Politecnico di Torino

Master's Degree Course in Computer Engineering

Master's Degree Thesis

Automated analysis and classification for software issue report using machine learning

Supervisor:
Prof. Luca Ardito

Candidate:
Dario Ciaudano

Co-supervisor:
Prof. Maurizio Morisio

Academic year 2019/20
Torino

Abstract

***Context*:** Issue reports are used to store the problems that occur using a software and them are submitted by developers and testers. After the software issue is detected then is redirect to an expert that operates in order to solve the problem. This operation is very time consuming, because in this process of creation of the bug report there is possible to find a wrong classification due to human misjudging.

***Goal*:** In this thesis' work, we build a tool, that using machine learning techniques, to classify in an automatic way the issue report with the most probable label class.

***Method*:** This works is based over the Mozilla bugs stored in Bugzilla, a bug tracker for general purposes, and it is focused to the correct classification of a new bug. The model works over an implemented version based on the Bugbug project and it creates a classifier with labeled bugs, that can be used with two implementation: *OneVsRest* or a *Binary.*

***Results*:** Ours work is testes over two different scenarios: and a single class behaviour, the class examined is considered as the positive class against all the others, and a multiple classes that groups different classes analyzed individually. The total accuracy obtained is higher than 73% with the *OveVsRest* approach, while with the *Binary* analysis, the higher result is higher than 80% obtained with the most prevailing cases, while the other class labels with minor weight obtained an accuracy slightly lower than 50%.

***Conclusions*:** Our tool works successful to fulfil the main goal it was designed for. This work can be further expanded in the future with a more homogeneous data set in its classes. It is also possible to improve slightly the precision of the classifier with some minor changes.

Contents

List of Figures

List of Tables

Chapter 1

Background and related works

1.1 Introduction to bug classification

The bug classification activity is of paramount importance to produce bug-free software. Bug classification process exists, because human produces software, that will never be free of bugs. This is an expensive and long task for the behaviour that correlated to the line of code in a program, in fact the faster those rise the LOC[1] and the more bugs can be founded.

Nowadays, those expensive tasks are solved by experts with different tools correlated to the machine learning and the artificial intelligence, that can reduce the overall time to process and solve a bug classification. The development of an automatic bug classifier would speed up the classification and consequently the bug removal process.

It is possible to define a list of steps to follow in order to characterize and classify a bug classification. Once a bug is reported, developers are requested to analyse the *bug report* and classify the bug. After that, it is assigned to the most skilled developers in the specific area of interested given by the classification. Each classification generally involves more than one person classifying the bug separately and then discuss the classification, this could take days.

As bug classification is mainly focused in assigning the bug removal work to the

[1]The line of code in a program

most qualified developer or team, reaching an high accuracy in the classification becomes the firs and most time-consuming step (Akila et al., 2015), especially since a wrongly classified bug might cause problem during the removal process.

Using a machine learning algorithm, in order to classify the bugs automatically, reduces not only the exploitation of resources and time, but this would limit the possible error correlated to the human behaviour, ie. generally issues in codes are reported in a wrong way from the developers, the tester or the customers. When a bug is misjudged and classified as a different issue category, it would be assigned to a developers that more likely do not have the desired skills needed to resolve the issue and the process can take more time than the expected. Moreover the maintenance of such code would be trivial in most of the cases.

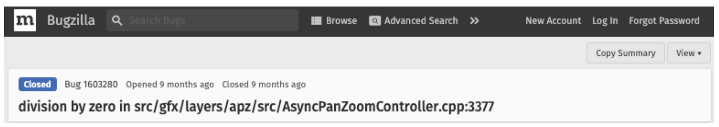
We argue that removing errors made by developers in the process of understanding the bug type can be beneficial in the process. It will properly identify the developer who should be assigned to its debugging, speeding-up the bug analysis and resolution process.
The removal of this type of errors greatly affects the quality of the classifier and therefore the aim of great importance is to achieve an high accuracy with the model. Otherwise, after a wrong classification there are an increase time expense to solve the bug, because we have to reassign the bug to the better developer.

1.1.1 Example of bug classification

In Figures 1.1, 1.2, 1.3 and 1.4 this is a brefly presentation of the different information that it is possible to find in a *bug report* taken from the *BugZilla* database. From this example the main steps of a bug classification can be analysed.

The bug report can be divided in four sections taken in account:

- title of the bug, it can be meaningful for the classification since in many cases synthesizes perfectly the problem at hand

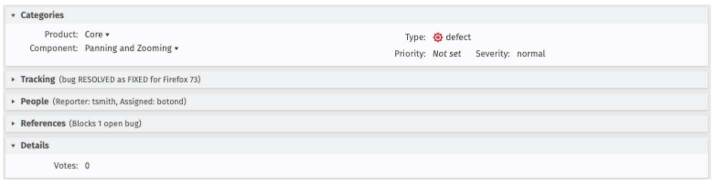


Bug 1603280 Opened 9 months ago Closed 9 months ago

division by zero in src/gfx/layers/apz/src/AsyncPanZoomController.cpp:3377

Figure 1.1: ex of *bug report* title

- the description in tags and labels of the bug: priority and severity, category, the people to which it is assigned, tracking point and useful flags



▾ **Categories**

Product: Core ▾
Component: Panning and Zooming ▾
Type: defect
Priority: *Not set* Severity: normal

▸ **Tracking** (bug RESOLVED as FIXED for Firefox 73)

▸ **People** (Reporter: tsmith, Assigned: botond)

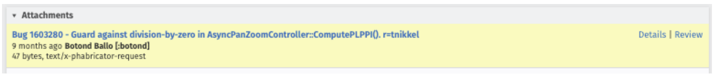
▸ **References** (Blocks 1 open bug)

▾ **Details**

Votes: 0

Figure 1.2: ex of *bug report* information

- the attachments of the bug: in most of the cases those can be pictures, videos or link to the section code of the error presented in the report

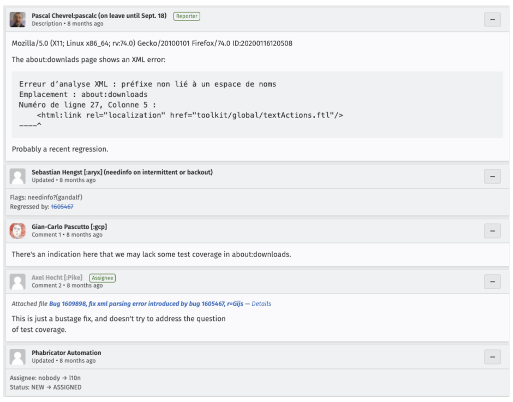


▾ **Attachments**

Bug 1603280 - Guard against division-by-zero in AsyncPanZoomController::ComputePLPPI(). r=tnikkel — Details | Review
9 months ago **Botond Ballo [:botond]**
47 bytes, text/x-phabricator-request

Figure 1.3: ex of *bug report* attachments

- the comments of the bug: in this section the developers assigned to the bug can exchanges ideas an insights on the bug taken in analysis (Figure 1.4)



Pascal Chevrel:pascalc (on leave until Sept. 18) Reporter
Description • 8 months ago

Mozilla/5.0 (X11; Linux x86_64; rv:74.0) Gecko/20100101 Firefox/74.0 ID:20200116120508

The about:downlads page shows an XML error:

```
Erreur d’analyse XML : préfixe non lié à un espace de noms
Emplacement : about:downloads
Numéro de ligne 27, Colonne 5 :
    <html:link rel="localization" href="toolkit/global/textActions.ftl"/>
----^
```

Probably a recent regression.

Sebastian Hengst [:aryx] (needinfo on intermittent or backout)
Updated • 8 months ago

Flags: needinfo?(gandalf)
Regressed by: ~~1605467~~

Gian-Carlo Pascutto [:gcp]
Comment 1 • 8 months ago

There's an indication here that we may lack some test coverage in about:downloads.

Axel Hecht [:Pike] Assignee
Comment 2 • 8 months ago

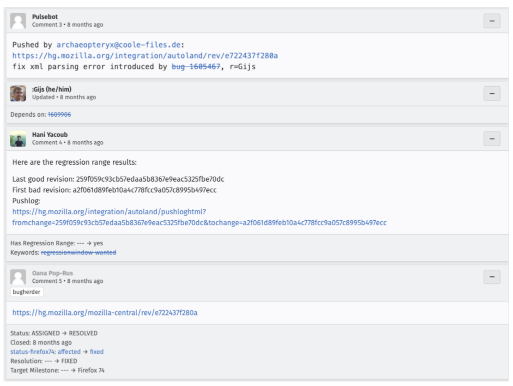
Attached file Bug 1609898, fix xml parsing error introduced by bug 1605467, r=Gijs — Details

This is just a bustage fix, and doesn't try to address the question
of test coverage.

