigns a record to one of two possible classes by studying the attributes of the record. In this project, the two classes are dynamically created based on the choices of the programmer, the positive class is passed as an additional argument and all the samples belonging to that class in the dataset will be labeled as 1, while all the other samples as 0.

In this thesis implementation the design of this second model was created trying to keep it as much as possible homogeneous with the already presented model. Following such idea, this *mode* only needs an additional argument passed as parameter from the command line: the name of the class under consideration, in addition to the argument that specify the choice of the use of the binary classifier. The algorithm takes care of the rest by assigning label 1 to the specified class and label 0 to all the other classes. In conclusion, it outputs a model able to classify a new bug as belonging to the specified class or not.

Also here the XGBoost (explained in Section 3.1.2) library was used. The available software here is employed again as the base for the model, producing the class assignment after the prediction of the label of the bug under analysis. In this classifier the same estimator used in the OneVsRest case was used to produce the binary classification.

3.4 Modifications

After testing the models with this configurations, the results (presented in Chapter 4) obtained were not the expected ones, especially for the OneVsRest case. This indeed required some changes to try to obtain a better accuracy. For this operation different approaches and changes were considered separately or combined.

As first thought, the main modification was in the dataset by removing all the bugs that were not from BugZilla[10] in order to have homogeneity in the dataset. 284 bugs were removed from the set, leaving the dataset with 243 bugs.

[10]Some of the bugs classified by Fabio and Gemma were from Eclipse

One other concern was on the XGBoost library, possibly a wrong choice for the project. For this reason the classifier was changed using only the *scikit-learn* choice, without the XGBoost implementation. The result of this operation was awful with the initial dataset, showing the need of the library in this project's tool, but after the modifications on the dataset the XGBoost library was removed due to the higher results of different algorithms.

Another different approach on the classifier was the employment of different types of learners passed to the OneVsRest constructor instead of the XGBoost learner. Different classifier were used, going from KNN to Linear SVM, passing through Naïve Bayes. The greatest accuracy was reached by SVM, presented with all the other output results in the next Chapter.

Furthermore, since one hypothesis of the bad results obtained was that the dataset was unbalanced and not big enough, more bugs were classified. The bugs added belonged to the classes *crash*, *performance* and *security* because they are easily classified thanks to the flag in the bug reports. Roughly 100 bugs were added for each category and at the end of this step the dataset contained 546 bugs. Again, all the new outcomes are presented in the next Chapter.

3.4.1 Minimum matching threshold

In machine learning, there is a model called Logistic Regression[11] that is used to compute the probability that a certain instance can be classified as 0 or 1. Here this estimator has been applied to predict the appearance of an instance to a specific class.

The logistic regression is used to get the probability inherent to a certain prediction. This value can be read as a simple probability that the event occurs, which in this thesis means that the bug under analysis belongs to a class, while in other cases the value is converted to a binary one. For example, a model that returns a prediction of 0.001, for a bug to be labeled as class A, is predicting that it is very unlikely that the bug can be categorised to class A, on the other hand a value of 0.999 means that there is a high chance that the bug stands in class A.

[11]Logistic Regression

Arguably the main threat is distinguishing middle cases. A classification threshold has to be picked to have a mapping between a regression value and a binary identifier. Moreover, this value indicates the point above which is possible to identify a record as belonging to a specific class with a certain margin of confidence. This threshold is indeed necessary and problem dependent, showing why its value is not set statically at 0.5.

The main focus was to find the minimum threshold value that maximises the confidence over the classification model.

Chapter 4

Results

The tool presented in this thesis has been developed to label bugs in a fast and, eventually, error free way, allowing the developers to only focus on the second part of the job, which is the bug removal task. However the accuracy reached in our experiments showed possible errors in our first assumptions or in the initial models, especially in the OneVsRest one, and this led to modifications that brought more consistent outcomes.

4.1 Tests

In our work, all the experiments with the classifiers were performed using the same configuration of the algorithm. In more details, many models were produced in each main experiment changing only the classifier part, leaving all the other components the same. Different configurations were employed in order to find the one with the highest accuracy.