Phabricator Automation
Updated • 8 months ago

Assignee: nobody → l10n
Status: NEW → ASSIGNED

(a) comments



Pulsebot
Comment 3 • 8 months ago

```
Pushed by archaeopteryx@coole-files.de:
https://hg.mozilla.org/integration/autoland/rev/e722437f280a
fix xml parsing error introduced by ~~bug 1605467~~, r=Gijs
```

:Gijs (he/him)
Updated • 8 months ago

Depends on: ~~1609906~~

Hani Yacoub
Comment 4 • 8 months ago

Here are the regression range results:

Last good revision: 259f059c93cb57edaa5b8367e9eac5325fbe70dc
First bad revision: a2f061d89feb10a4c778fcc9a057c8995b497ecc
Pushlog:
https://hg.mozilla.org/integration/autoland/pushloghtml?fromchange=259f059c93cb57edaa5b8367e9eac5325fbe70dc&tochange=a2f061d89feb10a4c778fcc9a057c8995b497ecc

Has Regression Range: --- → yes
Keywords: ~~regressionwindow-wanted~~

Oana Pop-Rus
Comment 5 • 8 months ago
bugherder

https://hg.mozilla.org/mozilla-central/rev/e722437f280a

Status: ASSIGNED → RESOLVED
Closed: 8 months ago
status-firefox74: affected → fixed
Resolution: --- → FIXED
Target Milestone: --- → Firefox 74

(b) comments

Figure 1.4: ex of *bug report* comments

The bug report changes and it can update its priority, severity and developers assigned during the resolving process, ie. the bug was wrongly classified.
In case the bug is wrongly classified the wrong developers will received and they will signal the mistake. This process lead to a new classification and a new assignation of the bug, losing time and resources.

1.2 Steps of bug classification

Classifying a bug is a challenging and time consuming procedure. Here are the steps:

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

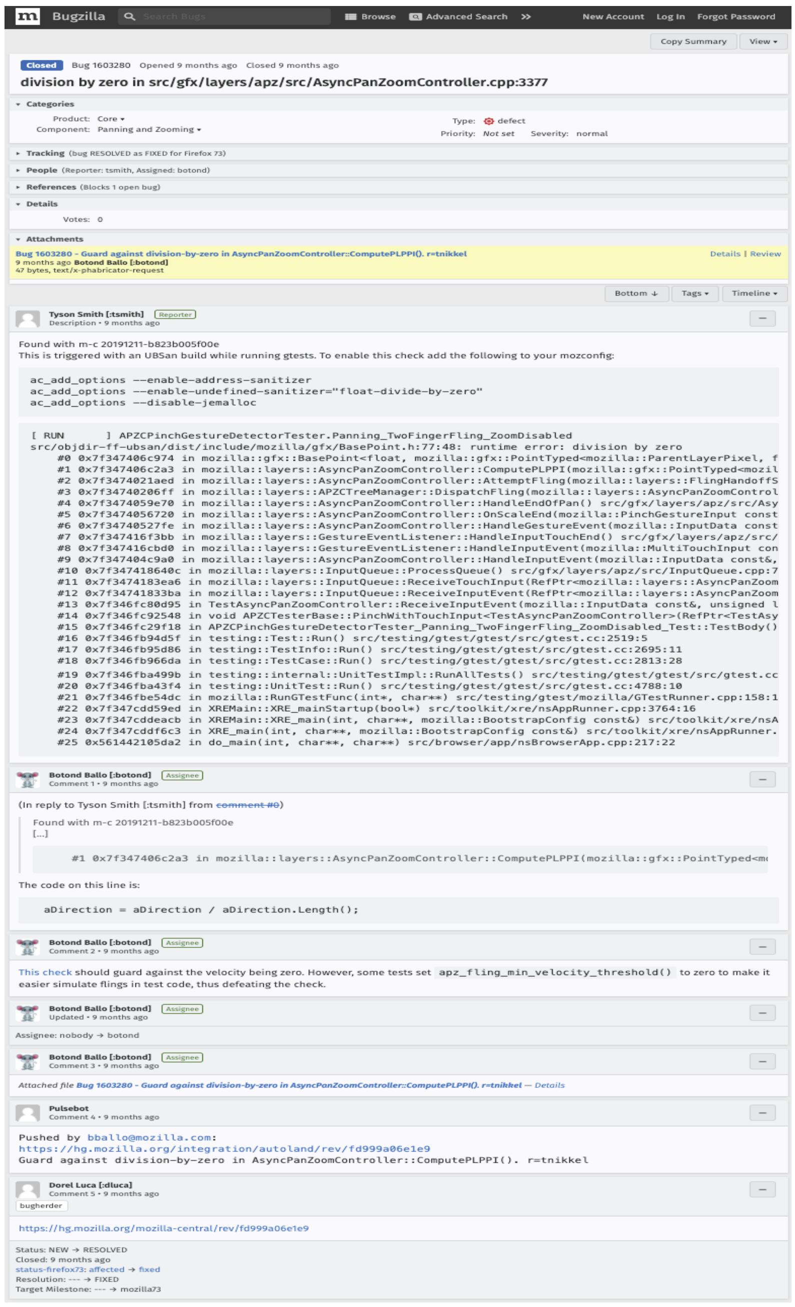
1.2.1 The cost of bug classification

It is possible to give a value to the duration time, that is needed to classify correctly the bug, and the effort amount of developers to achieve the task. The union of those factors is considered the cost of bug classification.
It is important to remind that a bug requires takes longer time to be classified, the more complex the bug is. On the other side dedicate a larger number of people for a specific task, it will increase the cost related to the classification, with a reduction of the process duration. In conclusion automatic bug classification will have a fundamental role in cost reduction.

It is possible to speculate over a generic bug and the cost related to it. The process cost of finding and removing a bug can change, if it is started during the development and test phases, or during the production phase. In order to be a little more practical, we take in consideration the last phase and in this one the classification and solving cost can reach \$ 10,000.00 [2]. The more expensive part of this process is called bug *triaging* phase, in which the bug was previously labeled, by the classification teams, and it is discussed between them in order to gain shared label classification, only at this point its sends to the right developer to be solved. Our work goal points to remove as much as possible this cost.

In addition, the cost increases when possible errors in the classification process are

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



Closed Bug 1603280 Opened 9 months ago Closed 9 months ago

division by zero in src/gfx/layers/apz/src/AsyncPanZoomController.cpp:3377

Categories

Product: Core
Component: Panning and Zooming
Type: defect
Priority: *Not set* Severity: normal

Tracking (bug RESOLVED as FIXED for Firefox 73)

People (Reporter: tsmith, Assigned: botond)

References (Blocks 1 open bug)

Details

Votes: 0

Attachments

Bug 1603280 - Guard against division-by-zero in AsyncPanZoomController::ComputePLPPI(). r=tnikkel
9 months ago **Botond Ballo [:botond]**
47 bytes, text/x-phabricator-request