Initially, different changes were applied in the code, in the feature selector or in the transformations used. In particular one possible change was to separate the first comment from the rest of the comments, as two separate features, giving more importance to the first comment, and, on other tests, they were left together. Moreover, since the classes distribution was unbalanced, a balancing technique was added as an additional transformation in some experiments.

As mentioned earlier, all these experiments were led using the same datasets[1] that was split in train and test set, with a ratio 90% and 10%. As it is showed in Section 4.1.3 other changes were made after the first experiments to try to solve the possible errors present in the algorithm components.

4.1.1 *OneVsRest* Classifier results

Here all the initial results obtained using the *OneVsRest* classifier (described in Section 3.2) are presented.

The accuracy of this classifier was very low at the beginning and indeed it was always below 30% in the first experiments. Furthermore, considering the first comment as separated from the other didn't provide much difference than considering all the comments together. In both cases the accuracy was around 28%.

In order to tune some parameters, like *min_ diff*, and check if it was useful to have the first comment separate from the other, the algorithm was tested with the values in Table 4.1. The Table shows that separating the first comment does not change much the accuracy, which is influenced only by the value of *min_ diff* in this situation. For this reason the comments were always considered all together and the value of *min_ diff* was set to 0.001 in all the other tests.

First Comment merged with Comments	*min_ diff*	Accuracy
Yes	0.01	27%
Yes	0.001	28%
Yes	0.0001	28%
No	0.01	27%
No	0.001	26%
No	0.0001	26%

Table 4.1: Results obtained when tuning *min_ diff* and the handling of the Comments

Moreover, using a balancing technique did not help either. More specifically the model reached an accuracy of 30.43%. This increase in the accuracy can be related

[1]Three sets were used with respectively 526, 243 and 546 bugs

to the fact that now the dataset is more balanced, lowering the probable overfitting that was present in the previous scenarios.

These results can be due to two main reasons: the samples in the dataset, or the algorithm definition. In the former case the problem can be due to the lack of samples for some classes, making the dataset unbalanced, leading to a wrong analysis and model. This unbalanceness creates a difficulty for the model, which is not able to learn enough information from the too few samples provided by the dataset.

On the other hand, also the algorithm can have some blame, since there can be errors in it, particularly in the feature extractor. Some features can be indeed useless or they can make classes too much similar to each other. For this reason the model will not be able to distinguish between some of the classes, lowering its accuracy with mis-classifications. This problem, even if all the correct features are selected, can still be present since it can be due to the fact that different problems are described in similar ways, making them alike to the classifier's "eyes". Two or more classes can indeed be really similar to each other for the classifier point of view, leading to wrong labeling in some cases.

As already mentioned, one modification of this code was to remove the XGBoost estimator, without obtaining meaningful results, probably because it is a state of the art library. On the other hand the modification of the estimator passed and the changes on the dataset brought a different outcomes, showing the highest obtained accuracy of 73.32% (showed in Section 4.1.3).

4.1.2 *Binary* Classifier results

The results of the *binary* classifier (presented in Section 3.3) were more satisfying than the initial outcomes of the OneVsRest. In all the experiments an accuracy above 50% was achieved. In deeper details here we run a binary classifier considering one class against all the other. The class under consideration was labeled as class 1 while all the other were labeled as 0. With this variation in mind the experiments performed were equal to the ones above performed on the OneVsRest classifier.

One experiment was done without using the balancing technique, the results are in Table 4.2, while another experiment was led applying a balancing technique to the

Label	Type	N. of Bugs	Accuracy	Precision	Recall
API	General	82	89%	0.0	0.0
Development	General	73	81%	0.64	0.45
GUI-related	General	109	82%	0.20	0.09
Network Usage	General	22	95%	0.0	0.0
Performance	General	15	99%	0.0	0.0
Program Anomaly	General	240	64%	0.68	0.78
Security	General	13	97%	0.0	0.0
Add-on or Plug-in Incomp.	Specific	38	99%	0.0	0.0
Compile	Specific	19	97%	0.0	0.0
Crash	Specific	64	94%	0.87	0.79
Hang	Specific	6	98%	0.0	0.0
Incompatibility	Specific	25	99%	0.0	0.0
Incorrect Rendering	Specific	12	95%	0.0	0.0
Permission/Deprecation	Specific	4	100%	0.0	0.0
Test code	Specific	58	84%	0.70	0.48
Web Incompatibility	Specific	2	99%	0.0	0.0
Wrong Functionality	Specific	78	70%	0.52	0.23

Table 4.2: Statistics without using the balancing technique

dataset, the output results are in Table 4.3.