Tyson Smith [:tsmith]
Description • 9 months ago

Found with m-c 20191211-b823b005f00e
This is triggered with an UBSan build while running gtests. To enable this check add the following to your mozconfig:

```
ac_add_options --enable-address-sanitizer
ac_add_options --enable-undefined-sanitizer="float-divide-by-zero"
ac_add_options --disable-jemalloc
```

```
[ RUN      ] APZCPinchGestureDetectorTester.Panning_TwoFingerFling_ZoomDisabled
src/objdir-ff-ubsan/dist/include/mozilla/gfx/BasePoint.h:77:48: runtime error: division by zero
    #0 0x7f347406c974 in mozilla::gfx::BasePoint<float, mozilla::gfx::PointTyped<mozilla::ParentLayerPixel, f
    #1 0x7f347406c2a3 in mozilla::layers::AsyncPanZoomController::ComputePLPPI(mozilla::gfx::PointTyped<mozil
    #2 0x7f3474021aed in mozilla::layers::AsyncPanZoomController::AttemptFling(mozilla::layers::FlingHandoffS
    #3 0x7f34740206ff in mozilla::layers::APZCTreeManager::DispatchFling(mozilla::layers::AsyncPanZoomControl
    #4 0x7f3474059e70 in mozilla::layers::AsyncPanZoomController::HandleEndOfPan() src/gfx/layers/apz/src/Asy
    #5 0x7f3474056720 in mozilla::layers::AsyncPanZoomController::OnScaleEnd(mozilla::PinchGestureInput const
    #6 0x7f34740527fe in mozilla::layers::AsyncPanZoomController::HandleGestureEvent(mozilla::InputData const
    #7 0x7f347416f3bb in mozilla::layers::GestureEventListener::HandleInputTouchEnd() src/gfx/layers/apz/src/
    #8 0x7f347416cbd0 in mozilla::layers::GestureEventListener::HandleInputEvent(mozilla::MultiTouchInput con
    #9 0x7f347404c9a0 in mozilla::layers::AsyncPanZoomController::HandleInputEvent(mozilla::InputData const&,
    #10 0x7f347418640c in mozilla::layers::InputQueue::ProcessQueue() src/gfx/layers/apz/src/InputQueue.cpp:7
    #11 0x7f3474183ea6 in mozilla::layers::InputQueue::ReceiveTouchInput(RefPtr<mozilla::layers::AsyncPanZoom
    #12 0x7f34741833ba in mozilla::layers::InputQueue::ReceiveInputEvent(RefPtr<mozilla::layers::AsyncPanZoom
    #13 0x7f346fc80d95 in TestAsyncPanZoomController::ReceiveInputEvent(mozilla::InputData const&, unsigned l
    #14 0x7f346fc92548 in void APZCTesterBase::PinchWithTouchInput<TestAsyncPanZoomController>(RefPtr<TestAsy
    #15 0x7f346fc29f18 in APZCPinchGestureDetectorTester_Panning_TwoFingerFling_ZoomDisabled_Test::TestBody()
    #16 0x7f346fb94d5f in testing::Test::Run() src/testing/gtest/gtest/src/gtest.cc:2519:5
    #17 0x7f346fb95d86 in testing::TestInfo::Run() src/testing/gtest/gtest/src/gtest.cc:2695:11
    #18 0x7f346fb966da in testing::TestCase::Run() src/testing/gtest/gtest/src/gtest.cc:2813:28
    #19 0x7f346fba499b in testing::internal::UnitTestImpl::RunAllTests() src/testing/gtest/gtest/src/gtest.cc
    #20 0x7f346fba43f4 in testing::UnitTest::Run() src/testing/gtest/gtest/src/gtest.cc:4788:10
    #21 0x7f346fbe54dc in mozilla::RunGTestFunc(int*, char**) src/testing/gtest/mozilla/GTestRunner.cpp:158:1
    #22 0x7f347cdd59ed in XREMain::XRE_mainStartup(bool*) src/toolkit/xre/nsAppRunner.cpp:3764:16
    #23 0x7f347cddeacb in XREMain::XRE_main(int, char**, mozilla::BootstrapConfig const&) src/toolkit/xre/nsA
    #24 0x7f347cddf6c3 in XRE_main(int, char**, mozilla::BootstrapConfig const&) src/toolkit/xre/nsAppRunner.
    #25 0x561442105da2 in do_main(int, char**, char**) src/browser/app/nsBrowserApp.cpp:217:22
```

Botond Ballo [:botond]
Comment 1 • 9 months ago

(In reply to Tyson Smith [:tsmith] from ~~comment #0~~)

> Found with m-c 20191211-b823b005f00e
> [...]
>
> ```
>     #1 0x7f347406c2a3 in mozilla::layers::AsyncPanZoomController::ComputePLPPI(mozilla::gfx::PointTyped<m
> ```

The code on this line is:

```
  aDirection = aDirection / aDirection.Length();
```

Botond Ballo [:botond]
Comment 2 • 9 months ago

This check should guard against the velocity being zero. However, some tests set `apz_fling_min_velocity_threshold()` to zero to make it easier simulate flings in test code, thus defeating the check.

Botond Ballo [:botond]
Updated • 9 months ago

Assignee: nobody → botond

Botond Ballo [:botond]
Comment 3 • 9 months ago

Attached file Bug 1603280 - Guard against division-by-zero in AsyncPanZoomController::ComputePLPPI(). r=tnikkel — Details

Pulsebot
Comment 4 • 9 months ago

```
Pushed by bballo@mozilla.com:
https://hg.mozilla.org/integration/autoland/rev/fd999a06e1e9
Guard against division-by-zero in AsyncPanZoomController::ComputePLPPI(). r=tnikkel
```

Dorel Luca [:dluca]
Comment 5 • 9 months ago

https://hg.mozilla.org/mozilla-central/rev/fd999a06e1e9

Status: NEW → RESOLVED
Closed: 9 months ago
status-firefox73: affected → fixed
Resolution: --- → FIXED
Target Milestone: --- → mozilla73

Figure 1.5: *ex of bug report*

considered.In the case that a bug is assigned to a wrong solver, the developers will be asked to fix it also without the needed skills and this can be translated in a waste of time and resources.

1.3 Introduction to bugs taxonomy

The first goal of this work is to produce a correct taxonomy2.2 for the bugs, that will be taken in analysis.

In software testing, the taxonomy aims to define the feature categories and group up the largest possible number of issue in each category. In our work, we set as a starting point, as it is suggested to help the selection of the categorization list, a previous work (Catolino, Palomba, Zaidman, & Ferrucci, 2019), and from that our work is pointed to figure out some defined macro groups and after that we define some different sub category related to the general one. We work closely with the tool of Mozilla Bugzilla[3], that contains all the bugs reported. With the analysis of different part of the bug report, i.e. the title name and the tags, we have continued the brainstorming and discussion concerning the nomenclature to adopt.

1.4 Goals

The goal of this thesis is to analyze and provide a valid tool for the solution of the bug classification. The nature of the topic indicate that the use of the machine learning approach in order to solve this problem.

Our approach is based on a tool, that extracts the most significant information from a bug report and the algorithm will returns the most probable classes for the bug taken in analysis. The aim will be to obtain a model that can assign different categories for a single instance taken in consideration.

The focus will be on the automation of the presented problem, to permit a important cost reduction in the bug classification approach, and in the future this work can be expanded to implement inside the model not only the classification phase, but also the assignment to the expert to work on the issue solution.

[3]Bugzilla

1.5 State of the art 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 tasks. This section will provide an overview of the previous studies and existing tools for 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, i.e., 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 classification, 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 (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 algorithm used as a starting point for this thesis was *BugBug*[4], a program developed by Marco Castelluccio[5] and other Mozilla developers in order to get bugs in front of the right Firefox engineers. As previously discussed, it is possible to decrease the time consumption to fix a new software bug, if the new one are quickly triage and classify by the owners.
This tool was the follow up work of a previous project that used a technique to differentiate between bug reports and feature requests.

This project lays the foundation in over twenty years of bugs that have been studied and review by the Mozilla community, the Mozillians[6], and those are used

[4]https://hacks.mozilla.org/2019/04/teaching-machines-to-triage-firefox-bugs/

[5]Marco Castelluccio

[6]Mozilla community

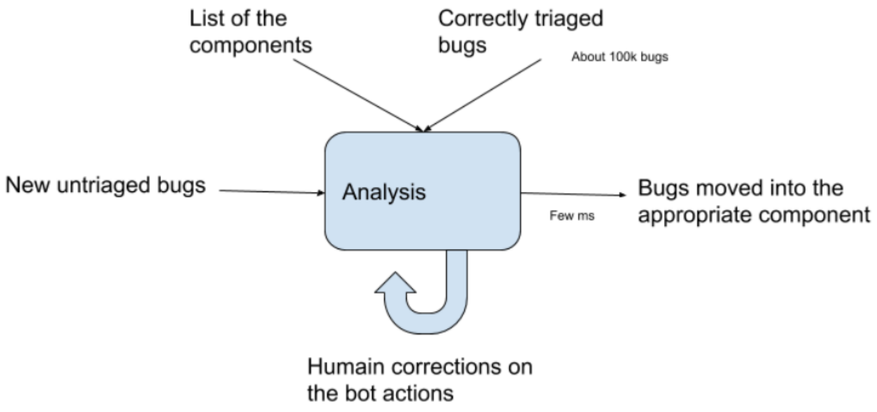


Figure 1.6: *BugBug* high-level training and operations

to improve the categorization and the overall work around the bug report.

In order to train the model using the same amount of information it would have during real operation, a "roll back" of the bug report to the time it was originally filed was performed. In this way any change to the bug after triage has been removed, indeed these information are inaccessible during real operation.
Also the taxonomy has been modified since in the past 2 years, out of 396 components, only 225 components had more than 49 bugs. In more details, 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 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

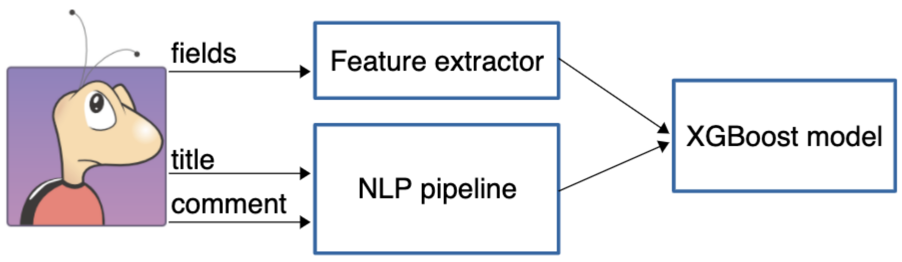


Figure 1.7: *BugBug* high-level model

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 BugBug's algorithm has starting point. Some changes were performed, especially in the feature selection component and in the taxonomy considered for the classification.

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 two approaches followed in our work, OneVsRest and Binary classifier;
- Chapter 4 covers the result obtained by all the experiments done;
- Chapter 5 presents our conclusions and suggestions for possible future works.