As for the *OneVsRest* case, changing how the comments were handled did not change much if not at all the accuracy obtained by the generated models. On the other hand, the use of a balancing technique here is destructive (see Table 4.3). The accuracy of each class, both general and specific, were all above 90% before the introduction of the balancing, while most of them dropped to 50%, 60% or 70% with it.

The results obtained with this classifier can be related to the fact that for the binary case the dataset can be seen as more balanced, but, still, some accuracy are two low for a binary classifier. The low accuracy can be again related to the fact that some classes can be too alike for the classifier, leading the model to mistakes in labeling the records belonging to those classes, also in the binary mode.

After the changes in the dataset, which means after the removal of the bugs not coming from BugZilla the binary classifier reached an average accuracy of 63%,

Label	Type	N. of Bugs	Accuracy	Precision	Recall
API	General	82	55%	0.20	0.58
Development	General	73	81%	0.40	0.76
GUI-related	General	109	64%	0.33	0.66
Network Usage	General	22	66%	0.07	0.58
Performance	General	15	47%	0.02	0.47
Program Anomaly	General	240	67%	0.63	0.65
Security	General	13	53%	0.02	0.57
Add-on or Plug-in Incomp.	Specific	38	63%	0.13	0.66
Compile	Specific	19	63%	0.05	0.52
Crash	Specific	64	92%	0.68	0.84
Hang	Specific	6	58%	0.02	0.40
Incompatibility	Specific	25	57%	0.07	0.60
Incorrect Rendering	Specific	12	64%	0.04	0.57
Permission/Deprecation	Specific	4	66%	0.0	0.0
Test code	Specific	58	85%	0.44	0.79
Web Incompatibility	Specific	2	99%	0.0	0.0
Wrong Functionality	Specific	78	68%	0.28	0.73

Table 4.3: Statistics using the balancing technique

which is still a low accuracy but now only the bugs coming from BugZilla are considered, i.e., all the bugs have the same kind of information. Table 4.4 shows the new bug label distribution together with the results obtained by the binary classifier. In this tests the balancing was included in the classifier, this mean that the learner took care of adjusting the input dataset in order to have it more balanced.

One other test was run on the binary classifier after the additional modification on the dataset. Using the dataset enhanced with the 300 new bugs, with a total of 546 bugs, the binary classifier reached the accuracy showed in Table 4.5. As it is clear from the results, the enlarging of the dataset boosted the accuracy, especially for the classes for which elements were added.

Label	Type	N. of Bugs	Accuracy	Precision	Recall
API	General	33	55%	0.13	0.45
Development	General	42	80%	0.47	0.85
GUI-related	General	49	59%	0.27	0.72
Network Usage	General	8	61%	0.06	0.60
Performance	General	3	60%	0.03	0.40
Program Anomaly	General	131	72%	0.77	0.71
Security	General	10	70%	0.12	0.80
Add-on or plug-in incompatibility	Specific	11	70%	0.11	0.80
Compile	Specific	5	48%	0.02	0.40
Crash	Specific	34	87%	0.57	0.72
Hang	Specific	3	56%	0.01	0.40
Incompatibility	Specific	6	68%	0.01	0.20
Incorrect Rendering	Specific	9	41%	0.04	0.80
Permission/Deprecation	Specific	1	f	f	f
Test Code	Specific	37	83%	0.52	0.81
Web Incompatibility	Specific	2	42%	0.01	0.20
Wrong Functionality	Specific	60	66%	0.41	0.69