Chapter 2

Architecture and design

In order to help with the software bug classification and assignment process to the correct developer, we have developed our thesis work. Our tool is written in Python and it can work in two different mode, that can be managed on the fly passing the desired arguments:

- OneVsRest Classifier, it exploits the multiple label and class classifier and it is possible to execute a specific set of labels, i.e. there are two label sets, the first one, that is also the default one, it is composed by all the labels collected in the macro sets, the second is a specification for the first one, in fact this contains the previous and add the more specific labels in each class (Section 2.2)

- Binary Classification, it works as a simple binary classifier, where the chosen label is treated as the positive one and its against all the other joined together. Also in this use mode, its possible to execute the model with the a desired label from one of both the sets (Section 2.2)

The configuration is chosen using the arguments passed to the algorithm.
In particular, we have 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.

2. A **Column Transformer**, that comes from the sci-kit learn *"sklearn"* library [1] and allows to apply transformers to column array or to a pandas [2] DataFrame. 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 most important component, is the one responsible for the classification of the bug. It receives as input the output of the pipeline and its output is a model.

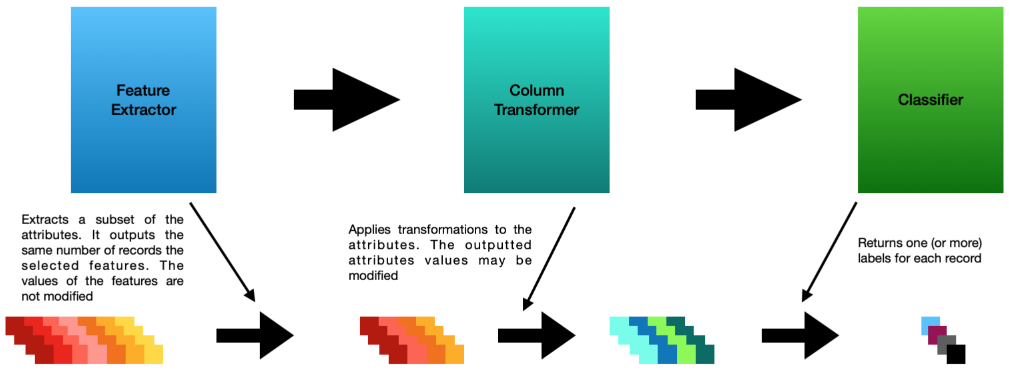

Feature Extractor
Column Transformer
Classifier
Extracts a subset of the attributes. It outputs the same number of records the selected features. The values of the features are not modified
Applies transformations to the attributes. The outputted attributes values may be modified
Returns one (or more) labels for each record

Figure 2.1: High level view of the basic model

2.1 Bug's Analysis

In order to reach our thesis goal, we started studying and analyzing how a human can classify in the right way a bug report. Our work followed again the footsteps of a previous paper(2019).

[1]sci-kit learn

[2]pandas

After the study over the classification method, we have split in two separated groups, in order to make the triage over a set of incremental bug report. Initially we started with a small amount of elements and every iteration we increased the number. At the end we had classified 250 bug reports (Section 2.3), and this was used as data set for the first analysis base to test the model prototype. This phase and also the following are characterized by the weekly meeting. This routine was done in order to keep up to date all the members in this work and also it is used to obtain feedback over the work done. At the end of the iteration, the data set was updated with the analyzed bug reports. After the last iteration it was updated the data set with the bug report previously classified in this paper[3]

Summing up briefly, the initial part is re presentable through a module, i.e. a sequence of actions, that are iterate different times. There are the assignment with the new bug list to classify manually, during the classification is presented some time spot where there are described the proposed classification and one is chosen, some time it was difficult to find a common classification; in those cases, during the meeting kept on Monday, those were discussed with the all members team, in order to find a right classification for the unsure bug reports.

2.1.1 Proposed metrics

At the end of each iteration, and before each meeting, we computed a value that measured the goodness of our classification; for this reason we used the *Cohen's Kappa* coefficient (Section 2.1.2), which measures the degree of accuracy and reliability in a statistical classification. Another relevant metrics, that we used to evaluate the goodness of the model, were the precision and accuracy values.

2.1.2 The *Cohen's Kappa*

The *Cohen's kappa* coefficient is a valid way to express the level of agreement between two judges, i.e. the two human that in initial part of this work had classify the bugs. 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

[3]Understanding, Characterizing, and Classifying Bug Types

exclusive categories[4]. This statistic was introduced by Jacob Cohen in the journal Educational and Psychological Measurement in 1960.

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

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 we refer to the following guidelines (see 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's possible that kappa is negative. This means that the two observers agreed less than would be expected just by chance.

In our study, this coefficient value increased every time regard the previous iterations, this is an interesting analysis point where the two judges after some iterations had more experience and confidence with the classification topic. The latest iterations had an higher agreement coefficient, but also a shorter needed time to perform the individual classification. Another measure considered was the percentage of agreement, ie. the value of p_0 in the *Coehn's kappa* equation. The following table shows all the element computed for each iteration.

[4] I Do Statistics

N. of Iteration	Po	Py	Pn	Kappa	Agreement
1	0.5671641	0.3653374	0.1563822	0.09501630	56.72%
2	0.6356589	0.3797848	0.1472267	0.2297039	63.57%
3	0.6774193	0.3625563	0.1582552	0.3268186	67.74%
4	0.6643598	0.3813412	0.1460471	0.2898183	66.44%

Table 2.1: Statistics for the Cohen's Kappa coefficient

2.2 Bugs 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.
Th starting point of our analyses was the work of our colleagues Catolino Gemma and Palomba Fabio (2019), who defined the bug's taxonomy in full details in their paper and presented later.

2.2.1 *Categories and Sub-Categories*

In this section the taxonomy used 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.

  Example summary.

  "Properly promote unexpected plugin wmode values to direct on Windows"

  [Bugzilla] - Bug report: 1620453

  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.

Example summary.

"Firefox 72 no longer loads Shockwave Flash. (after setting plugin.load_flash_ only pref is false)"

[Bugzilla] - Bug report: 1609257

- **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.

Example summary.

"VoiceOver doesn't announce clickable item" in the *Disability Access APIs* categories

[Bugzilla] - Bug report: 1616679

- **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.

Example summary.

"Enable additional permission for a specific bugzilla account"

[Bugzilla] - Bug report: 78453

- **Incompatibility:** refers to generic errors that do not belong to any of the categories above.

Example summary.

"CloseTab is not received when using deprecated TabDelegate"

[Bugzilla] - Bug report: 1620364

- **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.

> Example summary.
>
> *"TypeError: IndexedDB: clear() can not access property"*
>
> [Bugzilla] - Bug report: 1623481

- **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.

  > Example summary.
  >
  > *"Update verify failures in 74.0 build1"*
  >
  > [Bugzilla] - Bug report: 1619469

  Subcategories:

  - **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.

    > Example summary.
    >
    > *"../test/browser/browser_ aboutprofiling-features.js | Uncaught exception - NS_ ERROR_ ILLEGAL_ VALUE"*
    >
    > [Bugzilla] - Bug report: 1606082

  - **Compile:** compiling errors.

    > Example summary.
    >
    > *"Fix up some declarations used in bug 1512471 that break when chunking changes (or presumably non-unified builds)"*
    >
    > [Bugzilla] - Bug report: 1622016

- **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.

  Example summary.

  "Overlapping content on about:networking page with reduced width"

  [Bugzilla] - Bug report: 1620281

- **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.

  Example summary.

  "BroadcastChannel receives messages in onmessage after channel has been closed"

  [Bugzilla] - Bug report: 1622124

- **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.

  Example summary.

  "GeckoView re-sets some preferences unnecessarily"

  [Bugzilla] - Bug report: 1620105

- **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.

> Example summary.
>
> *"about:downloads shows an XML error"*
>
> [Bugzilla] - Bug report: 1609898

Subcategories:

- **Crash:**

  > Example summary.
  >
  > *"Prevent early crashes in non-content/plugin processes from crashing the main process"*
  >
  > [Bugzilla] - Bug report: 1616262

- **Hang:**

  > Example summary.
  >
  > *"repeating . and a letter (s) causes firefox to hang in browser"*
  >
  > [Bugzilla] - Bug report: 1603316

- **Incorrect Rendering:**

  > Example summary.
  >
  > *"synthetic bold on color emoji looks bad"*
  >
  > [Bugzilla] - Bug report: 1600470

- **Wrong Functionality:**

  > Example summary.
  >
  > *"Copy Image fails if image open in other than initial tab"*
  >
  > [Bugzilla] - Bug report: 1604003

- **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.

  > Example summary.
  >
  > *"Use MOZ_ WIDGET_ ANDROID instead of ANDROID for the enterprise roots for geckoview"*
  >
  > [Bugzilla] - Bug report: 1630031

In this classification taxonomy, it is possible to see thanks to the examples provided in the previous list, that when there is a subcategory label, this one can be defined also with the general one. In our taxonomy the general labels can be used also in the analysis with the subcategories classes, because we decided as assumption that a specific bug report can be identified at least with one label from the *general* classes otherwise there is the possibility the a issue do not have a right classification in the subcategory and it is labeled with the general class.

2.3 Building the Data Set

At the end of the first phase, based on the analysis and brainstorming regarding the taxonomy, we focused on the second part of our work: build a data set. All the bug reports taken in consideration during this phases were already resolved[5] bugs, taken from the *Bugzilla* database.

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

2.3.1 Bug Report Analysis

Each bug was considered individually, after the classification and triaging it was inserted in the data set. 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.

There are some bug reports, that are always self explained from the title, i.e. the classes like Security and Performance can also been individuated by the tag labels.

When is not possible to detect the correct classification with only the title and the tags, it possible to reading the comment section where is present the discussion between the solver assigned to the issue. Thanks to their insights 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.

In the case that it is not already classified the bug with the steps before, there one more resource, at the end of the report it is possible to find a section that would link to the code. Sometimes the code can be the key to label the bug to the correct class.

2.3.2 Batch Classification

The classification phase is divided in four batch, each of those moment was focused on a different set of issue reports, that needed to be classified. All the intermediate steps had to be completed by two person:

1. each one separately analyzed and tried to classify with one or more labels the bug;

2. then them discuss the two results for a specific bug, and they checked the meaning for a specific decision for the classification, and then is released the bug with the best label or labels;

3. those two step are repeated for each bug in the bug set;

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

2.3.3 The Data Set

After classifying all the batch, removing the duplicate bugs and the one that weren't bugs [6], the data set contained 181 bugs.
Since 181 is not enough for a good analysis, the bug reports classified previously in this article(2019) were added to the data set, reaching a total of 243 bugs. In the above mentioned work there are a number of bugs coming from different issue-tracking system, i.e. there are bug reports from *Apache*, *Mozilla* and *Eclipse*,but we only kept the bugs related to the Bugzilla system and the total amount of bugs[7] is 63.

The data set considering the general labels was composed of:

- 33 API
- 10 Database-related
- 42 Development
- 49 GUI-related
- 8 Network Usage
- 3 Performance
- 131 Program Anomaly
- 10 Security

while the classification using the specific sub-category labels was:

- 11 Add-on or plug-in incompatibility
- 2 Web incompatibility
- 1 Permission-Deprecation issue

[6]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

[7]Those bugs were previously classified with a different taxonomy, we have reconverted the classification according to the new taxonomy

- 6 Incompatibility
- 37 Test code
- 5 Compile
- 34 Crash
- 3 Hang
- 9 Incorrect Rendering
- 60 Wrong Functionality

as its clear by the numbers some the bug were only classified with a general label, due to lack of information to select a specific sub-category.

The entire data set will be used for training and validation of the model. In more details it will be divided in two sets: a training set of 243 records and a test set of 8, which means the former is 90% of the data set while the latter is the remaining 10%. Of course the generation of the two sets is performed randomly.

The classes population is highly unbalanced, going from 131 bugs for *Program Anomaly* down to 10 for *Security*, considering the general labels. Indeed this is 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, ie. a model with a low accuracy on the test set.

During our experiments we have added 100 items for each of these three classes:

- Performance
- Program Anomaly
- Security

this was done in order to fix the unbalance of the data set, the new elements were automatically classified by a script that which only extrapolates these specific classes from *Bugzilla* using keywords.

Chapter 3

Experiment design

3.1 The Machine Learning Model

This chapter presents the chosen machine learning model with the related advantages, disadvantages and the main part regards the two configurations used.

The aim of our work was to train a model on previously labeled bugs, classified by two humans (Chapter 2.3), and then to use that model to classify untriaged[1] bugs.

The presented work started, as previously mentioned, with the Bugbug[2] project analysis. Initially we focused, in particular, over the structure comprehension. As starting point we used the bugtype model[3] available in the Bugbug GitHub repository. The model, that were built up from those general premises, is subdivided by two different possible use modes:

- *OneVsRest*, this is a classifier where all the classes are taken one by one and there are produced different classifiers one for each class. This process can assign more than one label to a single instance.
- *Binary*, this is a classifier that considers only one class at time and the considered label is placed against the others, those are considered as a single

[1]Bugs that have to be labeled

[2]Bugbug

[3]Bugtype model - Bugbug section

indistinguishable instance. This process can assign only one label, in particular the classifier specifies if the considered class is represented by the instance or not.

3.1.1 BugBug

Bugbug is a project thought to help with bug and quality management, and other software engineering tasks using different techniques from the leveraging machine learning. All the data inside Bugzilla[4] is used to track anything, from the feature request to hardware/software malfunction passing through different bug categories. Bugbug has a specific area of competence, that is related only to the categories of bug[5] and the fake-bug[6]. The main reasons, why Bugbug works in this direction, are the quality analysis of a project, this can be used to measure different statistics over the overall release cycle, and the bug prediction, that can use the development history in order to identify a possible changes to solve the issue.

3.1.2 OneVsRest Classifier

We started ours work over the bug type classifier and we build a new model, that takes in consideration the studied taxonomy (Chapter 2.2) and uses as input a bug set classified by the team members. During the model development some bug sets are add to the analyzed pool, and in a couple of iteration the bug number arrives at the value of 253. The peculiarity of this classifier is characterised by two label sets, all the bugs are described by two classification, a generic and a specific one. This second classification extends the generic one, because a generic bug must be classified at least with one label from the generic set, but it is not mandatory that the same bug has a label from the specific set, for that reason when we decided to work also with the specific labels, we merged the two sets.

The classifier chosen was the OneVsRest[7], this performs a multiple class classification, because we had respectively for each label set: 8 labels for the generic one

[4]Bugzilla

[5]Bugs that actually are bugs

[6]Bugs that are not actually bugs

[7]OneVsRest Classifier

and 19, the 8 of the generals and the other new 11 labels of the specific set, for the specific classification. This is also a multi label classifier, this means that there aren't constraint on how many classes a bug is assigned to. The classifier behaviour can be split in three parts. In the first one, one classifier is fitted over a single class, in the next step all the classifier are taken individually and the class is fitted against all the other classes. The last part each classifier is used to predict more than one labels for a single instance, this is made using a two dimensional matrix and it's fitted with the following relationship, is given a generic matrix cell as $[i, j]$, with the respectively symbols i is the sample and j is the label:

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

One of the most strength aspect is the possibility to gain knowledge about one class by analyzing the classifier related to this.

3.1.3 Binary Classifier

During ours work with the OneVsRest classifier we found that the overall accuracy is not too much high, for this reason we had implemented in our model the possibility to choose the classifier use mode, with the multiple label classification, that is explained in the previous section, and the binary classifier is able to analyze a single class at time with the relative characteristic features. In ours implementation we decided from the design of this second model part to keep as much as possible homogeneous with the already presented code model, for this reason it is possible to specify the desired class to study. The classification tasks is related with only two classes, the selected one is considered as the positive label for the bug analyzed. This classifier is defined by n independent random couples of $(X_i, Y_i) \in \chi \times \{0, 1\}$, where X represents the features and Y the labels. The goal of this algorithm is to build a rule that is able to predict the output Y given the input X, that rule is a function called classifier and its defined as $h : \chi \rightarrow \{0, 1\}$. There are different classifiers, and each one is much suitable for a specific data prediction with the specific input data features, in the next section there are some practical cases taken in account during this work thesis.

3.2 The Model

In machine learning the classification methods are considered instances of supervised learning, this is a machine learning task that creates a function able to map a given input to an output based on example input output pairs. The main goal is to build up a inferred function that can map new examples from the same space of the training data. It is also possible to generalize in order to classify unseen situation in the most "probable" way from the training data. The main difference with the unsupervised learning is the data identification, in the supervised approach the learning is based on the observation of the training data that were correctly identified previously. Meanwhile, the other one tries to group up the data into categories based on measure of the similarity or the distance.

In our work we started with an approach that is focused over the *XGBoost library* 3.2. The entire thesis project has been designed and developed in a modular way, this allowed us to perform different tests even parallel in order to find the best configuration of our model. The different experiments led us to generate different scenarios for the model, in fact you can run the program with the desired data set or you can specify the operating mode of it.

During our tests we implemented several applications including a grid search, that is able to find the best hyper parameters for our data set, and a script that from the *Bugbug* section that we have modified in order to extract *Bugzilla* bug reports using keywords and tags[8]. However, the main part of our model is definitely the choice of the classifier, and that is why we have performed several analysis and tests with different algorithms to classify bug reports in order to get a better prediction.

The following list is composed of the classification algorithms used in our work.

- **XGBoost**

[8]This script allows us to expand our data set with additional independently classified items, even with more than one label if the issue falls into more than one class.