Table 4.4: Bugs label distribution and results on the cleaned dataset

Label	Type	N. of Bugs	Accuracy	Precision	Recall
API	General	33	65%	0.65	0.65
Development	General	42	76%	0.76	0.76
GUI-related	General	49	81%	0.81	0.82
Network Usage	General	8	50%	0.50	0.50
Performance	General	104	99%	0.99	0.99
Program Anomaly	General	232	72%	0.85	0.85
Security	General	111	96%	0.97	0.96
Add-on or plug-in incompatibility	Specific	11	90%	0.92	0.91
Compile	Specific	5	40%	0.38	0.40
Crash	Specific	135	96%	0.96	0.96
Hang	Specific	3	75%	0.83	0.75
Incompatibility	Specific	6	70%	0.70	0.70
Incorrect Rendering	Specific	9	50%	0.50	0.50
Permission/Deprecation	Specific	1	f	f	f
Test Code	Specific	37	85%	0.85	0.85
Web Incompatibility	Specific	2	50%	0.25	0.50
Wrong Functionality	Specific	60	71%	0.72	0.71

Table 4.5: Binary results on the changed dataset

4.1.3 Modifications

In this section the results obtained by the smaller and different trials are presented. The algorithm was slightly changed in order to run different scenarios to find meaningful information and possible solutions to the low accuracy obtained in the other cases.

As mentioned in Chapter 3, the major changes were done in the selection of the classifier, since the change in the dataset has already been presented. The classifier used for this tests were Naïve Bayes, K Nearest Neighbours and Support Vector Machines, all passed to the OneVsRest constructor.

All this experiments were run with three different datasets: the initial one with 526 bugs, the one with 243 bugs after the removal of the bugs not from BugZilla and the one with 546 bugs after the addition of the new bugs.

Also the first classifier, i.e., the OneVsRest with the XGBoost estimator passed to the constructor, was tested with the new datasets and obtained: 42% accuracy with the 243 bugs dataset and 68% with the 546 bugs one.

Naïve Bayes

The first classifier used was the Naïve Bayes. This classifier is usually used as benchmark in the analysis and it is based on the Bayes Theorem:

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

The accuracy was probably low for this algorithm because the Naïve Bayes theorem makes the assumption that the features of the input records are uncorrelated to each other. In this case there is an high chance that some of the attributes are somehow related. The best obtained accuracy was 66%. (results in Table 4.6)

Dataset	Accuracy	Precision
526 bugs	28.7%	
243 bugs	44.5%	0.40
546 bugs	65.98%	0.66

Table 4.6: Results Naïve Bayes

K Nearest Neighbours

On the other hand, the KNN classifier uses the entire dataset to label a new instance, instead of collecting information from the training set and tune some parameters accordingly. KNN computes the distance between the new instance and all the records in the set and uses the K nearest samples to label the new input record. Both the value of K and the measure of the distance have to be carefully selected. Here a value of $K = 19$ and the *Manhattan distance* were used. The best accuracy obtained was 60%. This model reached usually the lowest accuracy in all the experiments. (results in Table 4.7)

Dataset	Accuracy	Precision
526 bugs	23.6%	
243 bugs	35.8%	0.27
546 bugs	59.5%	0.59

Table 4.7: Results K Nearest Neighbours

Support Vector Machines

SVM is a linear model for classification and regression problems. The idea of this algorithm is to find the line or hyperplane which best separates the classes. To find the best separating line or hyperplane, SVM first finds the points closest to the line or hyperplane for two classes, these points are called support vectors. Now the distance of the support vectors and the line or hyperplane is computed. The best line or the hyperplane is the one for which the distance, also called margin, is maximised. The process is repeated for every couple of different classes. In formula:

$$\max_{w,b} \frac{1}{||w||} \quad \text{subject} \quad \text{to} \quad y_i[< x_i, w > +b] \geq 1$$

where w is the vector perpendicular to the hyperplane or line, b is the bias term, $||w||$ is the norm of w and the equation $y_i[< x_i, w > +b] \geq 1$ represents the fact that the SVM has to correctly classify every point x_i with label y_i. The *scikit-learn* LinearSVC[2] algorithm was employed here. (results in Table 4.8)

[2]LinearSVC

Dataset	Accuracy	Precision
526 bugs	35.95%	
243 bugs	46%	
546 bugs	73.32%	0.74

Table 4.8: Results Support Vector Machines

These tests were run only using the general labels, since they are less a better accuracy is expected. Running the best algorithm, LinearSVC[3], on the entire set of labels, considering also the specific ones, indeed generated an accuracy of 28% with the dataset of 243 bugs.