It is a project called *eXtreme Gradient Boosting*[9], this is an open-source library designed to have high efficiently and flexibility. It is possible to use this in a wide range of applications, from the regression to the classification and the forecasting problems. The others interest points are the portability trough the different operating systems, the active support for the main programming languages, i.e. *C++*, *Python* and *Java*, and also the cloud integration.

This project is based on the *Gradient Boosting*[10] technique. This provides a forecasting model based on a decision trees composed by a set of weak forecasting models.

In our work this is the first approach to predict the data, but for the lower accuracy results we decided to try different approach in order to improve the performances.

- **Naive Bayes**

  This is a set of methods, that apply the *Bayes' theorem*[11] with the assumption that every pair of features has the conditional independence. The considered assumption is a *naive* approach that considers the conditional independence between all the pairs for the given class.

  The following probability is the *Bayes' theorem*, where the y represent the values and from x_1 to x_n the features.

$$P(y|x_1,..,x_n) = \frac{P(y)P(x_1,..,x_n|y)}{P(x_1,..,x_n)}$$

  The independence condition is expressed like a probability of occurring together is the product of their individual probabilities.

$$P(x_i|y,x_1,..,x_i-1,x_i+1,..,x_n) = P(x_i|y)$$

  In order to define the $P(X_i|y)$, and since that the $P(x_1,..,x_n)$ is constant for

[9] XGBoost

[10] Gradient Boosting

[11]

the given input, it is possible to obtain with the MAP[12] use.

The different classifiers differ for the assumption over the data distribution. In our tests, we founded as the best one that fit our data with the higher accuracy: the *Complement Naive Bayes*[13]. This algorithm is specially suited for the imbalanced data sets.The CNB[15] uses statistics from the complement of each class to compute the model's weights.

- **Support Vector Machines**

  The SVM[16] are a set of supervised learning methods, that provide methods able to perform classification, regression or outliers detection. The most significant point of strength of those are the effective in high dimensional spaces and the possibility to choose the preferred *kernel function*[17] in order to improve the quality of the classification. The counterbalancing of those advantages there is mainly the impossibility to compute directly probability estimates, and the steps made to compute those are expensive, because it is needed a five-fold cross validation.

  The SVM constructs an hyper-plane or a set of them in an high dimensional space, the best performances are obtained when the margin between the hyper-plane and the data is large, in the cases that the support vector cannot be able to separate all the training data, there are set some *boundaries* that allow a misleading point to be classified correctly. The classifier, that we had chose for our experiments, is the LinearSVC[18], a fastest implementation using the SVM, that make use of the hinge loss[19].

[12]Maximum A Posteriori, this is an estimate of an unknown quantity

[13]This classifier extends the classical one called *Multinomial Naive Bayes*[14], that is generally used for the text classification.

[15]The Complement Naive Bayes

[16]Support Vector Machines

[17]The kernel function is a set of mathematical operations, that is used to transform the data in the preferred form

[18]Linear Support Vector Classification

[19]The hinge loss is a loss function used to find the most larger margin of the classification

The statistical formulation can be expressed as the following formula.

$$\min_{w,b} \frac{1}{2} w^T w + C \sum_{i=1} \max(0, y_i(w^T \phi(x_i) + b))$$

This implementation offers to compute process the data as a binary problem or a multi-class one with the possibility to choose the penalty and the loss function for the given data. The *Linear-SVC* cannot implement different kernel from the *linear* one, because is not present the inner products between the samples, this mean also that it is not possible to apply the *kernel trick*.[20]

- **Nearest Neighbors**

  The *Nearest Neighbors* are a set of both supervised and unsupervised learning methods. The idea behind this models is to find the much larger number of similar training in the same area with different mathematics computations.

  During our experiments we had work with the *K-Nearest Neighbors*[21], and its type is the *instance-based learning*[22]. This approach assigns to the instance the label of the most populated class, this count is obtained the greater class in the K neighbors. The value K need to be choose with the knowledge that a lower value can increase the noise on the data, but the borders are more smooth. the There are different solution for the distance algorithm, and the way it is applied. We tried to use the following two algorithms:

  - *Brute Force*, this algorithm computes all the distance for a single element with all the other, with large data it becomes infeasible. The computational cost is $O[DN^2]$.
  - *Ball Tree*, with this algorithm the main goal is to reduce the computation time using a tree structure. This process works with the triaging over three points, i.e. considering A is distant from B, and this one is closest to C, the algorithm do not compute the distance between A and C. The

[20]The kernel trick is a mathematical operation that permits to reduces the complexity of finding the mapping function.

[21]Also called KNN, this algorithm is used in the pattern recognition for the classification of instance based over the distance and the proximity.

[22]The method saves the instances of the training data without creating a general internal model

computational cost is reduced to $O[D \log(N)]$.

3.2.1 Balancing Technique

In our thesis work, after working on the taxonomy, the composition of the data set and then the model, we focused on the improvement of this model. In particular, we performed some balancing operations of the data set within the model itself, because not all the classes of our set are populated in the same way, and the classes with fewer instances were penalized.

To balance the model we used the technique of applying different weights for each class, in order to have each class with the total weight equal among all. This is obtained through the following formula

$$w_{class_i} = \frac{n_{samples}}{n_{classes} * n_{occurrences_i}}$$

as shown in the previous one, you must perform for each class its own weight. This value obtained is the ratio of the total of the instances present within the model and the number of occurrences within the selected class for all the classes present.

3.2.2 Minimum matching threshold

In machine learning, there are a model called logistic regression[23] that is used to compute the probability that a certain instance can be classified as zero or one, in ours case this method is applied to predict the appearance of an instance to a determinate 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 append, or in ours case that the bug taken in consideration is inside a class of the set, or it's possible to convert the obtained value to a binary one. For example, a model that returns a prediction of 0.001, for a bug to stay inside a class, is predicting that it is very likely not inside the class taken in analysis, otherwise a value of 0.999 means that there are high chances that the bug is inside the class. The difficult point is to distinguish the middle cases. In order to map a regression value to a binary

[23]The logistic regression is a non-linear regression model, that it is used when the dependent variable is a dichotomous variable.

identifier, it is mandatory define a classification threshold. This value indicates the threshold above which is possible to identify an instance with a specific class with a great margin of confidence. It is necessary the introduction of this new parameter, because the classification thresholds are problem dependent and this dependency affects the values that will be tune, for this reason the threshold is not set statically at the value 0.5. In our work, we have set a range of possible threshold values in order to find the best one through different evaluations of the results obtained from these.

Chapter 4

Results

This thesis work aims to offer a tool able to classify a bug report with one or multiple labels in a fast way and this generates a lower cost related to the specific issue report and allowing the experts to focus on the bug removal phase. But the results, during the first approach, is not as higher as expected, in particular the accuracy is 29.57%, and this value shows that there is error prone in some specific bug report categories. After several attempts and improvements, such as the one presented in the data set, through *XGBoost* obtains an accuracy of 40%.

For this reason we decided to temporarily remove *XGBoost* and try to focus on other classifiers. Initially we tried with the *Naive Bayer*, which gave slightly better results than previous texts with *XGBoost*, generating a total accuracy of 33.09%, and then with an *SVM* approach using *SVC*. The results this time were still slightly better than the *Bayer classifier* up to a value of 38.69%.

For the sake of completeness we tried an even more statistical approach and then tried to apply the *KNN*, which however did not show very good results, as it did not exceed the threshold of 27.89%, for this reason this classifier will no longer be taken into account during our analysis.

4.1 Experiments

The model is composed by two different way of use and those run over the same configuration of the environment, the difference between the two modes is mainly the classifier and the focus over the statistics taken in account.

In order to obtain an higher classification accuracy, we performed several runs with different configuration. The initially analysis were made with the *XGBoost Classifier*3.2, and we have focused over the features. In a generic bug report, the title is the richest resource of information in order to classify the issue, its is followed by the description with the related tags. After that we added also the comments as a features to keep in account and in the next experiments we tried to compare the feature goodness extracted by the different comments separating the first one from the others, in this way we tried to give more importance to this element.

The great issue, that we found, is into the distribution. The classes had different element amount and the overall distribution resulted unbalanced, for this reason we added as an additional transformation in some experiments to balance the classes with lesser instances.3.2.1 The different weight that each class has within the model is more visible in the configuration with the *subcategory* labels, from a certain moment the various tests will focus on the setup of the *generic* labels to provide a more stable model.

As mentioned earlier, all these experiments were led using the same data set, that was split in train and test set, with a ratio 90% and 10%. The results reported below are the most significant statistics extrapolated from the various experiments carried out, for this reason will not be presented all the combinations of the values obtained, but will be provided an overview for all the various tests and finally will be presented a full version of the most relevant statistics for the best configuration of the model.