4.2 Outcomes

Here a summary of the best obtained outcomes is presented in Table 4.9.

Algorithm	Accuracy
Naïve Bayes	65.98%
KNN	59.5%
Binary	75-85% (On Average)
XGBoost	68%
LinearSVC	73.32%

Table 4.9: Results

In conclusion, the best model obtained was the Linear SVM one. The classifier reached an accuracy of almost 74%.

[3]LinearSVC *scikit-learn*

Chapter 5

Conclusions

Bug triaging is a very important, long and difficult step in the bug removal process, especially in companies with multiple products receiving thousands of bug reports every week. Thankfully the software components are the same for a long period of time, rarely they change, and this means that a good model can be employed for years. Moreover, it can be easily updated using a new and more comprehensive dataset.

This thesis proposes a tool that would avoid useless work for the developers, who will have more time to focus on the removal part of the process. Our project is based and works with bug related to all the *Mozilla* products but with few changes it could be extended to work also with bugs coming from origins different than the *BugZilla* database.

Our model takes an *untriaged* bug and returns a label for it. Depending on the case the tool can use the *OneVsRest* (see Section 3.2) or the *Binary* (see Section 3.3) classifier. It goes without saying that the binary classifier takes more time than the others since it has to be run multiple times, i.e., one time for each class. Furthermore, the binary classifier result requires further computation, like comparing all the results in order to provide a label.

The model is based on the bugs classified by the human classifier, which are bugs that were randomly picked from the database. This operation may result in a biased model, since the classes are unbalanced and some classes in the taxonomy are never classified, like database-related bugs. As showed in Chapter 3, the unbalanceness was partially resolved after adding some other bug in the dataset and by using a

balancing technique, but some unbalanceness still persist and can be seen in the accuracy results (showed in Chapter 4).

5.1 Summary of the obtained results

As mentioned in Chapter 4, two were the main configurations of the experiments run in this thesis. However they were run multiple times to tune the main hyper-parameters of the model, especially *min_ diff* of the *Column Transformer* (see Table 4.1). All the test cases were run with the same training and test sets, selecting the model configuration using the command line arguments passed to the algorithm.

As discussed in Chapter 4, the initial results obtained for the *OneVsRest* model can be related either to the dataset or the algorithm. In the former case, the heterogeneity and unbalancedness of the data used could have led to a model with pour classification accuracy. This causes a problem in all the classifiers, since all the different learners do not have enough information on some classes, i.e., the one with too few elements. For this reason modifications of the dataset were implemented, leading to 243 bugs and later 546, all coming from the same database, BugZilla.

On the other hand the performance of the same records in the *Binary* experiment shows that the data have a good quality, even if also here some classes do not have an optimal accuracy. This can actually be related to the fact that some classes may be too similar to each other, creating problems in the classifiers that try to learn the different characteristics to distinguish between them.

This indeed leads to the ladder case, which is to consider that the poor accuracy of the *OneVsRest* model is caused by the algorithm. Since the chosen classifier is a state of the art classifier, coming from the XGBoost library, the problem can be in the features selected or in their handling. The feature selected could be not the perfect subset for the problem at hand. As already said the difficulty of the learner to distinguish between classes can be either due to the fact that in reality some of the classes are too close or because the feature extracted from the reports are the wrong ones to make the classifier strong enough.

In conclusion, after the changes in the dataset, the best classifier turned out to be the LinearSVC one. In more details, employing the dataset with 546 bugs, all

coming from BugZilla, with the LinearSVC estimator passed to the OneVsRest's constructor produced the best possible classifier with an accuracy of almost 74%. This can probably due to the fact that Support Vector Machines only take advantage of the samples near the hyperplane, i.e., the support vectors. In this way, regardless of the position of the other samples the model depends only on those records. Moreover, even if some elements, belonging to two different classes, are close to each other they will still be separated by the hyperplane making the model capable to distinguish between the two different classes, making less mistakes.

All future works have to start by dealing with the problems still present and explained above before any kind of enhancement or additional component can be brought to the algorithm.

5.2 Future works

Here a list of a few improvement point is presented. These can be aimed towards increasing the accuracy of the models or to enhance its capabilities.

- One obvious work could be the enlarging of the dataset with many other records, especially samples belonging to the less represented classes, i.e., the classes with fewer elements. A number as 1000 bugs could be the minimum to provide a better model, regardless the algorithm used;
- Another possible work could be the research of the best feature subset to extract. Since an accuracy of 74% is good but not perfect a good starting point could be the analysis of the extracted features;
- Some bugs are generated when resolving other bugs, they are called regression bug. With the right additional samples the tool can be updated to understand if a bug is a regression bug;
- Make the algorithm capable to understand if a bug report is of a bug or simply a feature request or not a bug could be another possible additional feature;
- Extending the tool to be able to handle bug reports coming from different sources. For now it is able to work only with reports coming from BugZilla, it would be an enhancement if it was able to label also bugs from other databases;

- A possible extension of the algorithm could be the identification of bugs that are duplicate. These duplicates are eventually found by the triage process, but finding duplicate bugs as quickly as possible provides more information for developers trying to diagnose a crash;
- Enhancing the algorithm to be able to understand the importance of a bug, to signal its priority for a given Firefox release would be an even further step;

References

Akila, V., Zayaraz, G., & Govindasamy, V. (2015). Effective bug triage–a framework. *Procedia Computer Science*, *48*, 114–120.

Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., & Guéhénec, Y.-G. (2008). Is it a bug or an enhancement?: a text-based approach to classify change requests. *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, *23*.

Catolino, G., Palomba, F., Zaidman, A., & Ferrucci, F. (2019). *Not all bugs are the same: Understanding, characterizing, and classifying bug types.* Retrieved from `https://dibt.unimol.it/staff/fpalomba/documents/J22.pdf`

Chillarege, R., Bhandari, I. S., Chaar, J. K., Halliday, M. J., Moebus, D. S., Ray, B. K., & Wong, M.-Y. (2018). Orthogonal defect classification-a concept for in-process measurements. *Applied Soft Computing*, *18*(11), 943–956.

Hernández-González, J., Rodriguez, D., Inza, I., Harrison, R., & Lozano, J. A. (2018). Learning to classify software defects from crowds: a novel approach. *Applied Soft Computing*, *62*, 579–591.

Huang, L., Ng, V., Persing, I., Chen, M., Li, Z., Geng, R., & Tian, J. (2015). Autoodc: Automated generation of orthogonal defect classifications. *Automated Software Engineering*, *22*(1), 3–46.

Javed, M. Y., Mohsin, H., & et al. (2012). An automated approach for software bug classification. *Complex, Intelligent and Software Intensive Systems (CISIS), 2012 Sixth International Conference on. IEEE*, 414–419.

Landis, J., & Koch, G. G. (1977). The measurement of observer agreement for categorical data. *Biometrics*, *33*, 159–174.

Murphy, G., & Cubranic, D. (2004). Automatic bug triage using text categorization. *Proceedings of the International Conference on Software Engineering & Knowledge Engineering (SEKE)*, 92–97.

Nagwani, N., Verma, S., & Mehta, K. K. (2013). Generating taxonomic terms for software bug classification by utilizing topic models based on latent dirichlet allocation. *ICT and Knowledge Engineering (ICT&KE), 2013 11th International Conference on. IEEE*, 1–5.

Thung, F., Le, X.-B. D., & Lo, D. (2015). Active semisupervised defect categorization. *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, *IEEE Press*, 60–70.

Thung, F., Lo, D., & Jiang, L. (2012). Automatic defect categorization. *Reverse Engineering (WCRE), 2012 19th Working Conference on. IEEE*, 205–214.

Xia, X., Lo, D., Wang, X., & Zhou, B. (2014). Automatic defect categorization based on fault triggering conditions. *Engineering of Complex Computer Systems (ICECCS), 2014 19th International Conference on. IEEE*, 39–48.

Zhang, T., Jiang, H., Luo, X., & Chan, A. T. (2016). A literature review of research in bug resolution: Tasks, challenges and future directions. *The Computer Journal 59*, *59*(5), 741-773.

Zhou, Y., Tong, Y., Gu, R., & Gall, H. (2016). Combining text mining and data mining for bug report classification. *Journal of Software: Evolution and Process*, *28*(3), 150–176.