An important part of the work was focused on finding the best classifier to adopt, in order to obtain better values the *grid-search* method was used to maximize accuracy with the best hyper-parameters. This was obtained by providing a classifier and a list of hyper-parameters to set, with specific operating ranges. This is made for all the four classifier presented in the chapter before 3.2. We will initially analyze separately the two ways[1] of use of the classifier within the implemented model.

4.1.1 Results *OneVsRest* Classifier

Here the results obtained using the *OneVsRest* classifier (Section 3.1.2).

As previously described, having a very low accuracy with the *XGBoost classifier*[2], we decided to use different classifiers and to parameterize them in order to obtain the highest possible accuracy. The following tables[3] shows the evolution[4] of our model through the different classifiers[5] and also through subsequent updates of the data set[6].

Tuning	N. of element	Accuracy	Roc&Auc	Precision
No	181	44.44%	0.54	0.31
	243	36.23%	0.54	0.28
	546	58%	0.66	0.54
Yes	181	46%	0.55	0.31
	243	58%	0.66	0.54
	546	59.47%	0.68	0.59

Table 4.1: Summary of OneVsRest approach statistics with the KNN classifier

Now the classifier that has provided the best results is SVC[7] and it is presented in its entire path[8]. First at all, we analyzed the possible kernels we could have used, the best one turned out to be the linear. The following list shows the respective accuracy and highlights:

- *rbf*, this kernel implements the *kernel trick*, see the section 3.2, and produce a stable accuracy of 58% with the overall precision equals to 0.48. The partial

[1]The OneVsRest Classifier (Section 3.1.2) and the Binary Classifier (Section 3.1.3)

[2]The accuracy obtained with this model is less than 30%.

[3]Those two tables represent a summary of the experiments with the KNN and the Naive Bayes Classifier.

[4]The Naive Bayes has a set of different classifier, in this brief summary is reported only the most significant one, the Complement Naive Bayes.

[5]Classifiers that did not generate expected results and were discarded.

[6]The data set evolution is expressed by the instance amount in the following table.

[7]Support Vector Machine Classifier

[8]The experiments performed before the completion of the date set have been excluded.

Tuning	N. of element	Accuracy	Roc&Auc	Precision
No	181	41.97%	0.50	0.7
	243	33%	0.51	0.7
	546	35.43%	0.57	0.37
Yes	181	49.38%	0.56	0.27
	243	58%	0.66	0.54
	546	65.98%	0.71	0.66

Table 4.2: Summary of OneVsRest approach statistics with the Naive Bayes classifier

results[9] show that the majority of classes, except for *Performance*, *Security* and *Program Anomaly* issue, are not represented and are not characterized.

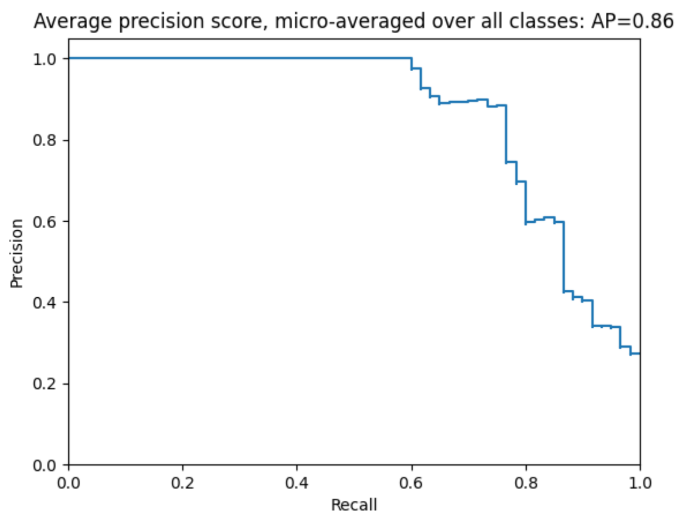


Figure 4.1: The average precision score obtained with the *rbf kernel*

- *poly*, this one allows the learning of non-linear models, representing the similarity of the vectors to a characteristic space in reference to the original features. the accuracy obtained is quite low, as well as the precision, that is 0.45. This kernel function like the previous one suffers from a bad classification due to the unbalance of the data and therefore the inability of the model to characterize the majority of classes.

[9]These partial results are all similar in these kernel functions, since the majority of the values are equal to 0, for this reason there are not fully reported.

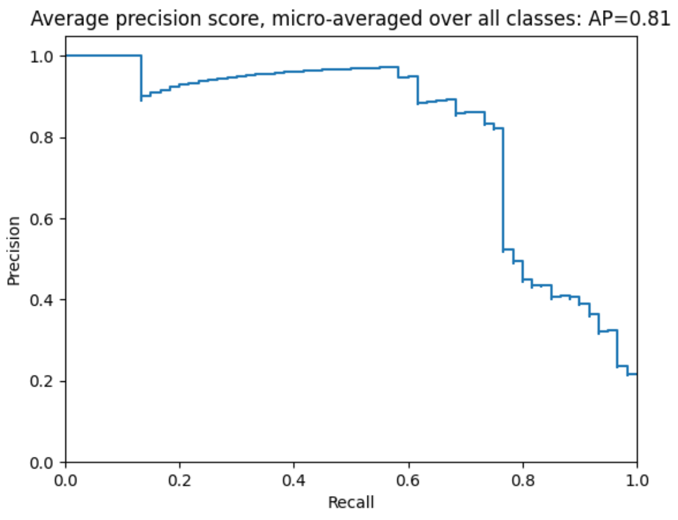


Figure 4.2: The average precision score obtained with the *poly kernel*

- *sigmoid*, this kernel function is similar to the sigmoid function in logistic regression, and the accuracy obtained is equals to 42.97% with the precision of 0.31. This is the worst of all the functions analyzed, for this reason in this case there are several classes that are not represented[10].

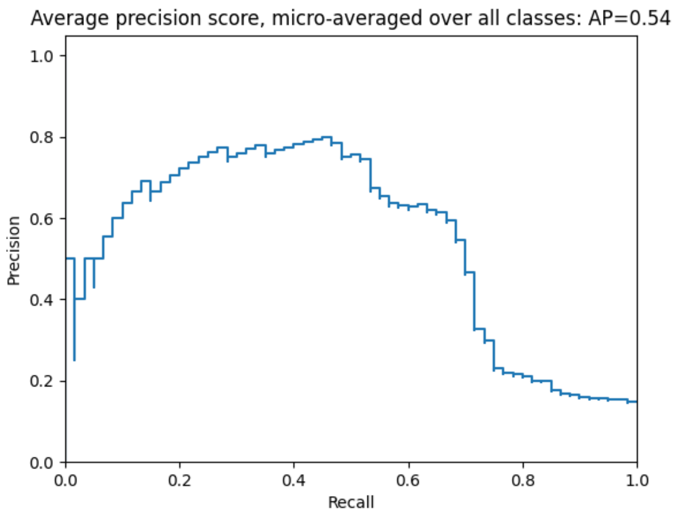


Figure 4.3: The average precision score obtained with the *sigmoid kernel*

[10]This trend is the same as for the two cases mentioned above.

- *linear*, this is the best kernel function, in fact it generates an accuracy of 73.32% and it has a precision of 0.74. In this case, only the *Network Usage issue* class isn't correctly represented in the model[11].

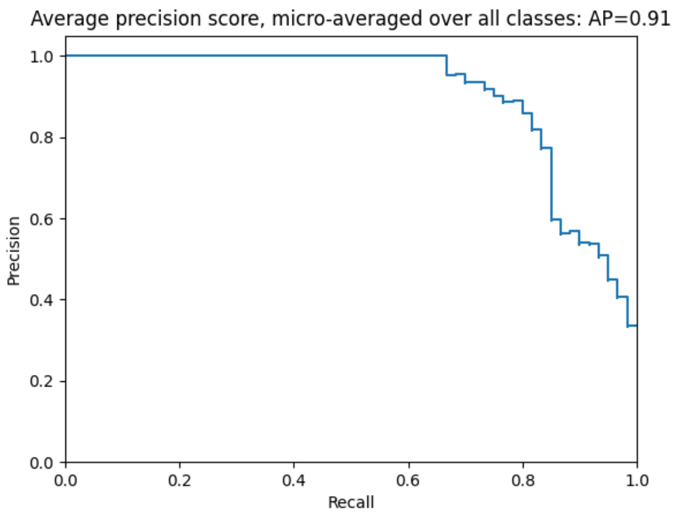


Figure 4.4: The average precision score obtained with the *linear kernel*

After analyzing the various functions of the kernel and finding the best one in the *linear*, we continued the study of the model only with a specific version, which is called *Linear SVC*[12].

In order to solve the problem related to the non-representativity of the class *Network Usage issue*, we have implemented balancing measures on the data set3.2.1. This has been done by calculating the weight that each class brings in the whole set.However, this measure has led to an overall improvement of the model, in terms of accuracy, but has not allowed the class in question to be classified correctly, so this problem can only be solved by adding other bug reports that also belong to the *Network Usage issue* class. From now on we will omit this class and focus on the remaining classes and the model in general.

After we have found a classifier that fits better over our data, and after cleaning

[11]This is because this class has too few elements.

[12]This is already presented in Section 3.2

up this data in order not to generate unbalanced situations between the different classes, we have performed a tune operation of the *Linear SVC* classifier. At this stage we have researched and established the best hyper-parameters for our model. In particular in the following list we show how we set the most relevant parameters:

- the loss function was chosen equal to the square of the hinge loss
- the search for optimization of the dual problem optimization
- the adjustment parameter is a parameter that serves as a degree of importance that is given to miss-classifications

All of these shrewdnesses have led to the formation of a model that has the following characteristics, always operating through the *OneVsRest* approach. The final accuracy is 73.11% with a global precision of 0.622, while the value of the overall log function has decreased to 6.3. In the following table the single classes are presented with the respective statistics.

Class	Precision	Recall	F1-score
API issue	0.36	0.40	0.38
Network Usage issue	0.00	0.00	0.00
Database-related issue	0.67	0.60	0.63
GUI-related issue	0.43	0.52	0.47
Performance issue	1.00	0.98	0.99
Security issue	0.99	0.93	0.96
Program Anomaly issue	0.82	0.85	0.83
Development issue	0.71	0.61	0.66

Table 4.3: The final statistics for the OneVsRest

4.1.2 Results *Binary* Classifier

The results of the *binary* classifier (Section 3.1.3). The analysis of the binary approach classifier will be presented only the best solution, because in section 4.1.1 the various experiments have already been shown, which did not produce a result as good as the *SVC* one.

All these experiments have an accuracy of over 50%. Analyzing in more detail with this mode of use we went each time to select a class to classify against all the other elements considered a single set. This analysis has served to study the single classes and to understand where to intervene and where there could be cases of errors within the model itself.

It was very important the attention we paid to the features, in fact several tests were carried out and from these we realized that the feature extraction module captured different parameters, but in particular it often happened that the most important feature was the article '*the*'. To solve this problem we have excluded from the extraction method features that were not important for the development of the model and that could potentially create over fitting situations[13] of the model.

Once the model modeling problem was solved we tried to apply the classifier with its settings that generated the best accuracy[14]. Below are the various results of each individual analysis for each class[15]. For the general label the first relevant experiment was done without using the balancing technique and the next with these technique, the results are in Table 4.4, while another experiment was led applying the same routines over the remaining labels, the *sub-categories* one, and the results are in the Table 4.5.

In this approach[16], using the model with balancing techniques is not always the best choice, as there are cases where the imbalance between the two classes[17] is excessive. It is also very important to note that especially in the label classification of subcategories, the classes are composed of very few elements, and this makes it difficult to generate the model. Generally again as in the *OneVsRest* results4.1.1, the best accuracy is obtained from generic classes and not from more specific ones. In the generic label only *Network Usage issue* turned out to be the weakest, just because it is the class with fewer elements than the others.

[13]Over fitting is created when the model is modeled not only on data features, but also on data noise.

[14]The reported results are computed over the last data set.

[15]The general classes are presented only once, and are therefore not reported in the collection of sub categories, because the values are the same from the moment that only the class is analyzed without the context of the set to which it belongs.

[16]The approach with the binary classifier

[17]The class examined and all the others gathered together.

Label	N. of Bugs	Balancing	Accuracy	Precision	Recall
API issue	33	No	63%	0.63	0.63
		Yes	65%	0.65	0.64
Development issue	42	No	78%	0.78	0.78
		Yes	75%	0.75	0.75
GUI-related issue	49	No	80%	0.82	0.80
		Yes	81%	0.82	0.81
Network Usage issue	8	No	41%	0.40	0.42
		Yes	50%	0.5	0.5
Performance issue	104	No	97%	0.97	0.97
		Yes	98%	0.98	0.98
Program Anomaly issue	232	No	83%	0.83	0.83
		Yes	85%	0.85	0.85
Security issue	111	No	97%	0.98	0.97
		Yes	96%	0.96	0.96

Table 4.4: Statistics for the general labels with the binary classifier

To summarize the results obtained, it is possible to see how classes with fewer elements have gained some more precision and accuracy in the *binary* version than the version calculated in *OneVsRest*, while classes such as *Security issue* or *Program Anomaly issue*, have seen a slight decreased the accuracy in the binary version.

Label	N. of Bugs	Balancing	Accuracy	Precision	Recall
Add-on or Plug-in Incomp.	11	No	90%	0.92	0.90
		Yes	90%	0.92	0.90
Compile	5	No	40%	0.38	0.4
		Yes	40%	0.38	0.4
Crash	135	No	95%	0.95	0.95
		Yes	95%	0.96	0.95
Hang	3	No	0.0	0.0	0.0
		Yes	75%	0.83	0.75
Incompatibility	6	No	60%	0.6	0.6
		Yes	70%	0.71	0.70
Incorrect Rendering	9	No	50%	0.5	0.5
		Yes	50%	0.5	0.5
Permission/Deprecation	111	No	0%	0.0	0.0
		Yes	0%	0.0	0.0
Test code	37	No	86%	0.86	0.85
		Yes	84%	0.84	0.84
Web Incompatibility	2	No	0.0	0.0	0.0
		Yes	50%	0.5	0.5
Wrong Functionality	60	No	73%	0.74	0.73
		Yes	70%	0.71	0.70

Table 4.5: Statistics for the subcategory labels with the binary classifier

Chapter 5

Conclusions

The bug classification task is really significance and expensive nowadays, a company can not afford to keep bugs in their software, in particular the much is dangerous the issue and the much is needed to resolved, i.e. the exposure of sensible data or the inefficiency program, that slows down the hardware performance. For this reason the automation of this task can help a lot the company to define the typology of the bug and to assign it to the right developers to solve the issue as soon as possible. Another important aspect is that the software will hardly be replaced within a few years, when those are adopted over a specific environment. In an other way, thanks to the modular structure of the tool, it is possible to update its for future uses also external this environment.
This can be achieved changing the *Bugbug*(Section 1.5.1) function that allow to read the bug report and then extract the features retained important. In second place it is also needed to update the data set composition, because our project is based and works with the bugs related to *Mozilla*[1].

With this work it is proposed a tool that will avoid useless work for the developers, that need to be focused on the bug fixing, and this model automate the classification process.

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

[1]We tried in a naive way to use also bug reports from different systems, but as we expected the accuracy went down a lot because the feature extraction module was not implemented except for bugs from Bugzilla.

biased model, since the classes are unbalanced and some classes in the taxonomy are never classified, like bugs related to API.

5.1 Summary of the results

In the previous chapter 4 were presented the two configuration related with this work. Each model can be executed more times with every time different hyper-parameters using a tuning library as help during the optimization process. Each test is made with the same train and test ratio between the train and the test set.

5.2 Future work

In this section, we show a set of possible points of view or tips, that can be done to increase the performance and improve the tool. Those suggestions can be aimed towards increasing its accuracy for all the classes and setting up a minimum threshold higher with the possibility of the feature addition.

- Add more instances to the data set. The data can be more homogeneous and have minor unbalanced between classes.You need to implement specific classes that have few elements, i.e. the *Network Usage issue* for the *general* labels, and also re-balance the more specific classes, that in our work have produced the worst results.
- Implement different modules, for reading and extracting features from different issue tracking systems different from *Bugzilla*, some examples can be *Apache* and *Eclipse.*
- Add before the test phase of the model a *evaluation phase*, that evaluates the goodness of the model just created and trained. This further phase needs a wider data set for the model that does not count many elements.
- Develop the last step of the automated work for this study, which is the direct assignment of the best developer to fix the bug report, because this face now remains the most critical and often also the most expensive one when it is done incorrectly.

- At the moment it is not foreseen to search for possible duplicates inside our data set, we have manually classified the bug reports, we could eliminate the duplicates beforehand, but in the future this work could be done by a module that will recognize the duplicate bugs.
- In our work we have considered only and always cases of bugs belonging to well-defined classes, but in reality there are often *fake bugs* or no-bugs[2], these need to be recognized and then classified as such.
- A further development could be *regression bug*, implementing a call back to the previous bug that generated the unexpected situation in the code after the last update. It is important to prevent this kind of bug[3], but in the case this is not possible, this must be have the highest priority to be solve.
- Will also be a possible future development, the prediction of a priority order for the allocation and resolution of bugs[4], giving more importance to bugs that affect the proper functioning of the software and those that undermine the security of the user, below come all the others according to the functionality of the problem, i.e. if a problem was only graphic, it would certainly have a lower priority than the resolution of a problem related to the drop in performance on a specific device.

[2]These are all those bug reports that do not present a problem that has occurred, but requests or different versions for the same.

[3]This kind of bugs erode the trust between the vendor and the customers, that can also leave the specific company when there are too much and too frequently bug of this category.

[4]A priority in the right sense, just as it is now when a new report is created.

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.